S D O S

APPLICATION PROGRAMMERS' GUIDE

4th Printing

TABLE OF CONTENTS

NOTICE

This manual describes Software Dynamics Operating System (SDOS) Version 1.1. Software Dynamics has carefully checked the information given in this manual, and it is believed to be entirely reliable. However, no responsibility is assumed for inaccuracies.

SD reserves the right to change the specifications without notice.

SDOS APPLICATION PROGRAMMERS' GUIDE

INTRODUCTION

This manual gives detailed information needed by programmers building programs to operate under SDOS 1.1. The reader should be familiar with SDOS concepts; the SDOS User's Guide provides the appropriate background.

This document presumes some familiarity on the part of the reader with assembly language coding for M68ØØ, M68Ø1 and M68Ø9 microprocessors. This knowledge is needed to understand fully the implications of the SDOS System Call (SYSCALL) interface and the rules about error propagation. Practical use of SDOS does not generally require assembly assembly language programming, as most programming is done in SD BASIC, which provides statements for performing SDOS System Calls.

This document covers three main areas:

SDOS SYSCALL structure and assembly language interface

Device Independent I/O - Concepts and device specific descriptions

SDOS File System Structure

DEVICE-INDEPENDENT I/O

SDOS allows user programs to view all disk files and I/O devices as being fundamentally the same, i.e., if one can perform an operation on a device of type x (say, LPT:), one can generally perform that same operation on a different device of type y.

Since disk files and devices are treated essentially identical, we will use file sometimes to mean device.

In this section, a conceptual model of how files/devices should act is presented (later sections describe in detail the system calls used to implement this model). SDOS is designed in such a way that disk files conform to this model very closely; exceptions will be noted later. Real devices such as line printers, CRT's, Digital-to-Analog converters, etc., are made to emulate this model as closely as possible via a device driver routine in the I/O package; the degree of closeness depends entirely on this driver. In many cases, it is not practical or appropriate for a device to match the desired model; this means that there are device-dependent (actually, driver-dependent) limitations on this device independence.

SDOS implements files for the purpose of storing and retrieving data. A file is assumed to consist of a sequential set of 8 bit data bytes, with the first byte being numbered zero, the second being number 1, the nth being numbered n-1. Each file has a size, which is equal to the number of bytes of data stored in the file. The data in a file can be read or written sequentially in variable-size blocks. If new data needs to be added to the end of a file, the file can be automatically extended. Commands exist to allow a file to be positioned to a specified byte position in preparation for a later read or write operation, thus providing random access. Data can be read or written in pure binary, or in ASCII (text) format.

A device is (usually) a physical piece of hardware capable of retrieving and storing data, converting data to/from printed form, etc. (some devices, such as the CLOCK:, are almost purely software). In many cases a device is treated as a file by SDOS. Some devices can actually store many separate data files (such as a disk device).

User programs communicate with files via mechanisms called "I/O channels". A channel remembers which file is being manipulated, and where in the file that the next data transfer should take place. Each user can have several I/O channels; typical SDOS systems allow eight I/O channels per user. I/O channels for a user are given numbers Ø to 255 maximum.

Virtually all operations on a file must be performed in conjunction with an I/O channel. An initial connection is established between a user-program specified I/O channel and a particular file by use of a SYSCALL:OPEN (or SYSCALL:CREATE).

All further operations on that file must specify the operation desired, and the I/O channel number associated with a file. Note that a particular file may be open on several I/O channels, thus causing interactions between what appear to be independent operations. The association between a channel and a file is broken with a SYSCALL:CLOSE operation; a channel on which this operation is the most recently executed valid operation is said to be CLOSED. No operations except OPEN or CREATE are valid on a closed I/O channel.

The I/O channel has associated with it several pieces of information: whether that channel is open or closed; the particular device driver which is responsible for that file; information selecting which file on that device is to be used; data selecting a position within that file; and a column count (next print position on a real or simulated printing device).

When a file is first opened, the position is reset to zero (beginning of the file). Each read or write operation on an I/O channel advances the position for that channel by the amount of data read/written. An End Of File condition is said to have occurred whenever the file position on a particular channel is equal or larger than the file size (in bytes). Note that two I/O channels open to the same file are not necessarily positioned to the same place within that file.

A column count is maintained for the purpose of "tabbing" (a text concept). This column count is zeroed whenever binary data (non-text) is read or written to a file, and adjusted to reflect the position along an imaginary typewriter line whenever textual data is copied to or from a file.

Operations prformed on files are done via SDOS System Calls (SYSCALLs). SYSCALLs specify an operation, a Write Buffer (containing data going to a file or to SDOS), a Reply Buffer (where data or status from SDOS is returned), a channel number and/or operation subcode, and a reply length (RPLEN).

Operations defined on files include, but are not limited to:

OPEN, CREATE, CLOSE, DELETE, RENAME, READA, READB, WRITEA, WRITEB, CONTROL, STATUS, POSITION, GETEOF, GETCOLCNT

Other operations are device-driver specific.

OPEN is intended to associate an I/O channel with a file (device) that already exists, for the purpose of reading (or updating) data in that file. Data-input only devices such as paper tape readers must be OPENed in order to read data. All devices can be OPENed so that the device type is easily read without knowing the kind of device being OPENed.

CREATE is intended to associate a file or device with an I/O channel which is to be used whenever an entirely new stream of data is to be written or stored. In particular, when a new disk file is needed, or data sent to an output-only device (such as a line printer) a CREATE should be performed. Some devices, like CRT's, which are both input and output, can be either OPENed or CREATEd.

CLOSE is used to break the association between a file and an I/O channel, and to cause the driver for the device on which that file resides to finish any operations on that file.

DELETE is used to delete (disk) files from devices that store multiple named files. Devices cannot be deleted. Once a file is deleted, it cannot be OPENed and its contents are permanently lost.

RENAME is used to change the name of a disk file, and is illegal when directed specifically at a device.

READA and WRITEA are used to read and write ASCII (textual) data. This is used to read data from consoles, print on line printers, etc. If a file has no more room for new data written, then the file is automatically expanded. A channel number must be given to select the desired file.

READB and WRITEB are used to read and write binary data to and from devices (data stored in a form convenient for the computer). An I/O channel number is required to select the desired file. Some devices, like Digital to Analog converters, can only perform Write Binary.

CONTROL operations are used to cause device-specific operations that do not fit into the above types of operations. Typical control operations are GETTYP (get device type), POSITION, DUMP BUFFERS, etc.

STATUS operations are used to read device or file specific data. Typical status data is DEVICE TYPE, FILESIZE, EOF flag and COLCNT.

POSITION is used to change the place in the file that the next read or write will start transferring data to or from. POSITION affects an I/O channel, not the file itself, so several I/O channels may be positioned to different points in the same file. A file can be positioned anywhere past the last data byte; this is used to expand a file. Although POSITION operations can be performed independently of read or write operations, it is generally more efficient to perform both in the same step; to allow this, an "implied position" operation can be added to read and write operations.

GETEOF is used to determine if the position of a particular file is at or past the file size (i.e., there is no more data to read).

GETCOLCNT is used to read back the simulated print head position of an ASCII text file (or an actual print head position for a line printer, etc.). This is useful when a tabular display is desired. Like the file position, this value is I/O channel dependent.

DEVICE DRIVER CHARACTERISTICS

This section describes the actual characteristics of the device drivers, and how operations on these drivers differ from an "ideal" device (as described under DEVICE-INDEPENDENT I/O).

These characteristics are observable directly by the assembly language programmer via "Syscalls". Many features of the device drivers may be masked by a high level language such as BASIC; to use these features, an escape to assembly language may be required.

DISK File Driver

Disk files under SDOS implement virtually all aspects of general file handling as described under Device-Independent I/O. This section details exactly the operations implemented by the SDOS Disk File Driver.

An SDOS disk file can physically contain as few as zero data bytes, and as many as the remaining free space after an SDOSDISKINIT. SDOS keeps track of disk file sizes accurate to the byte. Apparent file size may be much larger than the actually allocated disk space; such a file is said to be "sparse".

Disk files may be allocated "dense"ly or "sparse"ly. A dense file is one in which data clusters are allocated for each data byte whose position is less than the file size. A sparse file may have a position (with a smaller value than the file size) for which no data cluster is allocated (data read from this area of the file appears as zeroes).

An OPEN is used to open a disk file (that must already exist) for reading and/or update. If the file does not exist, an error will occur. A CREATE will CREATE a new disk file which will supersede the old version of the file when the new file is closed. The new file will contain zero data bytes after creation. A new file cannot be created if the old file is write protected, or a new file by that name is being created.

Any OPEN or CREATE that specifies a filename that does not contain an explicit device identifier will be automatically assumed to be a disk file on the default disk (DISK:). Also, any filename that is prefixed by a disk device name, and does not consist solely of the device name is assumed to be the name of a disk file on the specified disk.

For the form of disk file names, see the section on DEVICE and DISK FILE NAMES. Disk file names may include a parenthesized integer; this integer is used by CREATE to allocate enough disk space at file creation time to contain the number of data bytes specified by the integer. This has two advantages: first, it decreases the amount of time needed to allocate the space to the file (it is cheaper to allocate all at once than to allocate several little pieces when SDOS discovers it needs them) and it increases the probability the allocation of the file on the disk is contiguous, which decreases random access time to the file. No error is given if there is not enough disk space to satisfy the request. OPEN will parse but ignore the size.

If CREATE is used to make a new disk file, and there is an old file by the same name, the old file must not be delete or write protected or an error will occur and the new file will not be created (nor will the channel be opened). Also, no file by that name may be CREATEd simultaneously (i.e., in psuedo-BASIC,

```
CREATE #1,"X"
CREATE #2,"X"
```

will result in an error). Otherwise, the new file is created, and the channel is opened. As long as the newly created file is still open on the channel on which it was created, that new file is in the state of "being CREATEd". If an old file with the same name does exist, an OPEN SYSCALL executed after the CREATE, looking for the same file, will open the old file. If the system crashes before the new file is closed, the old file will be unaffected in any way. Even after the new file is closed, channels still open to the old file will not notice any difference. When the last channel OPEN to the old file is closed, the space for the old file is returned to free disk space.

Example:

TIME	OPERATION	ACTION
1	OPEN #1, "ABC"	Opens old ABC
2	CREATE #2, "ABC"	CREATEs a replacement
3	OPEN #3, "ABC"	Opens old ABC
4	CLOSE #2, "ABC"	Marks old verson of ABC as deleted
5	OPEN #4, "ABC"	Opens file generated at time 2
6	CLOSE #1,#3	Deletes old ABC

CLOSEing a disk file causes changes to the file size, protection, and other characteristics to be updated on the disk. IF THE SYSTEM CRASHES WHILE THE FILE IS OPEN, THESE CHANGES ARE LOST (NOT RECORDED IN THE DIRECTORY). If the disk file is newly created, and is not replacing another by the same name, closing will make its name appear in the directory. If the file is newly created, and it is a replacement for a file that already exists (i.e., one by the same name), then the new file will replace the old in the directory, and the disk space allocated to the old file will be returned to free space as soon as no other I/O channels remain open to the old version of the file. Disk space allocated to a file beyond the file size will be returned to the free disk space pool when a file is closed.

RENAME is used to change the name of a disk file. RENAMEing a disk file to its own name is legal, and can speed up later OPENs of that file since a rename causes the file name to be re-hashed into the directory. Refer to hash-lookup description of files. A disk file cannot be renamed if it is write protected, or a file by that name already exists, or a new file by that name is being created.

DELETE is used to free the space being used by a disk file and remove the filename from the directory. A file cannot be deleted if it is delete or write protected, or if a new version of the file is being created.

READA performs exactly as specified by SYSCALL:READA. READAing through a large, sparse portion of a file may take an excessive amount of time due to the automatic suppression of all the zero bytes found in the sparse area. WRITEA, WRITEB, and READB match the SYSCALLs exactly. If an error occurs during a read or write, the file position may not be advanced properly.

CONTROL operations available on disk files are the following:

CC:POSITION
: Used to set file position before a read or write operation. See also SYSCALL:WRITEx and SYSCALL:READx.

CC:DUMPBUFFERS
: Forces all data related to the file back to the disk media, so it is recorded permanently in case of a later crash.

CC:SETFILEDATE
: Sets the creation/update date of the file. The date supplied must be in the same format as returned by a SYSCALL:READB to the CLOCK: device. Note that the file date is automatically updated whenever a WRITE or CC:SETFILESIZE operation is applied to a file.

CC:SETFILEPROT
: Sets the file protection byte to the byte supplied. See DIRECTORY.SYS for structure of file protection byte. If the BACKUP protection bit is set, it will be cleared if any RENAME, CC:SETFILESIZE, or WRITE operation occurs. If the DELETE protection bit is set, the operations RENAME, DELETE, WRITE and CC:SETFILESIZE will not be allowed.

CC:SETFILESIZE
: Sets the file size to the current file position. This operation can be used to extend a file (the extension will be sparse until written) or to truncate a file (data written beyond the file position given by the file size will become inaccessible, and data clusters that were allocated beyond that point will be returned to the pool of free clusters when the file is closed).

CC:POSITIONTOEND
: Sets the file position equal to the file size; has the same effect as as a CC:POSITION applied to the result of an SC:GETFILESIZE. Generally used when extending a file is desired.

STATUSes obtainable from a disk file are:

SC:GETPOS
Read position of file.

SC:GETCOL
Get file column number. This value is zeroed by a CC:POSITION or READB/WRITEB and adjusted as data bytes are read or written in ASCII mode. The disk file driver advances the column count by one for any visible character read/written; decrements by one if ASCII:BS is encountered; zeros the column count if ASCII:CR is encountered; advances the column count to the next multiple of 8 if ASCII:HT is found; and leaves the column count alone for all other codes. The value of the column count at a particular point in a file thus depends on the last operation of a file; it is intended only for use with sequential ASCII reads and writes.

SC:GETEOF
Returns EOF hit flag. EOF is set if positioned at or past file size. EOF also set when last byte of file is read or overwritten, or file is extended. EOF is reset when file is positioned with a positioning value less than the file size.

SC:GETTYPE
Returns device type of DVTYP.FILE. See SDOSUSERDEFS.ASM. All devices (drivers) are able to return a device type.

SC:GETFILESIZE
Returns the position of the last data byte written to the file, plus 1. If file has no data written in it, returns zero.

SC:GETPARAMS
Returns data about the file, such as sector size in bytes, and the cluster size.

SC:GETFILEDATE
Returns the creation/update date of the file in the standard system date format (same format as a SYSCALL:READB would return from the CLOCK:). device.

SC:GETFILEPROT
Returns the protection byte currently associated with the file. See DIRECTORY.SYS description for format of protection byte.

No other status is obtainable from a disk file.

SDOS will allocate data clusters to a file automatically whenever a write request to a non-allocated part of a file occurs (it does not allocate from the current end of file up to the point of the write; it simply leaves that part of the file sparse). A cluster allocated in a formerly sparse part of a file is automatically zeroed to preserve the "zero" property of the part not modified.

SDOS attempts to allocate data clusters contiguously (with respect to Logical Cluster Numbers) to minimize scattering of the file over a disk and to minimize sequential processing time. If absolutely contiguous allocation is not possible, SDOS allocates the closest free LCN that starts a contiguous block of BOOT:MIDALLOC free clusters.

The SDOS disk file driver keeps track of OPENed (CREATEd) files via File Control Blocks. FCBs are in one-to-one correspondence with open files (not channels), and contain what amounts to a DIRECTORY.SYS entry. In particular, the FCB holds the amount of disk space allocated to a file and its apparent size. If a file is extended on one channel, the extension will be apparent immediately on a different channel on which that file is also open because of the shared FCB.

Disk sectors are kept in a pool of sectors to minimize disk reads of frequently accessed data. Data written into a file will be immediately available through another I/O channel on which that file is open because the (modified) disk sector in the pool is shared. Modified sectors in the pool are written back to the disk as space is required to bring in another disk sector according to a Least Recently Used discipline. The oldest sector on the queue will be written back if its disk is free.

These side effects of the FCBs and the disk sector buffer pool are subtle but desirable because it is appropriate that different programs be able to share a file and its contents exactly as it is in any instant in time. Many disk operating systems do not provide this exact sharing capability, and consequently make it hard to build a set of programs that interact through a common data base.

SDOS optimizes sequential I/O to disk files via "read-ahead". Whenever data from a particular sector of a disk file is fetched, SDOS pre-reads the next sector of that disk file into the sector pool. The read-ahead happens in parallel with processing of data from the first sector. This scheme decreases sequential file processing time, and lowers the cost of reading records that span sector boundaries to an acceptable level.

DISK Device Driver

The SDOS disk device driver allows access to the entire contents of a disk as though it were a single, large file. This facility is generally only used by utility programs to initialize, check out, and repair the file structure on a disk, but it may also be used to squeeze out the last ounce of available disk space, to cut down access time to a large file, or to read/write disks compatible with the drive but intended for other disk operating systems.

Disk device drivers may also be used to perform operations on the device itself, such as to dismount a disk.

A disk device driver is OPENed when SDOS is asked to OPEN a file whose name consists only of a disk name. (Writes to the device are illegal until a CC:UNLOCKDISK call is made to enable this; this protects the file structure against damage from casual programs since they typically don't issue this call.)

A disk device which has been DISMOUNTed recently will have a Map Algorithm of :ØØØ1. If the disk device is already mounted (i.e., has been used for disk file operations), then the map algorithm will be that given by the BOOT.SYS file on the disk.

The disk device driver treats CREATE calls exactly like an OPEN.

CLOSEing a disk device simply disassociates the I/O channel number, and otherwise does nothing.

RENAME and DELETE operations directed to a disk device will cause an error.

READA and READB act as described under SYSCALLs; the contents of the disk are treated as a single, large stream of bytes. WRITEA and WRITEB act as described (once enabled by CC:UNLOCKDISK), however, a disk device cannot be "extended" when more space is needed, so writing off the "end" of the disk device will cause an End of File error, and the written data will be lost.

Access to sectors may be obtained by positioning a disk device to a byte position which is a multiple of the sector size for that disk.

Disk device drivers support the following CONTROL operations:

CC:POSITION
To position for later reads/writes.

CC:DUMPBUFFERS
This control operation will cause all modified sectors belonging to the disk to be written back to it. It will also cause information changed in FCBs of files open on that disk to be written back. Information in FCBs for newly created but not yet closed files is NOT written back to the disk. This is not a substitute for a DISMOUNT control operation. No parameters are needed.

CC:UNLOCKDISK
This enables WRITEA and WRITEB to work properly on a disk device. If CC:UNLOCKDISK is not issued after OPENing a disk, and prior to a write, a "disk is software write protected" error will occur. Requiring this control operation to write on the disk device prevents accidental writing to a disk device. CLOSEing the disk device re-enables the write protection. No parameters are needed.

CC:DISMOUNTDISK
This operation is used to make SDOS let go of a disk entirely so it may be removed from the drive. An implied DUMPBUFFERS occurs. If there are any (new or old) disk files OPEN on that disk, an error will occur and the dismount operation will not take place (one should repeatedly issue dismounts until no errors are detected; a disk I/O fault on a dismount will probably require SDOSDISKVALIDATE to repair the disk). The disk I/O driver will be called so that it may physically eject the disk or perform other needed cleanup. A successful dismount also turns off the FORMAT mode switch in the disk sector I/O driver. The map algorithm is set to :ØØØ1 if the dismount succeeds.

CC:SETMAPALGORITHM
This allows the 16 bit Map Algorithm for the disk to be changed. An implied CC:DUMPBUFFERS occurs first; if there are any disk files OPEN on that disk, an error will occur. If any error occurs, the map algorithm will not be changed. The map algorithm is passed in the WRBUF of the SYSCALL block.

CC:FORMAT
CC:FORMAT is used to switch into "blind write" mode, intended for disk formatting purposes. See Disk I/O drivers. This operation may not be available on all disk devices.

Any other CONTROL code is simply passed by the SDOS Disk Device Driver to the Disk Sector I/O driver for its use.

STATUS information obtainable from a disk device is the following:

SC:GETPOS
: As described under SYSCALLs

SC:GETCOL
: As described under SYSCALLs

SC:GETEOF
: As described under SYSCALLs

SC:GETPARAMS
: Returns NBPS (number of bytes per sector), NSPT (number of sectors per track), NTPC (number of tracks per cylinder), and NCYL (number of cylinders) each as 2 byte values. See SDOSUSERDEFS.ASM for details on format of result.

SC:GETFILESIZE
: Returns the size of the disk in bytes; equal to NBPS*NSPT*NTPC*NCYL (the product of the sector size in bytes, and the number of sectors on the disk).

SC:GETTYPE
: Returns DVTYP.DISK

SC:GETLASTBADLSN
: Returns the Logical Sector Number of the disk sector which last caused a Seek, Read or Write error. The LSN is returned as 3 bytes; an LSN of :FFFFFF means "no bad LSN". Executing SC:GETLASTBADLSN, CC:DISMOUNT, or CC:SETMAPALGORITHM causes the value to be reset "to no bad LSN". This STATUS is intended primarily for use by SDOSDISKVALIDATE.

SC:GETERRORSTATS
: Returns error statistics collected by the disk driver selected. Such error statistics record counts and disk controller status after each failed attempt by the driver to perform a seek, read or write operation, and the the LSN of the sector involved when the failed attempt last occurred. Since the disk drivers retry failed attempts, nonzero error statistics can occur and yet the system will still function without error; such errors are known as "soft" errors and are only an indication that some difficulty may be present. Executing SC:GETERRORSTATS, CC:DISMOUNT or CC:SETMAPALGORITHM causes the value to be reset "to no bad LSN". This STATUS is intended primarily meant for display by the DISMOUNT command.

VIRTUAL TERMINAL DRIVER
(CONSOLE:, LPT: and Other ASCII-Oriented Serial Devices)

This section describes SDVT11C, known as the "Virtual Terminal Driver". The Virtual Terminal driver is intended to allow an applications program to operate with the majority of display-oriented display units (terminals), without knowing physical terminal characteristics. Inasmuch as printer devices and terminals have a great deal in common, with respect to output, the secondary intent of the VT driver is to give the application the same uniform view of printer devices.

This is accomplished by defining a set of display-oriented operations for an imaginary (virtual) terminal. The application controls the terminal with this set of operations, giving no regard to the type of physical terminal which may be ultimately used. At time of program execution, the operations commanded by the application are mapped into equivalent operations which the physical terminal can perform.

In the event that an applications programmer desires to explicitly reference a feature peculiar to a particular terminal, he may use installation-dependent CONTROL or STATUS calls, or the binary operations READB and WRITEB to bypass the general nature of the VT driver. In so doing, however, that program becomes tied to a particular terminal and is no longer portable to all terminals serviced by the VT driver.

The VT driver provides keyboard entry, line input editing, and text display functions. For CRTs, the VT driver also provides a standard method of dealing with cursor positioning, data entry via fields, and various screen attributes (denoted as "Coloring" in this document) thus making display-oriented applications portable over a wide variety of terminals.

For each virtual terminal device, the VT driver presents an indefinitely long input or output byte stream to the application. The path of input, from typist to application, travels through several territories, before reaching its destination. Keystrokes are first collected in a type-ahead buffer. When a request for data is made (via a READA or READB, for instance), characters are removed from the type-ahead buffer, in the order received, and assembled in the input line buffer. Characters are moved from the type-ahead buffer to the input line buffer, up to and including the character which terminates the buffer filling process. All subsequent data requests are satisfied from this line buffer, until it has been exhausted; then, the type-ahead buffer is again referenced. If the type-ahead buffer is empty, then input is taken from the keyboard, a keystroke at a time. The type of the last data request (READA, READB, etc.) determines how the type-ahead buffer is filled. If the binary mode has been selected (the last request was a READB), then all keystrokes are faithfully stored in the type-ahead buffer. On the other hand, if the ASCII mode has been selected (the last request was a READA or CC:ACTIVATIONCK control call), the parity bit is stripped from all characters; certain control characters are assigned special meaning (see Control Characters in this section) and are not stored in the type-ahead buffer. Editing of the input line is performed at the time of character transfer from the type-ahead buffer to the input line buffer: if a READA or a CC:ACTIVATIONCK control call initiated the transfer, then the input line buffer is filled in ASCII mode and line editing is performed; otherwise, the data is transparently copied through the input line buffer to the RDBUF specified by the request. When ASCII mode keystrokes are being stored in the type-ahead buffer, switch requests, such as ^A, ^C, ^S, and ^P (to name a few), are serviced immediately, and are not retained in the type-ahead buffer.

Associated with each virtual terminal device is a "Device Profile Block". The DPB customizes the terminal to operate with specific manufacturers' devices so that standard SDOS operations are converted to equivalent device-specific operations. This allows application programs to position cursors, "color" the screen or screen regions, or update and erase the screen without knowing the specific device type. A system command, SDOSSET, can be used to change which device profile is in use; some profiles are "malleable"; i.e., changeable, so even devices with properties not handled by standard DPBs in a system can be accommodated. There are also special control calls to allow an application to select or modify particular profiles.

A terminal may be OPENed or CREATEd, using the device name "CONSOLE:", "PORT1:", "PORT2:", etc.; a printer to "LPT:", "LINEPRINTER:", etc. Doing an OPEN or CREATE sets the ASCII activation set to <CR> only, sets the tabs to 8, 16, 24 .... up to 132, performs CC:ECHO and CC:KILLENABLE control calls, and sets the background color to "black" (see CC:BACKGROUND). CREATEs to non-ready devices are aborted with a "Device Not Ready" or "Printer Not Ready", depending on whether the device was a console or printer, respectively; this prevents applications from outputting data to un-ready devices in a way which is convenient to test. A terminal/printer may be open on several channels provided that all channels belong to the same task; output display by the terminal is exactly what would be seen if the I/O requests had been all directed to one channel in the same order.

CLOSE disassociates the I/O channel from the driver. For printers, if part of a line has been printed, the VT driver will complete the line by effectively WRITEAing ASCII:CR; if a partial page has been printed, it will finish the page by effectively WRITEAing ASCII:FF, thus assuring that each use of a printer leaves the paper aligned at top of form for the next use. CLOSE finally does an implied CC:DUMPBUFFERS, and gives an error if the device times out.

RENAME and DELETE operations are illegal.

READA and WRITEA are the normal I/O modes used with the terminal, and match the SYSCALL specification. A READA causes the characters to be taken from an input line buffer maintained by the driver. When the input line is exhausted, and a READA is issued, the driver processes characters from the type-ahead buffer, placing regular keystrokes in the input line buffer, performing editing as directed by control keys, and performing echoing for the typist's benefit. A ^Z read from the type-ahead buffer will cause an End of File condition to occur. Parity is stripped, leaving only 7-bit ASCII codes. Characters are not taken from the input line buffer until activation has been signaled. READA terminates when an activation character is encountered, or RDBUF has no room for the next character. In the latter case, an "Activation Not in Buffer" error is returned, along with as much data as RDBUF can hold. READA must be done in line mode: a non-line mode request for more than zero bytes will result in an "Illegal Device Operation" error; READA non-line mode for zero bytes is accepted for backwards compatibility reasons to allow change of mode from Binary reads to Ascii reads.

As a general rule, SDOS uses a single <CR> character to represent <CR><LF> as a pair. Line feeds are not an acceptable alternative.

When a READB is issued, keystrokes are accumulated in the input line buffer (and the type-ahead buffer, as necessary), with neither echoing nor pre-processing of any kind. The exact key codes generated by the terminal hardware are passed directly to the application, including the parity bit. READB is terminated when the reply buffer is filled. WARNING: an unsatisfied READB to a VT device cannot be aborted; we do not recommend using this.

If the last operation upon the terminal was READA, then most control keys, including ASCII:ESC and ASCII:RUBOUT cause various actions to be taken by the VT driver; these keystrokes are not passed to the application. If READB was last issued, no special interpretation of any keystroke is made; all keystrokes are placed in the type-ahead buffer for processing by the application. READA and READB permit a Ø-byte read request for the purpose of changing input modes. See section on Control Characters for a complete list of the control characters, and their actions, upon both input and output.

WRITEA causes text to be output to the terminal. All characters are first stripped of the "parity" bit (bit 7), and then inspected to determine their interpretation. Printing characters are sent to the device. Tab characters are expanded according to the tab table assigned to each terminal. ASCII:CR characters cause an ASCII:LF and a variable number of idle characters to be output after them. ASCII:FF (form) characters cause CRT screens to be cleared, and cause printers to move to top-of-next-page. Other control characters are generally printed as ^c, where c is the keystroke used with the control key. See Table of Control Characters for a complete list of the control characters, and their actions, upon both input and output.

WRITEB causes the bytes to be sent to the terminal exactly as specified in the write buffer, including the "parity" bit. No linefeeds or idles are inserted. The logical column count is zeroed, and the VT driver assumes it no longer knows the location of the cursor (the application must issue a CC:POSITION or perform an implied positioning call before the VT will know where the cursor is again).

The VT driver supports the following control operations:

CC:POSITION
The positioning information is treated as a cursor position of the form R*256+C, where R is the desired row (Ø is the top row), and C is the desired column (Ø is the leftmost column). Any value which would cause the cursor to position off the display, will result in an Illegal Device Operation error, and the cursor will not be moved. Positioning the cursor of a hardcopy terminal (display depth is zero) or a printer is not permitted, and will result in an Illegal Device Operation error. Note that SYSCALL:READA, SYSCALL:READB, SYSCALL:WRITEA and SYSCALL:WRITEB all allow implied positions in SCBLK:EXTENSION, so that a single call can both position the cursor and do I/O.

CC:DUMPBUFFERS
This is generally a no-op, since the driver dumps characters to the device as fast as it can; it does check for a device timeout. No parameters are needed.

CC:ECHO
This enables echo on READA. No parameters are needed.

CC:NOECHO
This shuts off echo on READA. No parameters are needed.

CC:WRAP
This enables line wrapping when a line exceeds the display width.

CC:NOWRAP
This disables line wrapping when a line exceeds the display width: the line is truncated, and the cursor is left on the same line, following the last character displayed.

CC:IDLES
This sets the number of idles to be transmitted after a <CR> or <LF>. The first byte in WRBUF is the idle count (Ø is legal), the second byte in WRBUF is the character after which the idles are to follow. If the second byte is not present, idle trigger defaults to <LF>. A character other than <CR> or <LF> will cause an "Illegal Device Operation" error. This information is not changed by OPENs, CREATEs, or CLOSEs. Note that the current profile must be either malleable or hardcopy (an option which must be SYSGENed into the I/O package); otherwise, a Profile Not Malleable error will be returned.

Caveat: Some terminals will behave differently for <LF><CR> than for <CR><LF>.

CC:TABS

This sets tab stops for tab simulation. The WRBUF must hold a string of bytes, each byte specifying the next tab stop. Each successive byte must contain a column number larger than the previous one. When the terminal is first opened, tab columns are set at every eighth column, up to 132 columns (Ø is the first column). Up to 16 tab stops may be set; if too many are supplied, an "Illegal Device Operation" error will result. If the order of the tab stops is incorrect (not monotonically increasing), an "Illegal Device Operation" error will be returned, and the old tab settings will be undisturbed. Since CONSOLE: devices tend to stay open for long periods of time, CONSOLE: tab settings have a tendency to remain in effect long after needed.

CC:SETACTBLOCK

This specifies a non-standard set of activation characters. The non-standard set is specified with a vector of 128 bits (arranged in WRBUF as 16 bytes), corresponding to the ASCII character set. The least significant bit in the first byte corresponds to character code ØØ, and the most significant bit of the 16th byte corresponds to character code :7F. When a bit is set, the corresponding character is interpreted as a non-standard activation character; when the bit is reset, the standard interpretation applies (see the chart of Control Characters, below). The activation set is restored to the standard interpretation (all bits reset) by OPEN and CREATE. When marked as activation characters, control characters and ASCII:RUBOUT are never echoed, while printing characters echo only if echo is enabled. Note that <CR> is always an activation character -- marking it as a non-standard activation character only changes its echoing characteristics (as a standard activation character, it echoes if echo is enabled; as not-standard, it does not echo).

CC:CLRINPUT

This clears the input line and type-ahead buffers. This is useful when input, following an abnormal condition, is required.

CC:CLROUTPUT

This clears the output buffer. It is generally useful only when the output buffer for a device is very big, or the device is very slow; otherwise, the buffer will empty quickly anyway.

CC:SETREADTIMEOUT

This sets a timeout on a subsequent READA or CC:ACTIVATIONCK control call. The timed period begins when the subsequent input operation is issued. When the timed period has expired, the input operation is terminated with a "Timed Input Expired" error, and the data input thus far is returned in the RDBUF supplied by the input operation. The length of the period is expressed in 60ths of a second, as a 16-bit value. Note that the period allowed is only approximately what is specified, but is guaranteed to be longer than the value given. The value is found in WRBUF, and WRLEN must be 2.

CC:SETPROFILE

This selects a new device profile, which includes a function mapping VT operations to physical terminal operations. Selection of a profile sets default device width, depth and output timeouts; it specifies how the device will position the cursor, clear the screen, erase to end of line, and go to new lines; it controls how "coloring" is to be displayed, etc. Such a profile generally represents a particular model of CRT/printer. The new profile replaces the old profile, and is retained until changed or the system is re-booted. Some profiles are malleable and may be somewhat altered to accomodate devices for which there is no specific profile (see below). As the malleable profile is a template, any alterations are retained with the device, rather than with the profile. Selection of a new profile will cause previous alterations to be lost. WRBUF contains one number, which is the profile "name". Specification of a profile not sysgenned into the I/O package will result in a "No such Profile" error. WRLEN must be 1. This call is normally only used by the SDOSSET program. For a list of profile names, see the documentation for SDOSSET or the file IOVTDPBS.ASM.

Note that adding a new profile requires changes to the I/O package.

CC:ALTERPROFILE

This alters the currently selected profile (see above), if it is malleable; if it is not, a "Profile Not Malleable" error is returned. The alterations are confined to defining a cursor-positioning sequence, an erase to end of line (EEOL) sequence, and a home and clear screen (CLEAR) sequence. An "Illegal Device Operation" will be given if the parameter supplied are unreasonable. Note that the cursor positioning sequence contains, in place of the row and column numbers, the offsets to be added to the row and column numbers supplied by the application; thus, the cursor positioning sequence could be used, by itself, to position to location (Ø,Ø). More extensive alteration must be accomplished by defining a new profile and incorporating it into a newly-generated system.

WRBUF must contain the following data:

ALTERPROFILE:CPLEN	significant length of cursor positioning sequence following; 1 byte in range 3 to 4. If this length is < 3, then the VT driver will output '@@' instead of a cursor position.
ALTERPROFILE:CPSEQ	cursor position sequence, which includes the row and column offsets; 4 bytes
ALTERPROFILE:CPIDLES	number of idles to follow cursor positioning sequence; 1 byte
ALTERPROFILE:ROWDISP	displacement into cursor positioning sequence of row number; 1 byte
ALTERPROFILE:COLDISP	displacement into cursor positioning sequence of column number; 1 byte
ALTERPROFILE:CLLEN	significant length of CLEAR sequence following; 1 byte in range Ø to 4. If Ø, a CLEAR sequence will be simulated by generating enough ASCII:LFs to move to the top of a page if a printer device (page depth can be changed by CC:SETPARAMS). This is a useful device if a system has different size paper forms, and no forms control tape.
ALTERPROFILE:CLSEQ	CLEAR sequence; 4 bytes

ALTERPROFILE:CLIDLES	number of idles to follow CLEAR sequence; 1 byte
ALTERPROFILE:EEOLLEN	significant length of EEOL sequence following; 1 byte in range Ø to 4. If Ø, the sequence will be simulated.
ALTERPROFILE:EEOLSEQ	EEOL sequence; 4 bytes
ALTERPROFILE:EEOLIDLES	number of idles to follow EEOL sequence; 1 byte.

CC:WRITEEDITLINE

This appends the contents of WRBUF to the end of the input line buffer as if the typist had entered that data. A subsequent READA or CC:ACTIVATIONCK control call will cause the data to be displayed, in the usual fashion, and the typist may edit the data until an activation character is entered. Note that using the CC:NOECHO control call, prior to invoking CC:WRITEEDITLINE, will inhibit that data from being displayed at the time of the READA or CC:ACTIVATIONCK control call. An activation character may be present in WRBUF, but will prevent the typist from editing characters prior to the activation character: activation will occur immediately. If WRLEN is greater than the space available in the input line buffer, or data follows an activation character in WRBUF, an Illegal Device Operation error is returned and no data is transferred to the input line buffer.

If present, the syscall block extension contains a cursor position at which the cursor should be left, after the data has been echoed by the input operation; otherwise, the cursor will be left at the end of the data supplied in WRBUF.

When a field has been explicitly defined (see the CC:SETFIELDSIZE control call, below), the 1-byte syscall extension is the column number at which the cursor is to be placed, at the time of the READA or CC:ACTIVATIONCK control call; if that column is in the middle of a tab expansion, the cursor will be positioned following the expanded tab.

CC:SETFIELDSIZE

This defines an input field for a subsequent READA or CC:ACTIVATIONCK control call. WRBUF contains the field width. The field width must be at least 1 and no greater than the width of the display. If the field width is Ø, or exceeds the limits of the display, a "Bad Field Width" error will be returned, and the field definition will not be made.

Unless any of the cursor control keys for moving left, right, up, and down have been designated activation characters, they may be used to position within the defined field. When an attempt is made to position the cursor beyond the boundary of the field, that character is treated as an activation character and the operation is terminated; the terminating cursor control character is appended as the activation character, and the cursor is not moved. An SC:GETACTCOL status call may be issued to determine the exact column of exit. If any of the cursor control characters is designated an activation character, then that character cannot cause a field exit condition, and will activate immediately upon use.

The field definition terminates upon field exit, or entry of an activation character. If the field, at the time of the input operation, is not contained completely within the display width, that input operation will terminate with an "Bad Field Width" error and the field input mode will be cancelled. ^C^C will cancel any outstanding field definition.

CC:SETPARAMS

Sets the width (1 byte) and the depth (1 byte) of the display; this overrides the default from the device profile chosen. Zero depth means that the terminal is not a paging device and will print ^L when given a form feed character.

CC:ACTIVATIONCK

This is used to enable keyboard input without causing the program to suspend operation. CC:ACTIVATIONCK returns an "Activation Received" error if an activation character is in either the input line buffer or the type-ahead buffer. If no activation character is present in either buffer, the input line buffer is filled from the type-ahead buffer, unless this has already been done by a previous call of CC:ACTIVATIONCK. A READA issued following an "Activation Received" error will always return immediately with the data requested and/or an error appropriate to a READA (If ^C^C has been seen while the SDOS/MT and KILLPROOF flags are set, a "Program Killed" error will be returned; otherwise, ^C^C will result in the program being killed.) *Once the CC:ACTIVATIONCK control call has been issued, subsequent I/O requests (with the exception of CC:ACTIVATIONCK, status requests, and SYSCALL:READA) will result in an "I/O In Progress" error. This state is exited by issuing a READA upon receipt of an "Activation Received" error. A CC:SETREADTIMEOUT control call issued prior to the initial CC:ACTIVATIONCK can be used to limit the time spent in this state. When the timed period expires, the next CC:ACTIVATIONCK will return an "Activation Received" error, and the subsequent READA will return the expected "Timed Input Expired" error, along with any data received prior to the expiration.

* See the section on SDOS/MT support for a caveat that applies to this note.

CC:SETBAUDRATE

This call is used to change the baud rate of a device. WRBUF contains a 16 bit unsigned integer representing the exact baud rate desired (rounded to an integer). An "Illegal Device Operation" is returned if the baud rate cannot be changed, or cannot be changed to the specified value.

CC:COLORING

For the purposes of this control call, a "color" is that which changes the appearance of text without changing its meaning or size. This call supports the myriad available features dealing with display appearance: these include, but are not limited to: color, intensity, underscoring, and blinking. It explicitly does NOT handle characters whose size is non-standard (i.e., double-width or double-height) for the device. 16 bits of data, found in WRBUF, specify the desired display mode for subsequent output: all display characteristics must be specified by the same control call at one time.

The mode change is made immediately, and the mode is saved for later use by the position control call. All characters output via WRITEA are "colored" according to the last color selected by this call. When a position control call is made, the "zero" coloring is selected (see CC:BACKGROUND, below), the positioning is performed, and the coloring selected by CC:COLORING is re-instated.

CC:COLORING does not cause the cursor to move (some terminals violate this, due to their design deficiencies).

Two bytes in WRBUF are used to specify the display modes. The first byte is divided as follows: 2 bits for intensity, 1 bit for blink, 1 bit for underscore, 1 bit for reverse video, 3 bits for (inverted) color (1 bit each for "not red", "not green", and "not blue"). The second byte contains 3 bits for selecting alternate Roman character sets; the remaining bits are undefined and must be zero. The default color of "zero" (both bytes zero) selects the standard Roman character set, standard intensity, no reverse video, no underscore, no blink, and the color white (i.e., the display mode obtained for virtually all "dumb" CRTs). The "zero" color is automatically selected by OPEN.

Although this control call is recognized by all systems, its actual implementation will vary according to the particular terminals being supported; in the simplest of cases, it will be implemented as a NOP.

CC:BACKGROUND

A "background" color is the color displayed in all screen locations which do not contain a character.

CC:BACKGROUND selects the default coloring to be used when the display is cleared, or when cursor positioning is to be done (see CC:COLORING, above). The required byte of data is found in WRBUF and is of the same format as for the first byte of CC:COLORING, above. A black background (hex :Ø7) is automatically selected by OPEN.

Although this control call is recognized by all systems, its actual implementation will vary according to the particular terminals being supported; in the simplest of cases, it will be implemented as a NOP.

CC:KILLPROOF

This is used to KILLPROOF a specific VT input device. What that means is that ^C^C and ^D will be rejected with a beep when they are entered. ^C while killproof clears the type-ahead buffer.

CC:KILLENABLE

This is used to cancel the effect of a CC:KILLPROOF control call directed at the same VT input device. Note that SYSCALL:KILLPROOF is not overridden by this control call.

CC:SETEXCEPTION

This call is used to specify exceptions to VT driver processing. At this time, the only exception defined is for SEDIT; and specifies that fields also activate on ASCII:RUBOUT at left end of field, and on ^U or ^L at right end of field.

CC:SETOUTPUTTIMEOUT

This call is used to specify a new value for output timeout interval, and overrides the default selected by the Device Profile Block last chosen. It is especially useful with the VT:MALLPT profile when the printer has a large buffer of its own, and goes "BUSY" for long periods while it prints. The interval is specified as a two byte number in WRBUF in 6Øths of a second.

STATUS OBTAINABLE FROM THE VT DRIVER

Many of the statuses available from the VT driver are simply images of data specified by Control calls to the driver. This is for convenience of the SDOSSET program, and allows it to show the operator the "current" settings of things before modification.

SC:GETPOS
Reads the cursor position in the same form as CC:POSITION.

SC:GETCOL
If the input line buffer is empty, this returns the output column number; otherwise, this returns the column number corresponding to the first byte to satisfy the next read. The column number is the same as used in CC:POSITION. A READB zeroes the column number. Returning the column number corresponding to the next input character when there is a partially-read input line makes it possible to distinguish between "TERSE" command lines and "VERBOSE" command lines; if the column count is zero when a program gets control, there must be nothing in the line buffer and so VERBOSE mode is desired; otherwise, something is in the line buffer and so TERSE mode is desired (see COMMAND INTERPRETER).

SC:GETEOF
This returns a non-zero byte if ^Z was seen while in READA mode, and the input line buffer is empty; otherwise, this returns a zero byte. End of File status is never set while in READB mode to a VT device. Note that the only way to reset this status is to CLOSE and reOPEN the channel.

SC:GETTYPE
Returns DVTYP.CONSOLE or DVTYP.PRINTER, as appropriate.

SC:GETPARAMS
Returns the width (1 byte) and the depth (1 byte) of the display. Zero depth means that the terminal is a hardcopy device with continuous paper. Printers return paper width and depth.

SC:GETPROFILE
Returns the current profile "name" (a one byte number); suitable for use by the CC:SETPROFILE control call.

SC:GETPROFILENAME
Returns a one to 16 character ASCII text string corresponding to the numeric profile "name" (1 byte) specified in WRBUF. This call does NOT change the profile currently selected on the device. Return a "No Such Profile" error if the profile name specified in WRBUF is not sysgenned into the I/O package. This call is used to all SDOSSET produce a human-readable list of DPBs configured into a system.

SC:GETPROFILEALTERATION
Returns the current profile alterations in exactly the format given to CC:ALTERPROFILE. Gives a "Profile Not Malleable" error if the profile currently selected is not malleable (and therefore has no alterations).

SC:GETFREECOUNT
Returns a 16 bit integer specifying how much room is currently available in the output buffer for a device (memory-mapped video displays always return "1").

SC:GETDATACOUNT
Returns a 16 bit integer specifying how much data is currently available in the input ring buffer for this device. Can be used to prevent hanging the system when doing READB.

SC:GETOUTPUTTIMEOUT
Returns the current value of the Output Timeout for this device, in a form suitable for use with CC:SETOUTPUTTIMEOUT.

SC:GETBAUDRATE
Returns the current baud rate for this device, in exactly the form required for CC:SETBAUDRATE. Devices which cannot change baud rates usually return "Ø".

SC:GETTABS
Returns the current tab settings for this device, in exactly the form required for CC:TABS.

SC:GETIDLES
Returns the count of idles to follow a Newline sequence, and the Idle trigger character, in exactly the form required for CC:IDLES.

SC:GETWRAP
Returns a non-zero byte if Wrapping (see SC:WRAP) is enabled, else return a zero byte (wrapping is disabled).

SC:GETCOLORING
Returns 2 bytes of Coloring information in exactly the form required by CC:COLORING.

SC:GETBACKGROUND
Returns 1 byte of Background Coloring information in exactly the form required by CC:COLORING.

SC:GETACTCOL

Returns both the column position and the line buffer displacement at which the last activation character was entered (the activation character, itself, is placed at the end of the input line and is obtained via READA or READB). Note that if echoing is disabled, the returned column position value will be meaningless.

SC:ATTENTIONCK

This checks for "Operator Requested Attention" status. If found, the status is cleared and an "Operator Requested Attention" error is returned.

SC:STATUSCK

This returns a "Status Has Changed" error if the VT device has had an interesting change of status, which include receipt of an activation character, receipt of ^C^C, a "Timed Input Expired" error, a "Device Timed Out" error, etc.

CONTROL CHARACTERS

This table describes how control characters are treated if they are NOT marked as activation characters (see CC:SETACTIVATION).

:ØØ	NUL	input:	ignored
		output:	discarded
:Ø1	^A	input:	toggles the CAPS LOCK switch, echoes immediately at the end of the line
		output:	prints ^A
:Ø2	^B	input:	requests BASIC breakpoint, does not echo
		output:	prints ^B
:Ø3	^C	input:	clears the input and output buffers, resets the FREEZE OUTPUT and DISCARD OUTPUT switches (see ^S, ^O), resets the PAGE MODE switch (see ^P), echoes immediately at the end of the line, aborts the program if two ^C's are received in succession; ^C^C will be rejected with a beep if KILLPROOF is set.
		output:	prints ^C
:Ø4	^D	input:	invokes the debugger immediately, does not echo; a beep is echoed if no debugger is available (CNFG:VTDEBUG=Ø). Illegal under SDOS/MT.
		output:	prints ^D
:Ø5	^E	input:	causes all input at, and to the right of, the cursor to be erased from the display and deleted from the input buffer.
		output:	erases the remainder of the display line (erase to EOL)
:Ø6	^F	input:	positions cursor at left side (front) of current input field. Illegal for hardcopy terminals.
		output:	prints ^F

:Ø7	^G	input:	causes BASIC to resume execution from the current breakpoint, does not echo
		output:	beeps
:Ø8	^H	input:	implements the backspace function, does not echo
		output:	implements the backspace function. Backspace across edge of screen is not allowed.
:Ø9	^I	input:	positions the cursor at the next tab column, when read
		output:	positions the cursor at the next tab column
:ØA	^J	input:	rejected with beep; see CC:SETFIELDSIZE
		output:	discarded
:ØB	^K	input:	rejected with beep; see CC:SETFIELDSIZE
		output:	prints ^K
:ØC	^L	input:	implements the forespace function, does not echo
		output:	causes a PAGE BREAK if the PAGE MODE switch is set (see ^P), homes the cursor, selects the background color, and clears the display if depth is not zero, prints ^L if the the depth is zero; for a printer device, moves paper to the top of form, such that the next character will be printed in the first position of the line.
:ØD	^M	input:	echoes <CR><LF>, causes program activation
		output:	prints <CR><LF>
:ØE	^N	input:	passed to the application, echoed when read
		output:	prints ^N
:ØF	^O	input:	toggles the DISCARD OUTPUT switch (see ^Q, ^C), echoes immediately; not functional while a READA or CC:ACTIVATIONCK control call is being satisfied
		output:	prints ^O

:10	^P	input:	toggles the PAGE MODE switch (see ^Q, ^C), echoes immediately at the end of the line
		output:	prints ^P
:11	^Q	input:	resumes output suspended due to a PAGE BREAK (see ^P), resets the DISCARD OUTPUT switch (see ^Q), resets the FREEZE OUTPUT switch (see ^S), does not echo
		output:	prints ^Q
:12	^R	input:	if CRT, positions cursor to Right end of current input field; for hardcopy devices echoes ^R<CR><LF> followed by all data entered since the last activation
		output:	prints ^R
:13	^S	input:	sets the FREEZE OUTPUT switch (see ^Q, ^C), echoes immediately
		output:	prints ^S
:14	^T	input:	toggles the BASIC line trace switch, does not echo
		output:	prints ^T
:15	^U	input:	deletes the character at the current cursor location, deletes the corresponding character from the input buffer, does not echo
		output:	prints ^U
:16	^V	input:	toggles the BASIC single step switch, does not echo
		output:	prints ^V
:17	^W	input:	for CRTs, causes the last input line to be retrieved as though the typist had entered it explicitly, if no other keys have been typed since the last input. Illegal for hardcopy devices.
		output:	prints ^W

:18	^X	input:	clears the input buffer; for hardcopy, echoes ^X<CR><LF> and positions to the column at which input began; for a terminal, erases, from the display, the data entered since the last activation character, and positions the cursor at the location where input began; for a terminal with fields defined, erases the displayed field contents, and positions the cursor at the first location of the field
		output:	prints ^X
:19	^Y	input:	passed to the application, echoed when read
		output:	prints ^Y
:1A	^Z	input:	causes END OF FILE status to be set, causes program activation with an END OF FILE error, echoes immediately at the end of the line
		output:	prints ^Z
:1B	ESC	input:	causes cursor to be placed at right end of current input field, OPERATOR REQUESTED ATTENTION status to be set, and returns "Operator Requested Attention" error.
		output:	prints ^[
:1C	^\	input:	passed to the application, echoed when read
		output:	prints ^\
:1D	^]	input:	passed to the application, echoed when read
		output:	prints ^]
:1E	^^	input:	passed to the application, echoed when read
		output:	prints ^^

:1F	^_	input:	passed to the application, echoed when read
		output:	prints ^_
:7F	RUB	input:	deletes the character preceeding the current cursor location, deletes the corresponding character from the input buffer, does not echo
		output:	discarded

NOTE: No control character is passed to the application unless explicitly noted.

SOFTWARE SWITCHES AFFECTED BY CONTROL CHARACTERS

CAPS LOCK
When set, READA will interpret the lowercase letters a-z as uppercase letters. ^A toggles the switch. If a terminal is stuck in upper case, and the alpha lock key isn't the problem, someone probably typed ^A by accident.

FREEZE OUTPUT
When set, further output will be suspended until the switch is reset. On CRTs, ^S will be displayed to remind the typist that the switch has been set. ^S sets the switch, ^Q and ^C reset the switch.

DISCARD OUTPUT
When set, all output will be discarded until either the switch is reset or a READA/READB is issued. ^O will be displayed to remind the typist that the switch has been set. A READA will reset the switch and overwrite the "^O" with "? ". A READB will simply reset the switch. ^O toggles the switch, ^Q and ^C reset the switch.

PAGE MODE & PAGE BREAK
When set, subsequent WRITEA lines will be counted, and when <display depth> lines have been output, a Clear screen request is output, or cursor positioning is attempted, then a PAGE BREAK will occur, and no more output will occur until the typist has acknowledged the page break. This gives the typist a chance to read what is displayed before more output occurs. On a CRT, a page break will be signalled by ^P being displayed in the lower right-hand corner of the screen; on hardcopy devices, output will simply cease. The acknowledgement can be ^P (which prevents further page breaks), ^Q (which allows output until the next page break), or ^C (which prevents further page breaks). On CRTs, a Clear screen requests causes a page break BEFORE the screen is cleared, so the text may be read before it disappears; on hardcopy terminals, the page break occurs AFTER the FORM character moves the paper to top-of-page, so individual sheets of paper may be conveniently printed. All lines output while in page mode will be truncated to fit within the current display width, thus ensuring that line wrapping does not occur so that all lines between page breaks will be captured on the display.

NOTE: All reminders are displayed in the lower, right corner of the display. Reminders will overwrite any characters already in those locations.

SDOS/MT SUPPORT

The following control and status functions are included for complete documentation only. They are subject to change without notice.

Caveat Emptor!!

MULTIUSER CONTROL FUNCTIONS

CC:SETTIMESHARE
: Sets the SDOS/MT flag, which results in different handling of the line flags and ^C^C abort. If the flag has already been set, an "SDOS/MT Already Running" error is returned. RDBUF is filled with system-dependent linkage information for use by SDOS/MT.

CC:STOPTIMESHARE
: Turns off the SDOS/MT flag. SHOULD NOT BE EXECUTED BY USER PROGRAMS, OR A SYSTEM CRASH WILL RESULT.

CC:WRITEANOWAIT
: This defines, for the VT driver, WRBUF as the source of data for an asynchronous WRITEA of WRLEN bytes. RDBUF contains 3 bytes, the first of which the VT driver will set to zero when the request is accepted, and set to non-zero when WRLEN bytes have been written; the remaining two bytes will contain either an error code, or zero if the operation had no errors. Note that WRBUF must not be modified until the request is complete (the first byte of RDBUF becomes non-zero).

CC:WRITEBNOWAIT
: This defines, for the VT driver, WRBUF as the source of data for an asynchronous WRITEB of WRLEN bytes. RDBUF contains 3 bytes, the first of which the VT driver will set to zero when the request is accepted, and set to non-zero when WRLEN bytes have been written; the remaining two bytes will contain either an error code, or zero if the operation had no errors. Note that WRBUF must not be modified until the request is complete (the first byte of RDBUF becomes non-zero).

MULTIUSER STATUS FUNCTIONS

SC:GETLINEFLAGSHINT

Returns zero if no line flags have been collected since the last call to SC:GETLINEFLAGS, otherwise returns non-zero value. The value returned is only intended as a hint; the program must call SC:GETLINEFLAGS to get the true line flags and acknowledge their receipt. Don't ask why.

SC:GETLINEFLAGS

Exchanges a zero with the line flags, and returns that byte. If ^C^C has been seen while the SDOS/MT and KILLPROOF flags are set, a "Program Killed" error will be returned; otherwise, ^C^C will result in the program being killed.

SC:GETTIMESHARE

This checks to see if SDOS/MT is running. If it is, an "SDOS/MT Already Running" error will be returned; otherwise, a normal return will be made.

SC:ALLSTATUS

This checks to see if an SC:STATUSCK status call issued to any VT device would return a "Status Has Changed" error as a response; if so, a "Status Has Changed" error is returned. Note that this status call supplies only a hint.

The CLOCK: Device Driver

The CLOCK: device is used to set and read the current time and date. Since its function is limited, so is its conformance to the SDOS file concept.

The CLOCK: device can only be OPENed. CREATE, RENAME, DELETE, WRITEA, and CONTROL operations are illegal. CLOSE does nothing except to disassociate the I/O channel from the driver.

A READA directed to the CLOCK: device returns a string of 17 bytes in the following form:

```
HH:MM:SS MO/DD/YY
```

where HH is hours on a 24 hour clock, MM is minutes, SS is seconds, MO is the month, DD is the day number, and YY is the year modulo 1ØØ. An ASCII:CR is appended if the READA has line mode enabled and buffer space permits.

A READB returns 6 bytes exactly in the following form:

```
T T T M D Y
```

where T T T is a 24 bit binary value equal to the number of 1/6Ø second clock ticks since midnight; D is the day, M is the month, and Y is the year modulo 1ØØ, all in BCD.

A WRITEB must write exactly 6 bytes in the format read by READB, and is used to set the time of day.

The only status syscall accepted is SC:GETTYP, which returns DVTYP.CLOCK.

SYSCALLS - CONCEPTS

Programs running under SDOS communicate with it via system calls (SYSCALLs). A SYSCALL is a subroutine call (from the user program to SDOS) with a parameter block describing the function to be performed.

This section describes the general philosophy behind the SYSCALLs and their general format. It assumes some knowledge of assembly language.

The most general form of a SYSCALL contains a function code, some fixed parameters needed by the function, a (pointer to) Write buffer and a (pointer to) Reply buffer. Essentially, the SYSCALL causes the specified function to be performed according to the parameters, using data from the write buffer, and storing a result in the reply buffer. Many readers will recognize this as an implementation of

```
RDBUF:= F(PARAMS, WRBUF)
```

The purpose of constraining all SYSCALLs to this form is to simplify the process of transmitting a request from one computer to another, to facilitate networking of multiple computers.

Conceptually, SYSCALL execution proceeds as follows:

1) The user program issues a SYSCALL.
2) SDOS transmits the function code, the parameters, and the contents of the WRITE Buffer from the user's computer to some target computer.
3) The target computer processes the SYSCALL and produces a reply.
4) The reply, along with any error information, is sent back to the SDOS which sent out the request.
5) SDOS places the reply in the user program's reply buffer.

In a stand-alone system, the target computer and the user's computer are one and the same.

The primary advantage of this scheme is that by forcing all SYSCALLS to have a fixed form for transmitting, performing, and receiving replys to function requests, the software logic processing the request can forward it to another computer without having a lot of function-specific knowledge. In particular, it means that the forwarding logic need not be changed even when new functions are added to the list of legal SYSCALLs.

Typical SYSCALL functions are: OPEN file, READ byte stream, LOAD a program, etc. Not all functions require write data (i.e., a STATUS Syscall needs only the function, some parameter bytes and a reply buffer); nor do all functions return a result (WASCII writes a string of ASCII bytes to a file and returns no result). Some functions have neither write nor reply buffers (i.e., EXIT to system). Furthermore, many functions have side effects (like CLOSE I/O channel).

SYSCALL Format:

The following definitions give the formats of a SYSCALL block (SCBLK).

```
*
*       SYSCALL BLOCK DISPLACEMENTS
*
ORG      Ø
SCBLK:OPCODE    RMB     1        Primary SYSCAL Function (Open,
                                 Read, Etc.)
SCBLK:WLEN      RMB     1        Wait Flag Bit (Ø=Wait) and SYSCALL
                                 Block Length (Ø..127)
SCBLK:PARAMS    RMB     2        Parameter Bytes to Opcode (Secondary
                                 Opcode, Channel #)
SCBLK:WRBUF     RMB     2        Pointer to Write Data Buffer
SCBLK:WRLEN     RMB     2        Number of Bytes in Write Data Buffer
SCBLK:RPLEN     RMB     2        Length of Reply (Result of SYSCALL)
SCBLK:RDBUF     RMB     2        Pointer to Read Data Buffer (Where
                                 Result Goes
SCBLK:RDLEN     RMB     2        Ceiling on Size of Reply (Read Data
                                 Buffer)
SCBLK:DATA      RMB     Ø        Other Parameters for SYSCALL; up
                                 to 127-14=113 Bytes
SCBLK:END       RMB     Ø        End of SYSCALL Block;
                                 Assert SCBLK:WLEN[1..7]=
                                 SCBLK:END-SCBLK:OPCODE
```

SCBLK:OPCODE is the desired function, and occupies a single byte. Legal functions under SDOS 1.Ø are shown in table 1. (Definitions of all values for SYSCALL opcodes and related information is given in the SDOSIOPKDEFS.ASM listing in the back of this manual).

SCBLK:WLEN is a single byte with two parts: a Wait flag (the most significant bit) and a LENgth (2 to 127, measured in bytes) (the SYSCALL block length). The wait flag is intended to allow overlapped READ and WRITE operations to files, but is not implemented in SDOS 1.Ø. When this bit = Ø, it means "wait for operation complete before returning control to user program". "1" means "don't wait". To retain compatibility with future releases of SDOS, the user is advised to leave this bit reset (Ø). The LEN field specifies precisely how long the SYSCALL block is. Each opcode requires that this byte have some minimum value, or the SYSCALL will be aborted. The LEN field is used to determine how much data must be sent to another computer. The LEN field can specify more bytes than actually needed by the SYSCALL without ill effect, but processing the unused bytes may increase the execution time of the SYSCALL. All SYSCALLs have at least the SCBLK:OPCODE and SCBLK:WLEN bytes.

SCBLK:PARAMS are 2 bytes used for sundry purposes as parameters to the opcode requested. Three cases are of particular note: first, one of the two parameter bytes is generally used to hold an I/O channel number on I/O-oriented SYSCALLS. Second, a parameter byte may contain an opcode extension byte, as with the STATUS and CONTROL SYSCALLs; the parameter byte selects which control function is to be performed or the particular piece of status information to read back. The third case is some 16 bit number, such as passing an error code to SDOS via the SETERROR SYSCALL. In no case may these two bytes contain a pointer or any other kind of reference to other data in the memory of the user's computer; only data values or relative references to data in the write buffer or the SYSCALL block itself are legal (because after the SYSCALL has been sent to another computer, how could we follow a pointer?) SCBLK:PARAMs need not be included in the LEN count for SYSCALLS such as SYSCALL:CLOSE, SYSCALL:EXIT, etc.

SCBLK:WRBUF and SCBLK:WRLEN define the starting address of the write data buffer, and its length in bytes. SCBLK:WRBUF contains the address of the first byte of the buffer; SCBLK:WRLEN contains the number of bytes in the buffer (Ø to 65535). Note that SCBLK:WRLEN is the actual number of bytes to be processed by the SYSCALL, not the allocated size of the buffer. These parameters are used in SYSCALLs involving filenames to specify the (device and) filename desired, or as data buffer definitions for SYSCALL:WRITEB (Write Binary), etc.

SCBLK:RDBUF and SCBLK:RDLEN select a buffer address and size in which a SYSCALL result/reply is returned. The SCBLK:RDLEN must contain the expected maximum size of the result (in bytes). SCBLK:RPLEN is set to the actual length of the reply given, that is, the actual number of reply bytes placed in the RDBUF. Many SYSCALLs do not return a result. If the SYSCALL block includes space for SCBLK:RPLEN, it will be zerod if no reply is given. If RDBUF overlaps any part of the SYSCALL block or the WRBUF, the SYSCALL operation is not well defined. When an error is returned by a syscall, RPLEN and RDBUF contents are undefined (unless explicitly specified by description of the particular syscall). In particular, there is no guarantee that the RDBUF contents are preserved (even in the presence of an error).

Bytes in the SYSCALL block beyond SCBLK:RDLEN are interpreted in a manner specific to the particular SYSCALL opcode (like the SCBLK:PARAMs bytes). Most SYSCALLs do not need or use these bytes.

An error occurring during execution of a SYSCALL is handled in the manner described under SDOS Error handling. The calling sequence for SYSCALLS is thus:

```
        .
        .
        .
        LDX     #SYSCALLBLOCKADDRESS
        JSR     SYSCALL$        (Equated TO $FB)
        BCS     OOPS            (Go Process Error Code In X)
        .
        .
        .
```

ERROR HANDLING

Error handling is an important part of any programming system. It allows application programs to continue or effect recovery in spite of problems encountered. The error handling strategy outlined here is used throughout most SD software. Facilities to handle errors in a similar fashion are provided by the SD BASIC Compiler, so application programs can also support the same scheme.

Errors detected by SDOS are passed back to the user program for inspection or handling. Each error which can occur is assigned a 16 bit error code (0 to 65535). Blocks of codes are assigned to each possible detector of an error (i.e., errors which SDOS detects have codes from 1000 to 1999, compiled BASIC programs detect errors 2 to 99, EDIT errors are 200 to 299, etc.).

Each (assembly or SYSCALL) subroutine has two exits: a success exit (meaning no unexpected/unrecoverable errors occurred) and an error exit (meaning some error which the subroutine cannot handle occurred).

If the success exit is taken, normal processing can continue. If the error exit is taken, an error code is passed back to the caller for his inspection. The caller has three options:

1) Process and recover from the error (example: for "No Such File" error on an OPEN, a standard default file name might be OPENed).
2) Give up; notify the operator of the error and exit.
3) Decide to pass the error back to his caller with an error indication. This option is particularly important when the caller can fail in many ways not understood by the caller (such as I/O faults).

Processing the error requires explicit checking for each of the possible error codes of interest (due to the large number of unexpected errors, an "if it's not this, it must be that" scheme is not safe; one should ALWAYS check explicitly). Sometimes, data associated with the error is needed for the processing routine to continue; in these cases, the original detector of the error must have saved that data in a place agreed upon by the detector and the routine attempting recovery. An example is a "recovery" routine that prints out the Logical Sector Number of a disk sector on which a read error occurred -- the recovery routine must know that a GETLASTBADLSN STATUS syscall will retrieve the LSN desired.

"Giving up" is aided by the SDOS SYSCALL:ERROREXIT. The error code is stored into the SYSCALL block, and the SYSCALL is executed. SDOS will print a text message corresponding to the error code, and pass control to the command interpreter (DEFAULTPROGRAM). The command interpreter can retrieve this error code, and a DO file can process it via IFERROR statements (see command interpreter description).

Passing back the error code to the next level of subroutine is generally done only if the recovery routine does not find an error code it is willing to handle. This provides an opportunity for subroutines at successively higher levels to effect recovery.

The subroutine calling convention that implements this error handling philosophy is as follows:

```
        *                           (S) = K HERE
        JSR     Subroutine
        BCS     ERROROCCURRED
        *       SUCCESS EXIT        (S) = K HERE
        .
        .
        .
        CLC                         FLAG "SUCCESS EXIT"
        RTS
ERROROCCURRED   EQU     *           (S) = K HERE
        CPX     #ERR:--
        BEQ     HANDLE1STERROR
        CPX     #ERR:--
        BEQ     --
        .
        .
        .
        SEC                         (6809 "CMPX" DESTROYS CARRY BIT)
        RTS                         WITH CARRY SET, INDICATING ERROR

HANDLE1STERROR  EQU     *           TO RECOVER FROM 1ST ERROR
        .
        .
        .
        CLC                         (OKRTS IN DEFS)
        RTS
```

Carry reset on exit means the subroutine completed successfully. The carry set on exit from a subroutine means "error occurred" (only for those subroutines which adhere to this convention!); the X register contains a 16 bit error code. Note that the calling subroutine must provide a BCS after the JSR in order to detect an error. The ERROROCCURRED routine tests the X register for errors from which it can recover; if the wrong error happens, no test will match and another RTS (with carry set) will occur, providing the next higher level subroutine a chance at processing the error code. In either case, error or not, the contents of the stack above the return address is untouched. The stack register itself has the original value of the stack pointer at the time of the JSR, so that all higher level routines can be returned to exactly as normal. Last, notice that the HANDLEERROR and the success paths both exit by clearing the carry (indicating "success" exit).

SYSCALLs are implemented as subroutine calls and follow the above convention with one variation. If an error occurs, SDOS unwinds the stack until a return address on top of the stack points to a BCC or BCS. This means that a SYSCALL must be followed by a BCC/BCS or SDOS will unwind the stack too far, with unpredictable results. The unwinding process consists of repeatedly popping two bytes, and assuming the top of the stack is a return address, (with obviously bad consequences if this is not true) until an appropriate return address is found (This scheme was chosen to minimize the amount of processing an SDOS routine had to do when it didn't care about errors, and has the side effect of speeding things up 5 to 1Ø percent).

SYSCALLS - Implementation

This section details the SYSCALLs implemented in this version of SDOS. See SDOSUSERDEFS.ASM listing for opcode values.

Errors listed are only common errors, i.e., ones for which application programs attempt recovery. Many other (even hardware specific errors) are possible, but due to the size and changing nature of the list, are not recorded here.

Table 1 - Syscalls implemented in SDOS 1.1

```
*       SYSCALL$ OPCODE DEFINITIONS
*
                    ORG     Ø
SYSCALL:OPEN        RMB     1   Open File
SYSCALL:CREATE      RMB     1   Create a New File
SYSCALL:CLOSE       RMB     1   Close a File
SYSCALL:RENAME      RMB     1   Rename a File
SYSCALL:DELETE      RMB     1   Delete a File
SYSCALL:LOAD        RMB     1   Load an Overlay
SYSCALL:CHAIN       RMB     1   Chain to a File
SYSCALL:CREATELOG   RMB     1   Create the Log File
SYSCALL:CLOSELOG    RMB     1   Close the Log File
SYSCALL:DISKDEFAULT RMB     1   Select Default Disk Device
SYSCALL:READA       RMB     1   Read ASCII Bytes From a File
SYSCALL:READB       RMB     1   Read Binary Bytes From a File
SYSCALL:WRITEA      RMB     1   Write ASCII Bytes To a File
SYSCALL:WRITEB      RMB     1   Write Binary Bytes To a File
SYSCALL:CONTROL     RMB     1   Perform a Control Operation
                                On a File/Device
SYSCALL:STATUS      RMB     1   Read File/Device Status
SYSCALL:WAITDONE    RMB     1   Wait for I/O on Channel to Complete
SYSCALL:EXIT        RMB     1   Give Control Back to Operating System
SYSCALL:ERROREXIT   RMB     1   Exit to System With Error Code
SYSCALL:SETERROR    RMB     1   Report an Error To The System
SYSCALL:GETERROR    RMB     1   Read Back the Last Error Code
SYSCALL:DISPERROR   RMB     1   Display Error Message Corresponding
                                To Last Error Code
SYSCALL:KILLPROOF   RMB     1   Prevent User Program Being Killed
SYSCALL:KILLENABLE  RMB     1   Allow User Program to be Killed
SYSCALL:DEBUG       RMB     1   Call System Debugger
SYSCALL:ATTNCHECK   RMB     1   Operator Attention Request Check
SYSCALL:ISCONSOLE   RMB     1   Check Channel Ø Input Device = Console:
SYSCALL:INTERLOCK   RMB     1   Perform Interlock functions on objects
SYSCALL:DELAY       RMB     1   Delay for n 1/6Øths of a second
SYSCALL:NOTUSED     RMB     1
SYSCALL:GETSERIALNUMBER RMB 1   Get processor serial number
```

SYSCALL:OPEN

This SYSCALL is used to establish an association between an existing file (to be read and/or updated) and an I/O channel.

OPEN SYSCALL Block Format:

```
SCBLK:OPCODE    FCB     SYSCALL:OPEN
SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS    FCB     CHANNELNUMBER,IGNORED
SCBLK:WRBUF     FDB     FILENAMESTRING
                        POINTS TO FIRST BYTE
SCBLK:WRLEN     FDB     FILENAMELENGTH
                        IN BYTES
SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 2
SCBLK:RDBUF     FDB     SCANNEDCOUNT
                        # FILENAME CHARACTERS
                        PROCESSED
SCBLK:RDLEN     FDB     2       SIZE OF RDBUF
SCBLK:END       EQU     *
```

The WAIT flag must be zero. The first parameter byte is the channel number desired. The second parameter byte is not used. The Write Buffer (WRBUF) contains the filename (including device name, etc.) desired, WRLEN contains the number of bytes in the filename.

The OPEN SYSCALL checks the channel to ensure that it is not open already. If not open, the filename is scanned to determine the selected device (default to DISK: if no device) and a filename on that device. The number of bytes scanned is returned as a 2 byte value in the buffer selected by RDBUF; the rest of the bytes in WRBUF are ignored. Leading blanks on the filename are ignored, but are included in the scanned count. (Note: All SYSCALLs that deal with file or device names return the number of bytes of the filename scanned as the result. The entire filename is scanned even if an error occurs.) The device is searched for the file if it is a directoried device, and an error issued if not found. If the device is not a directoried device, the device is simply opened. The file is positioned so that a subsequent read will read the zeroth (first) byte of the file.

(Some) possible errors are:

- Bad File Name
- No Such File
- Can't Open, Must Create
- No Such Device
- Channel Busy

SYSCALL:CREATE

This SYSCALL is used to CREATE a new file and establish an association between an I/O channel number and the new file. It is also used when a program will do output only to a device (such as a line printer; the philosophy is that such output is a new file).

CREATE SYSCALL Block Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:CREATE
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     CHANNELNO,IGNORED
        SCBLK:WRBUF     FDB     FILENAMESTRING
        SCBLK:WRLEN     FDB     FILENAMELENGTH
        SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 2
        SCBLK:RDBUF     FDB     SCANNEDCOUNT
        SCBLK:RDLEN     FDB     2       SIZE OF SCANNED COUNT
        SCBLK:END       EQU     *
```

The WAIT flag must be zero. The first parameter byte is the desired channel number; the second parameter byte is ignored. WRBUF points to the filename (device name) of the new file.

Like SYSCALL:OPEN, RDBUF points to a 2 byte area in which the number of bytes of the filename scanned by SDOS is placed on completion of the SYSCALL.

If a disk file is specified and there is an old file, the old file must not write protected or an error will occur and the new file will not be created (nor will the channel be opened). Otherwise, the new file is created, and the channel is opened. If an old file does exist, an OPEN SYSCALL executed after the CREATE, looking for the same file, will find the old file. If the system crashes before the new file is closed, the old file will be unaffected in any way. Even after the new file is closed, channels still open to the old file will not notice any difference. When the last channel to the old file is closed, it is deleted and the space for the old file is returned to free disk space. Effectively, a CREATE includes an "implied" delete of the older version of the file.

The file is positioned so that a write will write its first byte in byte #Ø of the file.

Possible errors are:

- File is Delete Protected
- File is Write Protected
- No Such Device
- Channel is Busy
- Bad Filename
- File is Being Created

SYSCALL:CLOSE

The CLOSE SYSCALL is used to break the association between an I/O channel number and a file.

CLOSE SYSCALL Format:

```
SCBLK:OPCODE    FCB     SYSCALL:CLOSE
SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS    FCB     CHANNELNO,IGNORED
SCBLK:END       EQU     *
```

This SYSCALL frees the I/O channel to be opened to another file, and causes the CLOSE entry point of a device driver to be called. Action of the driver is driver-dependent.

If the channel was open to a disk file, then changes to the file size, protection, and other characteristics are updated on the disk (not before). If the disk file is newly created, and is not replacing another by the same name, closing will make its name appear in the directory. If the file is newly created, and it is a replacement for a file that already exists (i.e., one by the same name), then the new file will replace the old in the directory, and the disk space allocated to the old file will be returned to free space as soon as no other I/O channels remain open to the old version of the file.

Possible errors are:

Illegal Channel Number
Channel is Already Closed

SYSCALL:RENAME

The RENAME Syscall is used to change the name of a file. The file must be open on some channel; it must not be a newly created file, and no file (new or old) having the new name must exist.

RENAME SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:RENAME
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     CHANNELNUMBER,IGNORED
        SCBLK:WRBUF     FDB     NEWFILENAME
        SCBLK:WRLEN     FDB     NEWFILENAMELENGTH
        SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 2
        SCBLK:RDBUF     FDB     SCANNEDCOUNT
        SCBLK:RDLEN     FDB     2
        SCBLK:END       EQU     *
```

The SYSCALL format is identical to that of an OPEN syscall; parameters and results are passed the same way.

This SYSCALL affects nothing except the name of the file.

RENAMEing a disk file to its own name is legal, and can speed up later OPENs of that file since a rename causes the file name to be re-hashed into the directory. Refer to hash-lookup description of files.

Possible errors are:

- Channel Not Open
- Bad File Name
- File is Being Created
- Can't Rename to a Different Device
- File is Delete Protected
- File is Write Protected
- New File Already Exists

SYSCALL:DELETE

The DELETE SYSCALL is used to delete a file from a disk device.

DELETE SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:DELETE
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     IGNORED,IGNORED
        SCBLK:WRLEN     FDB     FILENAMEBUFFER
        SCBLK:WRBUF     FDB     FILENAMESIZE
        SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 2
        SCBLK:RDBUF     FDB     REPLYBUFFER
        SCBLK:RDLEN     FDB     REPLYBUFFERSIZE
        SCBLK:END       EQU     *
```

The file specified on the specified device is deleted (this syscall is not legal for devices which do not have directories). No I/O channel is specified or needed. If the deletion is successful, the directory entry is removed so that the file can no longer be opened. If the file is open on some I/O channel when the delete SYSCALL is issued, then the SYSCALL will complete successfully, but the file will not actually be deleted until the last channel open to the file is closed (in fact, the file may actually be allocated more disk space via the other channel!).

The reply buffer is loaded with the actual length of the filename (see SYSCALL:OPEN).

Possible errors are:

No Such File
File is Delete Protected

SYSCALL:LOAD

The LOAD Syscall is used to load an overlay program segment into memory, without transferring control.

LOAD SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:LOAD
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     IGNORED,IGNORED
        SCBLK:WRBUF     FDB     FILENAMESTRING
        SCBLK:WRLEN     FDB     FILENAMELENGTH
        SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 4
        SCBLK:RDBUF     FDB     COUNTANDSTART
        SCBLK:RDLEN     FDB     4       MINIMUM REQUIRED
        SCBLK:END       EQU     *
```

No channel number need be specified.

The filename specified is opened on a special system channel, and checked to see if a load format file is given (first byte must be ASCII "S" or Hex :Ø1). If so, the file contents are loaded into memory as specified by the load records (see LOADER FORMATs). Scatter loading (loading into non-contiguous parts of memory) is possible. Upon completion of the loading process, control is returned to the user, and the file is closed.

The results returned in the reply buffer are 2 bytes of filename count (the first 2 bytes; see SYSCALL:OPEN) and 2 bytes of start address (the second 2) as specified by the load records.

Load records which would load on top or above SDOS cause the load to be aborted.

A load record whose address conflicts with that of the reply buffer may be damaged; conversely, the reply may be garbled. Loading into the area used by the stack may cause SDOS to crash. SDOS does not check for this.

Errors while loading cause the error exit of the Syscall to be taken.

In any case, on completion of the load, the file is closed.

Attempting to LOAD a program with a different encryption key is illegal.

Possible errors are:

Not a Load File
No Such File
EOF Hit
Checksum Error
Load Record Format Error
Bad Filename
Bad Filename Size

SYSCALL:CHAIN

The CHAIN Syscall is used to load and transfer control to an overlay or program segment.

CHAIN SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:CHAIN
        SCBLK:WLEN      FCB     SYSCALL:END-SYSCALL:OPCODE
        SCBLK:PARAMS    FCB     IGNORED,IGNORED
        SCBLK:WRBUF     FDB     FILENAMESTRING
        SCBLK:WRLEN     FDB     FILENAMELENGTH
        SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 4
        SCBLK:RDBUF     FDB     COUNTANDSTART
        SCBLK:RDLEN     FDB     4       MINIMUM REQUIRED
        SCBLK:END       EQU     *
```

CHAIN first closes all I/O channels except channel Ø. It then causes all modified disk sectors in the LRU queue to get written back to the disk to ensure validity of disk contents, and then performs exactly the same function as SYSCALL:LOAD. If an error occurs, control will return to the caller only if no data has been loaded into the user space. The most common causes of this are the following errors:

    Bad File Name
    Bad File Name Size
    File Not Found
    Not a Load File
    No Start Address

All other errors will cause an implied SYSCALL:ERROREXIT to be executed (because of the possibility of the program issuing the CHAIN being overlayed).

On successful completion of the load, control will be transferred to the start address of the file. The stack pointer is set to the contents of $FC,$FD, minus 1 (see SDOS Memory Map).

Chaining to a program with a different encryption key will cause the user space to be zeroed before control is transferred.

SYSCALL:CREATELOG

There are occasions on which a record of a terminal session would be very convenient, such as when a purported bug arises, or when an example is required. This copy can be laboriously constructed by hand, or it can be made automatically via a CREATELOG syscall.

CREATELOG SYSCALL Format:

```
SCBLK:OPCODE    FCB    SYSCALL:CREATELOG
SCBLK:WLEN      FCB    SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS    FCB    CHANNELNO,IGNORED
SCBLK:WRBUF     FDB    FILENAMESTRING
SCBLK:WRLEN     FDB    FILENAMELENGTH
SCBLK:RPLEN     RMB    2        EXPECTED RETURNED VALUE OF 2
SCBLK:RDBUF     FDB    SCANNEDCOUNT
SCBLK:RDLEN     FDB    2        SIZE OF SCANNED COUNT
SCBLK:END       EQU    *
```

CREATELOG creates a new file (just like the CREATE syscall), but no channel number is given (SDOS reserves a special, unnumbered, I/O channel specifically for this purpose). It returns file name size information in the same manner as OPEN.

There is no way for a user program to explicitly read or write data to the log channel; all I/O through the log channel is done invisibly by SDOS. Essentially, any data written via a Write ASCII to channel Ø (the control channel) is also copied to the log file. Data read via a Read ASCII on channel Ø is also written to the log file. In this way, a complete copy of console sessions (carried on through the control channel) is recorded in the log file for later retrieval. The writes to the log file are done only when the log file is open (has been created).

STATUS and CONTROL syscalls are re-directed from channel Ø to the log channel when it is open, so that status information read from channel Ø may not actually be that of channel Ø. All other channel-oriented syscalls (in particular, Read Binary and Write Binary) are not affected by the log channel. If the log channel is not open, it has no effect whatsoever on channel Ø operations.

The log file will not be found in the directory until it is closed (via CLOSELOG). Like any CREATEd disk file, PROGRAM KILL (^C^C) automatically closes the log file. This Syscall is used mainly by SDOSCOMMANDS to implement the LOG and DO commands.

A program can set up a DO file by:

1) Verifying that the DO file exists by OPENing it on some channel.
2) CLOSEing channel Ø
3) OPENing channel Ø to the DO file
4) CREATELOG on the "CONSOLE:" device

Further input will come from the DO file. If an error occurs during step 2 or 3, the program must reOPEN channel Ø to the CONSOLE: or no further console I/O can occur.

Possible errors are:

Channel Already Open
Illegal File Name
No Disk Space

SYSCALL:CLOSELOG

This Syscall is used to close the special log I/O channel (see SYSCALL:CREATELOG).

CLOSELOG SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:CLOSELOG
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:END       EQU     *
```

This Syscall performs the same operation as a CLOSE Syscall on the Log channel. No channel number or other parameters are needed.

Possible errors are:

Channel Not Open

SYSCALL:DISKDEFAULT

This SYSCALL is used to select which disk is default-selected when a file name with no explicit disk device indentification is given.

DISKDEFAULT SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:DISKDEFAULT
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     IGNORED,IGNORED
        SCBLK:WRBUF     FDB     FILENAMESTRING
                                POINTS TO FIRST BYTE
        SCBLK:WRLEN     FDB     FILENAMELENGTH
                                IN BYTES
        SCBLK:RPLEN     RMB     2       EXPECTED RETURNED VALUE OF 2
        SCBLK:RDBUF     FDB     SCANNEDCOUNT
                                # FILENAME CHARS PROCESSED
        SCBLK:RDLEN     FDB     2       SIZE OF RDBUF
        SCBLK:END       EQU     *
```

DISKDEFAULT parses the device name, and ensures that the device name is a valid disk device name (filenames passed with the device name are not examined). The specified disk will then be used whenever a filename with no device specification is encountered by a filename SYSCALL.

No channel number is needed.

Data is returned in the same form as an OPEN syscall.

After a successful return, the device name DISK: refers to the default disk.

Possible errors are:

Device is Not a Disk

SYSCALL:READA

This SYSCALL is used to read (ASCII) textual data from a file. The file must be open on some I/O channel.

READA SYSCALL Block Format:

```
SCBLK:OPCODE    FCB    SYSCALL:READA
SCBLK:WLEN      FCB    SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS    FCB    CHANNELNUMBER,LMFLAG
SCBLK:WRBUF     RMB    2
SCBLK:WRLEN     FDB    Ø
                       (MINIMIZES PROCESSING TIME)
SCBLK:RPLEN     RMB    2
                       ACTUAL NUMBER BYTES READ
SCBLK:RDBUF     FDB    READBUFFER
                       WHERE TO PUT DATA
SCBLK:RDLEN     FDB    READBUFSIZE
                       MAXIMUM NUMBER BYTES TO READ
SCBLK:END       EQU    *
```

READA will read the specified number of bytes into the read buffer from the file open on the specified channel, and advance the file position past the number of bytes examined, subject to the following conditions: the file has enough bytes, and no errors occur during the read. Nulls (:ØØ), line feeds (:ØA), and rubouts (:7F) are deleted from the stream of characters read from the file/device.

Bit 7 of all characters read via SYSCALL:READA is zeroed. Other characters may be removed from the input stream by the particular device driver in use.

The column count for this channel is updated for each byte placed in the read-back buffer, according to the following rule: a printing character (:2Ø-:7E) causes the column count to be incremented. CR (:ØD) causes the column count to be zeroed. All other codes leave the count alone. The column count can be read by a SYSCALL:STATUS call.

If LMFLAG is non-zero, the read proceeds in single line mode. If a CR (:ØD) character is encountered, it will be placed in the read buffer, and the read will be terminated. LMFLAG=Ø prevents CRs from terminating the read, so the buffer will be filled.

SCBLK:RPLEN is set to the actual number of bytes read, even if an error (such as End of File) occurs.

The WRBUF is ignored if supplied.

All data read from channel Ø via a READA is copied (via WRITEA) to the log file if the log channel has been opened. A READA with LMFLAG=1 directed at channel Ø will be completed from the CONSOLE: device if a complete line cannot be read because of an EOF error (this finishes a partial line from a DO file).

The overhead for doing a single-byte SYSCALL:READA is fairly high; larger buffers will cause this overhead to be divided between all the bytes transferred. Large buffers can achieve a 4Ø to 1 speedup over single byte transfers. Such speed ups are also typical for SYSCALL:WRITEA, SYSCALL:WRITEB, and SYSCALL:READB.

If the SYSCALL block length is 18 bytes or more, then the first four bytes of the extension hold a file position, and an implied positioning operation is performed BEFORE the actual read takes place. Compared to a CC:POSITION call followed by a SYSCALL:READA, a combined position/read operation is considerably more efficient in a network environment, so it is encouraged. Similar efficiencies accrue for combined position/write operations.

An EOF hit error will occur: (1) if not in line mode and the buffer cannot be filled; (2) if in line mode and no CR character is encountered before EOF.

An end-of-file condition (which can be sensed via a SYSCALL:STATUS) is set whenever a read of the last data byte of the file occurs.

Possible errors are:

Channel Not Open
EOF Hit

SYSCALL:READB

This SYSCALL is used to read binary data from a file. The file must be open on some I/O channel.

READB Syscall Block format

```
SCBLK:OPCODE    FCB     SYSCALL:READB
SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS    FCB     CHANNELNUMBER,IGNORED
SCBLK:WRBUF     RMB     2
SCBLK:WRLEN     FDB     Ø
                        (MINIMIZES PROCESSING TIME)
SCBLK:RPLEN     RMB     2
                        ACTUAL NUMBER BYTES READ
SCBLK:RDBUF     FDB     READBUFFER
                        WHERE TO PUT DATA
SCBLK:RDLEN     FDB     READBUFSIZE
                        MAXIMUM NUMBER BYTES TO READ
SCBLK:END       EQU     *
```

READB will read the specified number of bytes into the read buffer from the file opened on the specified I/O channel, and advance the file position by the number of bytes actually read. In order for the specified buffer to be completely filled, the distance between the current file position and the end of the file must be greater or equal to the buffer size, and no errors may occur during the read. The data bytes read from the file are not changed in any way.

SCBLK:RPLEN is set to the actual number of data bytes read (usually equal to the buffer size).

Using a READB SYSCALL causes the column count for the specified channel to be zeroed.

SCBLK:WRBUF is ignored if supplied; however, its length should be specified as zero to minimize SYSCALL processing time.

An EOF error will occur if the read request is not completely satisfied (i.e., the buffer was not filled).

The overhead for doing single-byte reads is high; long buffers will distribute this overhead so that the average time per byte is some 4Ø times faster than single byte reads.

If the SYSCALL block length is 18 bytes or more, then the first four bytes of the extension hold a file position, and an implied positioning operation is performed BEFORE the actual read takes place.

Possible errors are:

Channel Not Open
EOF Hit
Disk Read Error
Device Not Ready
Device Timed Out

SYSCALL:WRITEA

WRITEA is used to Write ASCII data to a file. The primary difference between this and WRITEB is that the column count gets updated, and certain output editing is done.

WRITEA SYSCALL Format:

```
SCBLK:OPCODE    FCB    SYSCALL:WRITEA
SCBLK:WLEN      FCB    SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS    FCB    CHANNELNUMBER,IGNORED
SCBLK:WRBUF     FDB    WRITEDATABUFFER
SCBLK:WRLEN     FDB    NUMBEROFBYTESTOWRITE
SCBLK:END       EQU    *
```

The data bytes in the WRITEDATABUFFER are copied to the file open on the specified I/O channel. The file position is advanced by NUMBEROFBYTESTOWRITE. Disk files are extended automatically, if necessary, to make more room and the file size is changed. The column count for this I/O channel is changed according tc the same rules as specified by SYSCALL:READA. The output stream may be modified by the device driver; a CRT driver will typically add LF (:ØA) and nulls (idle characters) after a CR (:ØD) character.

SDOS conventions dictate that LF characters are superflous in the presence of CR characters. To write a line of text to a file (or device), terminating it with a CR is sufficient.

An EOF condition will happen if the last data byte of the file is overwritten, and/or the file was extended in order to accomodate the write request. An EOF condition on a WRITE to a disk does not cause an error.

Data written via WRITEAs to channel Ø is also sent (via WRITEAs) to the log channel if the log channel is open.

Multi-byte writes are more efficient than single-byte writes.

No read-back buffer is required.

If the SYSCALL block length is 18 bytes or more, then the first four bytes of the extension hold a file position, and an implied positioning operation is performed BEFORE the actual read takes place.

Possible errors are:

- Channel Not Open
- Disk Space Exhausted (for disk files)
- Disk Write Error
- Device Timed Out
- Device Not Ready

SYSCALL:WRITEB

The WRITEB SYSCALL is used to write binary data to a file. The stream of data bytes is copied directly to the file or device without any change to its content.

WRITEB SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:WRITEB
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     CHANNELNUMBER,IGNORED
        SCBLK:WRBUF     FDB     WRITEDATABUFFER
        SCBLK:WRLEN     FDB     NUMBEROFBYTESTOWRITE
        SCBLK:END       EQU     *
```

The data bytes in the specified buffer are copied without change to the file that is open on the specified I/O channel. The file position is advanced by NUMBEROFBYTESTOWRITE. If necessary, a disk file is extended automatically to make more room, and the file size is adjusted accordingly. The column count for this channel is zeroed.

Multi-byte writes are more efficient than single-byte writes.

An EOF condition will happen if the last data byte of the file is overwritten, and/or the file was extended in order to accomodate the write request.

No read-back buffer is required.

If the SYSCALL block length is 18 bytes or more, then the first four bytes of the extension hold a file position, and an implied positioning operation is performed BEFORE the actual read takes place.

Possible errors are:

- Channel Not Open
- Disk Space Exhausted
- Illegal Device Operation
  (for line-printer-like devices)
- Disk Write Error
- Device Not Ready

SYSCALL:CONTROL

This SYSCALL is used to control or modify the operation of a device/file. The first parameter byte selects the I/O channel number; the second parameter byte determines the actual operation performed (rewind, eject, dismount, etc.) so this SYSCALL actually represents an entire class of operations. A control operation may be issued only to an I/O channel that is already OPEN.

If logging is active, and a CONTROL operation is issued for channel Ø, the control operation is actually applied to the log channel.

CONTROL SYSCALL Block Format:

```
        SCBLK:OPCODE     FCB      SYSCALL:CONTROL
        SCBLK:WLEN       FCB      SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS     FCB      CHANNELNUMBER
                         FCB      CC:controlcode
        SCBLK:WRBUF      FDB      CONTROLPARAMETERS
        SCBLK:WRLEN      FDB      NUMBEROFCONTROLBYTES
        SCBLK:END        EQU      *
```

SDOS divides device control operations into two classes: common, and device specific. Common control operations are those operations for which all devices generally have a capability. Currently only the following operations fit in the category of common:

CC:POSITION and CC:DUMPBUFFERS

All other control operations are device specific and are documented with the specific device driver. Typical device-specific operations include: select echo mode, set tabs, and dismount disk.

The format of the CONTROL SYSCALLs varies because different device operations require different parameters. In particular, most CONTROL SYSCALLs do not require a write buffer. For specific formats, refer to the device driver descriptions.

CC:POSITION

CC:POSITION is used to select the next byte of a file to be read/written. A 4 byte, 2's complement integer is used to select the byte index into a (disk) file (it can also be used as a record number, a port number, a screen position, or whatever is appropriate for the device). The number must be positive (i.e., the sign bit must be zero) or an error will result. Following a CC:POSITION command, further read/writes start from the specified file position and advance sequentially. A "rewind" is obtained by specifying a zero for the value of the 4 byte integer.

Setting a file position which is equal or greater than the size of the (disk) file will cause an EOF condition to occur and cause an error.

No reply is given for this syscall.

Alphanumeric CRTs are an interesting special case. It is standard for SDOS CRT drivers to interpret the positioning parameter as cursor positioning data. The parameter is interpreted as 2 bytes of zero, 1 byte to specify the screen row number (zero being the top screen row) and 1 byte of column number (zero being the leftmost column). Given R for row and C for column, the value of the positioning parameter is then Row*256+Column. In this way, cursor positioning on screens is generalized to work for a broad variety of CRT displays.

CC:POSITION SYSCALL Format:

```
        SCBLK:OPCODE     FCB     SYSCALL:CONTROL
        SCBLK:WLEN       FCB     SCBLK:RPLEN-SCBLK:OPCODE
        SCBLK:PARAMS     FCB     CHANNELNUMBER,CC:POSITION
        SCBLK:WRBUF      FDB     POSITIONDATA
        SCBLK:WRLEN      FDB     4
        .
        .
        .
        POSITIONDATA     RMB     4       NEED FILE POSITION
```

For CRTs, POSITIONDATA has the following form:

```
        POSITIONDATA     FCB     Ø,Ø
        SCREENROW        RMB     1
        SCREENCOL        RMB     1
        .
        .
        .
```

CC:DUMPBUFFERS

CC:DUMPBUFFERS is used to force an I/O device to dump any buffers it may still have filled. CC:DUMPBUFFERS is particularly useful in transaction oriented programs which need to force all disk file changes back to the disk. No parameters are required; operation is device specific.

CC:DUMPBUFFERS Format:

```
        SCBLK:OPCODE     FCB      SYSCALL:CONTROL
        SCBLK:WLEN       FCB      SCBLK:WRBUF-SCBLK:OPCODE
        SCBLK:PARAMS     FCB      CHANNELNUMBER,CC:DUMPBUFFERS
```

SYSCALL:STATUS

The STATUS Syscall is used to read file or device-dependent descriptive data about that file or device (as opposed to reading data from the file or device itself). This syscall is really an entire group of operations; a parameter byte selects the device-specific data to read. A STATUS Syscall must reference an open I/O channel. Like READA and READB, the data is read back into the reply buffer.

If a STATUS syscall is issued for channel Ø, and logging is active, the status read back will be that of the log channel, not channel Ø.

STATUS SYSCALL Block Format:

```
        SCBLK:OPCODE     FCB      SYSCALL:STATUS
        SCBLK:WLEN       FCB      SYSCALL:END-SYSCALL:OPCODE
        SCBLK:PARAMS     FCB      CHANNELNO,SC:statuscode
        SCBLK:WRBUF      FDB      IGNORED
        SCBLK:WRLEN      FDB      IGNORED
        SCBLK:RPLEN      FDB      CHANGED
        SCBLK:RDBUF      FDB      STATUSBUFFER
        SCBLK:RDLEN      FDB      STATUSCODEDEPENDENTLENGTH
```

There are two classes of STATUS requests: those standard across all devices, and those specific to the particular device type. The following status information is obtainable from most devices:

SC:GETPOS
SC:GETCOL
SC:GETEOF
SC:GETFILESIZE
SC:GETTYP
SC:GETPARAMS

All other status-reading operations are device specific and are detailed under the specific device drivers.

SC:GETPOS is used to read the current position in a file, i.e., if one executes a CC:POSITION command, an SC:GETPOS will read back the same value as the positioning value given for the CC:POSITION. SC:GETPOS always reads back four data bytes (the interpretation of these bytes is up to the device driver).

SC:GETCOL reads back the print position of a simulated print head on a particular I/O channel (see READA, WRITEA syscalls). Ø means "no characters printed on this line." Only one data byte is returned.

SC:GETEOF returns a single-byte flag indicating whether the I/O channel has positioned, read or written past the last data byte in the file. A non-zero returned byte indicates past or at end of file; zero means more data can be read from the file before the end of file is encountered.

SC:GETFILESIZE returns the size of the file (in bytes). The size is returned as a four byte integer, appropriate for use in a positioning command (this is convenient for appending data to the end of a file). This is normally only implemented on disk files.

SC:GETTYP returns a single-byte device type code, which places a device into one of the following classes: FILE, DISK, TAPE, DIRECTORIED TAPE, CONSOLE, LINEPRINTER, SERIALOUT, SERIALIN, PARALELLOUT, PARALELLIN, DUMMY. Other device types may be added as needed.

SC:GETPARAMS reads device class-specific parameters. To know what kind of data to expect for a reply, the program must first determine the device type (using SC:GETTYP). Currently defined device-specific parameters are:

Disk FILE:

DVDAT:NSPC	Number of Sectors Per Cluster
DVDAT:NBPS	Sector Size in Bytes

The maximum file size may be computed as:

(NBPS*NSPC/2-1)*NBPS*NSPC

DISK Device:

DVDAT:NBPS	Number of Bytes Per Sector
DVDAT:NSPT	Number of Sectors Per Track
DVDAT:NTPC	Number of Tracks Per Cylinder
DVDAT:NCYL	Number of Cylinders

CONSOLE:

DVDAT:WIDTH	In Characters
DVDAT:DEPTH	Screen or Page Depth in Lines, or Ø If Continuous Form Paper

PRINTER:

DVDAT:WIDTH	In Characters
DVDAT:DEPTH	Page Depth in Lines

SYSCALL:WAITDONE

This system call is used to wait for an operation initiated on an I/O channel to complete.

This SYSCALL and the parallel initiation feature ARE NOT IMPLEMENTED IN FINAL FORM. It currently is a no-operation, and is provided to allow programs to be coded as though parallel SYSCALLS were implemented.

WAITDONE SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:WAITDONE
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB     CHANNELNUMBER
        SCBLK:END       EQU     *
```

If any parallel SYSCALL (a syscall with the WAIT flag = "don't wait") was issued on the specified I/O channel, WAITDONE delays the execution of the user program until that operation is complete. Error status returned is that of the parallel SYSCALL returned as though the parallel SYSCALL had the WAIT flag reset when executed.

A second WAITDONE issued on an I/O channel, without any other intervening SYSCALLS, returns immediately with no error possible, so multiple WAITDONEs on a channel may be performed without conflicts arising.

SYSCALL:EXIT

This syscall is used by a user program to pass control to the DEFAULTPROGRAM. It is an indication that the user program completed execution successfully.

EXIT SYSCALL Format:

```
        SCBLK:OPCODE     FCB      SYSCALL:EXIT
        SCBLK:WLEN       FCB      SCBLK:END-SCBLK:OPCODE
        SCBLK:END        EQU      *
```

There are no parameters, and control does not return to the user program.

All I/O channels except channel Ø are CLOSEd.

SDOS does a quick checksum on itself after an EXIT is completed, and reports an error if it thinks memory is starting to fail; otherwise, no errors are possible.

This syscall is functionally identical to SYSCALL:ERROREXIT with an error code of Ø.

SYSCALL:ERROREXIT

This syscall is used by a user program to cease execution abnormally, and notify the operator the reason for stopping.

ERROREXIT SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:ERROREXIT
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FDB     ERRORCODE
        SCBLK:END       EQU     *
```

The error code is displayed on the console as either

Error <CR>

or

<TEXT MESSAGE> <CR>

depending on whether SDOS can successfully extract the corresponding text message from the ERRORMSGS.SYS file on drive Ø (see SYSCALL:DISPERROR). If the error code is Ø, a message is not displayed. Control is then passed to the DEFAULTPROGRAM (usually the SDOS command interpreter, which can interrogate and conditionally branch on the error code if a DO file is being processed). No error is possible.

This syscall is intended to be used as very simple error handling in user programs.

Example:

```
                LDX     #PARAMETERLISTADDRESS
                JSR     SYSCALL$
                BCS     OOPS      B/ ERROR
                .
                .
                .

OOPS            CPX     #ERR:...
                BEQ     ICANHANDLEIT1
                CPX     #ERR:...
                BEQ     ICANHANDLEIT2
                .
                .
                .
IGIVEUP         STX     ERROREXIT+SCBLK:PARAMS
                LDX     #ERROREXIT
                JSR     SYSCALL$
                BCS     *         CAN'T GET HERE!
                JMP     *

ERROREXIT       FCB     SYSCALL:ERROREXIT
                FCB     4         SCBLK:WLEN
                FDB     Ø         SCBLK:PARAMS
```

SYSCALL:SETERROR

This syscall, coupled with SYSCALL:DISPERROR, is used by a program to display the reason a SYSCALL failed.

SETERROR SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:SETERROR
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FDB     ERRORCODE
        SCBLK:END       EQU     *
```

The user program first stores an error code into the syscall block, and then issues the syscall. The error code has now been stored in SDOS for use by the DISPERROR and GETERROR syscalls. Normally, a SETERROR is followed by a DISPERROR, so that a text display of the error cause occurs. Since control returns to the user program, this is an effective procedure for displaying the cause of an error without EXITing to the DEFAULTPROGRAM.

A GETERROR syscall can be used to later retrieve the error code. A subsequent EXIT or ERROREXIT syscall will change the code set by SETERROR.

SYSCALL:GETERROR

This syscall is used to retrieve an error code given to SDOS by EXIT, ERROREXIT, or SETERROR syscalls.

GETERROR SYSCALL Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:GETERROR
        SCBLK:WLEN      FCB     SYSCALL:END-SCBLK:OPCODE
        SCBLK:PARAMS    FDB     IGNORED
        SCBLK:WRBUF     FDB     IGNORED
        SCBLK:WRLEN     FDB     IGNORED
        SCBLK:RPLEN     FDB     2       EXPECTED RETURNED VALUE
        SCBLK:RDBUF     FDB     ERRORCODEBUF
                                WHERE TO PUT ERROR CODE
        SCBLK:RDLEN     FDB     2       LENGTH OF 16 BIT ERROR
                                        CODE
        SCBLK:END       EQU     *
```

The 2 byte error code last given to SDOS is returned in the reply buffer. No parameters other than the reply buffer discriptor are necessary.

Possible errors are:

Syscall Length Too Short
Read-Back Buffer Too Short

SYSCALL:DISPERROR

The DISPERROR is used to display a text message corresponding to the most recent error code given to SDOS by SYSCALL:EXIT, SYSCALL:ERROREXIT, or SYSCALL:SETERROR.

DISPERROR SYSCALL Block Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:DISPERROR
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:END       EQU     *
```

No parameters are needed.

Either

```
        ERROR nnnnn <CR>
```

or

```
        <TEXT FROM ERRORMSGS.SYS> <CR>
```

is displayed on channel Ø. If the error code is Ø, and error message IS displayed (see SYSCALL:ERROREXIT for contrast). If channel Ø is not open, SDOS automatically opens it to the CONSOLE: device. SDOS gets the text message from the ERRORMSGS.SYS file based on the error code. If SDOS cannot retrieve the error message from the ERRORMSGS.SYS file, it displays the simpler form, with nnnnn being the decimal equivalent of the error code. No carriage return is output, so that the user program may precede or append text to the error message (such as ... AT LINE 1ØØ for BASIC).

If an error occurs during the process of displaying the message, SDOS will hang. The operator must re-boot. This can only occur if SDOS cannot output to the CONSOLE:.

SYSCALL:KILLPROOF

This SYSCALL is used by an application which needs to perform a long computation or large amounts of I/O without being killed by the operator for correct operation. This situation occurs when several files need to be updated in order to maintain data base consistency.

KILLPROOF SYSCALL Block Format:

```
SCBLK:OPCODE    FCB     SYSCALL:KILLPROOF
SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
SCBLK:END       EQU     *
```

Normally, when the operator types ^C^C, SDOS kills the currently running program and causes a forced ERROREXIT. This in turn displays an appropriate message and causes the DEFAULTPROGRAM to be loaded.

A double ^C is deferred if a SYSCALL:KILLPROOF has been executed more recently than a KILLENABLE. Operation of the program continues undisturbed until it executes SYSCALL:KILLENABLE, at which point the program is stopped. The user program can still sense operator attention requests via the ATTNCHECK syscall.

On EXIT, SDOS switches user programs back to KILLENABLEd mode automatically, (actually, the DEFAULTPROGRAM is loaded as a KILLENABLEd user program) so a set of programs invoked by a DO file is killable. SYSCALL:CHAIN does not affect the KILLENABLE status of a program, so a large program consisting of several serially executed segments can operate entirely KILLPROOFed if needed.

Possible errors are:

Syscall Block Too Short

SYSCALL:KILLENABLE

This syscall allows a program to be killed by the operator. It is normally only used after a critical portion of a program, running KILLDISABLEd, is finished executing.

KILLENABLE SYSCALL Block Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:KILLENABLE
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:END       EQU     *
```

Executing this syscall will allow a program to be killed when the operator types ^C^C (when the I/O package calls SDOS:KILLPROGRAM). If a ^C^C (call to SDOS:KILLPROGRAM) has occurred while the user program was KILLPROOF, execution of the SYSCALL:KILLENABLE will cause the program to quit execution immediately (i.e., control does not return to the user program in this case).

SDOSCOMMANDS (the command interpreter) runs KILLENABLEd and loads user programs initially KILLENABLEd. The user program must execute a SYSCALL:KILLDISABLE syscall before performing any critical operations (see SYSCALL:KILLDISABLE). CHAIN syscalls do not affect the KILLENABLE status of the user program.

Possible errors are:

    Program Killed
    Syscall Block Too Short

SYSCALL:DEBUG

The DEBUG syscall is used to transfer control from a user program to the local system debugger.

DEBUG SYSCALL Block Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:DEBUG
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:END       EQU     *
```

No parameters are needed. Control is passed to the system debugger's entry point. The actual method of passing control is I/O package dependent. If there is no debugger, an ERROREXIT is forced.

For systems with IDB (an SD assembly language debugging tool), control is passed to the debugger in such a way that a non-maskable interrupt appears to have occurred. EXIT from IDB should be made via a "G" command. Using nnnnG to exit IDB and return to the user program will also work. If a "G" command is executed, control returns to the user program just beyond the call, as with any other SYSCALL.

Possible errors are:

Syscall Too Short
No Debugger

SYSCALL:ATTNCHECK

This SYSCALL is used to determine if the operator would like to interact with the user program (the operator normally signals this by striking the ESCape key on his console; the actual mechanism is determined by the I/O package).

ATTNCHECK SYSCALL Block Format:

```
        SCBLK:OPCODE     FCB     SYSCALL:ATTNCHECK
        SCBLK:WLEN       FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:END        EQU     *
```

The ATTNCHECK syscall will return normally if no attention has been requested since the last ATTNCHECK syscall. If the operator has requested attention at least once since the last ATTNCHECK SYSCALL was issued, then an error exit is taken with error code ERR:ATTENTION.

There are no parameters and no returned results.

Note that depressing ESCape terminates line input mode from the CONSOLE:; thus, with suitable program design, ESCape can be used to get a program out of one interaction mode and into another mode of interaction.

SYSCALL:ISCONSOLE

This system call is used to determine if channel zero is open to the operator's console (this is needed because a STATUS syscall will read back the status of the log channel if logging is active).

This SYSCALL is used primarily by the command interpreter (when an error is encountered) to determine whether or not a DO file should be aborted.

ISCONSOLE SYSCALL Block Format:

```
        SCBLK:OPCODE    FCB     SYSCALL:ISCONSOLE
        SCBLK:WLEN      FCB     SCBLK:END-SCBLK:OPCODE
        SCBLK:END       EQU     *
```

There are no parameters and no returned results. A normal exit indicates that channel zero truly is open to the console device; otherwise, an error exit occurs. The only possible errors are:

Channel is Not Open at All
Channel Ø is Open; But Not to the Console

SYSCALL:INTERLOCK

This SYSCALL enables multiple users to synchronize usage of one or more resources (under single-user SDOS, these calls are null operations). Each resource is represented by an INTERLOCK "object" (note: future SDOS's will provide for many other abstract object types), and the means of referencing that object is called a CAPABILITY. The functions which the INTERLOCK syscall will perform, include creating a capability to an interlock object; destroying an existing capability to an interlock object; reserving an object for exclusive use (also known as "locking" an object), and, if the object has been already locked, suspending execution of the caller until that object has been released; releasing the object, allowing the next suspended requestor to resume execution; conditionally locking an object, returning an error if that object is already locked; and releasing and removing all requests for an object. Note that objects and capabilities do not "belong" to users (e.g., user 1 may create an interlock capability to an object named "MYFILE", communicate that capability to user 2, and proceed to lock MYFILE twice, thereby blocking himself; user 2 subsequently releases MYFILE, which causes user 1 to be unblocked).

The function codes are expressed as 16-bit values in the PARAMS field of the SYSCALL block; the specific functions and their requirements are:

IC:CREATE

Create a capability to an interlock object. ERR:NOSUCHOBJECT will be returned if the named object is invalid.

WRBUF must contain an object name, and WRLEN must be 16. A 16-byte capability to the object will be returned in RDBUF.

IC:DESTROY

Destroy the usefulness of all capabilities to the named interlock object. Release the object if it has been locked; release all requests for the object; release all suspended requestors of the object, with ERR:OBJECTDESTROYED. If the capability is invalid, ERR:NOSUCHOBJECT will be returned.

WRBUF must contain a valid capability to the object, and WRLEN must be 16.

IC:LOCK

Lock the named interlock object. If the object is already locked, the caller's execution is suspended until the object has been released. Under SDOS/MT 1.2, no more than 32 different objects may be locked at any one time (implementation restriction); attempted violation of the restriction will result in ERR:IMPLEMENTATIONLIMITREACHED. If the capability is invalid, ERR:NOSUCHOBJECT will be returned.

WRBUF must contain a valid capability to the object, and WRLEN must be 16.

IC:RELEASE

Release the named interlock object. If the object has not been previously locked, ERR:NOTLOCKED is returned. If the capability is invalid, ERR:NOSUCHOBJECT will be returned.

WRBUF must contain a valid capability to the object, and WRLEN must be 16.

IC:TEST

Lock the named interlock object. If the object is already locked, no further action is taken and ERR:ALREADYLOCKED is returned. If the capability is invalid, ERR:NOSUCHOBJECT will be returned.

WRBUF must contain a valid capability to the object, and WRLEN must be 16.

IC:RESET

Unconditionally release the named interlock object, if locked; remove all requests for the object. Callers suspended, awaiting use of the object, will be returned to execution with ERR:LOCKRESET. If the capability is invalid, ERR:NOSUCHOBJECT will be returned.

WRBUF must contain a valid capability to the object, and WRLEN must be 16.

SYSCALL:DELAY

This system call is used by a program to wait for some fixed period of time before continuing execution. This is useful on multi-user systems when a periodic check is required, as no resources are used while a program is waiting for the delay to complete.

DELAY SYSCALL Block Format:

```
        SCBLK:OPCODE    FCB    SYSCALL:DELAY
        SCBLK:WLEN      FCB    SCBLK:END-SCBLK:OPCODE
        SCBLK:PARAMS    FCB    DELAY ; in 1/60th second units
        SCBLK:END       EQU    *
```

The delay is a 16 bit value given in 1/60th second units (i.e., 60 = 1 seconds, 3600 = 1 minute, etc.). The actual delay is at least that requested, and may be longer.

Possible errors:

Syscall Block is Too Short

SYSCALL:GETSERIALNUMBER

This system call is used to read the 8 byte hardware serial number of the computer.

GETSERIALNUMBER SYSCALL Block Format:

```
SCBLK:OPCODE   FCB   SYSCALL:GETSERIALNUMBER
SCBLK:WLEN     FCB   SCBLK:END-SCBLK:OPCODE
SCBLK:PARAMS   FDB   IGNORED
SCBLK:WRBUF    FDB   IGNORED
SCBLK:WRLEN    FDB   IGNORED
SCBLK:RPLEN    FDB   8          EXPECTED RETURNED VALUE
SCBLK:RDBUF    FDB   SERIALNUMBERBUFFER
SCBLK:RDLEN    FDB   8
```

Possible errors:

Syscall Block is Too Short

```
ASM/6809 1.4A1: 0000
09/17/84 19:43:57; Page 1; Form 1        *** SDOS SYSCALL Example ***
listfile.asm
                  * This is a sample assembly language program to list
                  * a file to the console: device (i.e., it does exactly
                  * the same thing as a list file command does), and
                  * illustrates use of syscalls and error recovery logic.
                  *
 0200                      org    $200             nice place for program
                  *        set up the equs we need
                  *
 0000             channel0 equ    0                user terminal channel
 0001             channel1 equ    1                channel for file i/o
 0001             linemode equ    1                input in "line mode"
                  *
                  * Print a "hello" message on user channel
                  *
0200 8E024C       listfile ldx    #himessage
0203 9DFB                  jsr    syscall$
                  * If we get an error when printing the "hi" message
                  * (i.e., the carry is set), this BCS will take us
                  * to the error routine which will do an error exit
0205 2528                  bcs    error
                  *
                  * Now input the name of the file the user wishes
                  * to list to his/her terminal
                  *
0207 8E0277                ldx    #inputfilename
020A 9DFB                  jsr    syscall$
020C 2521                  bcs    error
                  *
                  * Next, open the file...to do this we set the length
                  * of the file name in the OPEN syscall block equal
                  * the number of characters read in by the last syscall.
                  * We don't have to move the file name anywhere since
                  * we very cleverly made the place that SDOS will look
                  * at for the file name the same place where SDOS
                  * read in the string from the user
                  * (similar to INPUT a$\OPEN #1,a$ in BASIC)
                  *
                  * Get how many chars the user typed in
020E BE027F                ldx    inputfilename+reada:actualcount
                  * Set the length of file name to number of chars read
0211 BF028D                stx    openfile+open:length
0214 8E0287                ldx    #openfile        address of syscall bloc
0217 9DFB                  jsr    syscall$         make SDOS open the file
0219 2514                  bcs    error            take branch if "no such
                  *                                file", "bad name", etc.
```

```
ASM/6809 1.4A1: 0219
09/17/84 19:43:57; Page 2; Form 1        *** SDOS SYSCALL Example ***
listfile.asm
                    * main program loop
                    *
0221B 8E0291        readloop ldx    #readaline      read line from...
021E 9DFB                    jsr    syscall$        the input file
                    *
                    * Now check to see if the read got an error.
                    * If it did, see if the error was an end of file.
                    *
0220 2517                    bcs    checkforeof
                    *
                    * If we get to here, we know we didn't get an error.
                    * So set the length of the write buffer equal to the
                    * number of characters read in
                    *
0222 BE0299                  ldx    readaline+reada:actualcount
0225 BF02A7                  stx    writealine+writea:count
                    *
                    * and then send the line out to the user
                    *
0228 8E02A1                  ldx    #writealine
022B 9DFB                    jsr    syscall$
                    * If no error on output, go read another line
022D 24EC                    bcc    readloop
                    *
                    * Error routine: copy error code in X to a syscall
                    * block which will have SDOS print out the
                    * corresponding error message and exit
                    *
022F BF02AB         error    stx    errorexit+errorexit:code
0232 8E02A9                  ldx    #errorexit
0235 9DFB                    jsr    syscall$        SDOS shouldn't return,
0237 25FE                    bcs    *               should never get here
                    *
                    *  Check for EndOfFile: if so, wrap things up and exit.
                    *  Otherwise, do an error exit.
                    *
0239 8C03E9         checkforeof cpx #err:eofhit EndOfFile error?
023C 26F1                    bne    error       if not, go complain
023E 8E02AD                  ldx    #byemessage print "I'm done" message
0241 9DFB                    jsr    syscall$
0243 25EA                    bcs    error       murphy's law strikes again!
0245 8E02BD                  ldx    #exit       now exit
0248 9DFB                    jsr    syscall$
024A 25E3                    bcs    error       this can't happen
                    *
                    * end of code
```

```
ASM/6809 1.4A1: 024A
09/17/84 19:43:57; Page 3; Form 1        *** SDOS SYSCALL Example ***
listfile.asm
                 98: * blocks for syscalls
                 99: *
  024C          100: himessage ; syscall block to output "hello" message
024C 0C         101:         fcb     syscall:writea
024D 08         102:         fcb     writea:sclen
024E 00         103:         fcb     channel0
024F 00         104:         fcb     ignored         filler
0250 0254       105:         fdb     hitext          pointer to message
0252 0023       106:         fdb     hitextlen       length of message
                107:
0254 48692128 108: hitext  fcc     'Hi! What file do you want to list?
  0023          109: hitextlen equ   *-hitext        length of message
                110:
  0277          111: inputfilename ; syscall block to accept line from user
0277 0A         112:         fcb     syscall:reada
0278 0E         113:         fcb     reada:sclen
0279 00         114:         fcb     channel0        from the user
027A 01         115:         fcb     linemode        input up to a <cr>
027B 0000       116:         fdb     ignored         dummy write buffer stuff
027D 0000       117:         fdb     ignored
027F 0000       118:         fdb     ignored
0281 02BF       119:         fdb     filenamebuf     read buffer
0283 0100       120:         fdb     filenamebufmax  max amount to read
0285 0002       121:         rmb     2               amount read (set by SDOS
                122:
  0287          123: openfile ; syscall block to open a file
0287 00         124:         fcb     syscall:open
0288 0E         125:         fcb     open:sclen
0289 01         126:         fcb     channel1
028A 00         127:         fcb     ignored         filler
028B 02BF       128:         fdb     filenamebuf     where user's input is
028D 0004       129:         rmb     4               buffer length (set by pg
                130:
  0291          131: readaline ; syscall block to read a line from a file
0291 0A         132:         fcb     syscall:reada
0292 0E         133:         fcb     reada:sclen
0293 01         134:         fcb     channel1
0294 01         135:         fcb     linemode
0295 0000       136:         fdb     ignored         dummy write buffer stuff
0297 0000       137:         fdb     ignored
0299 0000       138:         fdb     ignored
029B 02BF       139:         fdb     readbuffer
029D 0100       140:         fdb     readbuffermax
029F 0002       141:         rmb     2               how much data read
```

```
ASM/6809 1.4A1: 029F
09/17/84 19:43:57; Page 4; Form 1       *** SDOS SYSCALL Example ***
listfile.asm
  02A1         writealine ; syscall block to write a line on terminal
02A1 0C                fcb     syscall:writea
02A2 08                fcb     writea:sclen
02A3 00                fcb     channel0
02A4 00                fcb     ignored         filler
02A5 02BF              fdb     writebuffer
02A7 0002              rmb     2               length of line

  02A9         errorexit ; syscall block to effect error exit
02A9 12                fcb     syscall:errorexit
02AA 04                fcb     errorexit:sclen
02AB 0002              rmb     2               set to error code by pgm

  02AD         byemessage ; syscall block to print "done..."
02AD 0C                fcb     syscall:writea
02AE 08                fcb     writea:sclen
02AF 00                fcb     channel0
02B0 00                fcb     ignored
02B1 02B5              fdb     byetext
02B3 0008              fdb     byetextlen

02B5 646F6E65  byetext fcc     "done..."
02BC 0D                fcb     $0d             carriage return
  0008         byetextlen      equ             *-byetext

  02BD         exit ; syscall block to effect normal exit
02BD 11                fcb     syscall:exit
02BE 02                fcb     exit:sclen

               * and here's the i/o buffer
  02BF         filenamebuf     equ     *
  02BF         readbuffer      equ     *
  02BF         writebuffer     equ     *
  0100         filenamebufmax  equ     $100
  0100         readbuffermax   equ     $100
02BF 0100              rmb     readbuffermax   space for buffer
               *
               * that's all folks!
               *
                       end     listfile
```

WRITING and DEBUGGING User Assembly Programs

Writing a User Assembly-Language program to run under SDOS requires the following steps:

1) Use EDIT (or some other means) to place the desired assembly source program on a disk.

2) Use ASM to produce a listing (optional) and a .BIN (Binary) version of the desired program.

3) a) Execute the program by typing its name
   or
3) b) Debug the program by typing

   .DEBUG name

   This will pass control to the local system debugger (usually IDB) and debugging may commence.

Note: Breakpoints should not be placed on a BCC/BCS after a SYSCALL (SDOS will not see the BCx if an error occurs and a system failure will result). Further, breakpoints should all be removed before a SYSCALL:EXIT or SYSCALL:ERROREXIT is executed. Also, SDOS has no "warm start" entry point; if the program runs away, the operator's only safe choice is to re-boot.

MEMORY MAP

The memory of the 6800/6809 computer, when executing a user program under SDOS, has the following layout:

LOCATIONS	CONTENTS
\$0-\$7 (6800 and 6809) \$18-\$1F (6801 and 6811)	Scratch temporaries, usable by user program. Note: These temporaries are also used by SDOS; so any SYSCALL will destroy their contents.
\$20-\$EF	User program page zero. Not used by SDOS or the I/O package.
\$FO-\$FA	System dependent data used by system hardware (ROM), I/O package or interrupt routines for any purpose; see specific I/O packages. User programs must not disturb this data; references to this data will make the program hardware or configuration dependent.
\$FB,\$FC,\$FD	SYSCALL entry point. These three bytes contain a JMP to the SYSCALL entry point in SDOS. All user programs should define SYSCALL\$ as \$FB; this will make them independent of the actual location of SDOS. These bytes are initialized by SDOS whenever a CHAIN or LOAD SYSCALL is executed. Bytes \$FC,\$FD form a 16 bit pointer to the first byte of SDOS (to the first byte above the memory space available to the user program).
\$FE,\$FF	Reserved for system dependent data (typically a pointer to last byte or page of RAM). User program must not disturb or use.
\$100-(SDOS-1)	User program area. Used in any way desired by user programs. Last byte of this area has an address equal to contents of (\$FC,\$FD) minus 1. On entry (CHAIN) to a user program, the stack register is set to this value (SDOS-1). Generally, user programs have a start address of \$100.
SDOS --	Beginning of SDOS (and/or I/O package). User program may not overlay or store any byte on or above this boundary.

Typical SDOS Address Space

```
                 ---------------------
      LOW        !                   !
                 !    PAGE ZERO      !
                 !___________________!
   $:Ø1ØØ        !                   !
                 !                   !
                 ! SD SOFTWARE or    !
                 ! CUSTOMER SOFTWARE !
                 !                   !
                 !___________________!
                 !         .         !
                 !         .         !
                 !         .         !
                 !
                 !                   !
                 !                   !
      :78ØØ      !___________________!
  TTY,           !                   !\
  LINEPRINTER,   !   MISCELLANEOUS   ! \
  DRIVERS,       !                   !  \
  (ETC.)         !____DISK READ______!   |
                 !    DISK WRITE     !   |
                 !  DISK WAITDONE    !   |  I/O Package
                 !___________________!   |
                 !     DISK          !   |
                 !     SECTOR        !   |
                 !     BUFFERS       !  /
                 !                   ! /
                 !-------------------!
                 !     VT DRIVER     !
                 !___________________!
                 !      SDOS         ! 48K
                 !   (DISK FILE      !
                 !    MANAGEMENT)    ! 63K DEPENDING ON
                 !___________________! CONFIGURATION
                 !///////////////////!
                 !///////////////////! NON-EXISTENT MEMORY
                 !///////////////////!
                 !-------------------!
      :FØØØ      !       IDB         !
                 !___________________!
      :FCØØ      !    SDOS BOOT      !
                 !___________________!
      :FFØØ      !                   !
                 !    I/O SPACE      ! ROM VECTORS TO OTHER ROM
      HIGH       !___________________!
```

SDOS LOADER FORMATS

SDOS will load files containing one of two types of records:

1) SDOS Load Records
2) Encrypted Load Records

A file to be loaded must contain only SDOS load records, or encrypted load records.

SDOS LOAD RECORD FORMATS

SDOS Load Records are designed to let SDOS load large blocks of contiguous memory efficiently, and still retain scatter-load capability. A file containing SDOS Load Records appears as a stream of load records. Each load record has a type and a format. There are four SDOS load record types; all four contain binary information for ease of processing by the loader and to minimize file space occupied. Each load record type is identified by its first byte. One record immediately follows another.

SDOS load record type 1 must be the first record (i.e., start on byte Ø) of the file. It is followed by 2 bytes forming a 16 bit start address, MSB first. The next two bytes are the 16 bit one's complement of the start address, MSB first (this record format makes it extremely improbable that a non-load format file is actually loaded by accident). The first byte of a A Type 1 load record specifies the CPU type:

$Ø1	68ØØ
$Ø3	68Ø1/68Ø3
$Ø2	68Ø9
$Ø7	63Ø3
$11	6811

SDOS load record type Ø is a skip record. The two bytes following the record type byte form a 16 bit count (MSB first) of the number of bytes following the skip record to ignore. The loader processes this record by positioning the file to the file position after the skip record, plus count bytes. This record format is used to align following load records on power of two boundaries which can speed up loading of larger data records.

SDOS load record types 2 and 3 are identical in format. Both record types are used to load blocks of data into the memory address specified by the two bytes following the record type byte (MSB first). The number of bytes to be loaded is given by the 16 bit count specified by the next two bytes (MSB first). The data bytes to be loaded immediately follow the count bytes.

A type 2 record specifies that another load record follows (i.e., that EOF does not immediately follow the records) and that further load record processing is needed. A type 3 record indicates that the load process is complete once the data bytes in the type 3 record are loaded (i.e., there are no more load records in the file). After processing a type 3 record, a SYSCALL:CHAIN will transfer control to the start address specified by the type 1 record.

SDOS LOAD RECORD FORMATS

Command

```
 _____________________________________      _______________
!      Ø !   COUNT     !         !.....!                 !
!________!______!______!_________!     !_________________!
           2 BYTES     !_____________________!
                                 !
                                 !
                            COUNT BYTES
```

Meaning: Skip COUNT bytes to find next command. Used as a space filler to pad to the next physical sector boundary.

```
 __________________________________________________________
!CPU Type !   ADDRESS    ! 16 BIT CHECKSUM !
!_________!______!_______!_________!_______!
                2 BYTES         2 BYTES
```

Set start address. Must be first command in file. CHECKSUM is :FFFF - address.

```
 ____________________________________________________________
!      2 !  ADDRESS    !    COUNT    !    DATA BYTES    !
!________!______!______!______!______!__________________!
            2 BYTES       2 BYTES     !_______________!
                                              !
                                              !
                                         COUNT BYTES
```

Causes data bytes to be loaded sequentially into memory starting with the specified address.

```
 ____________________________________________________________
!      3 !  ADDRESS    !    COUNT    !    DATA BYTES    !
!________!______!______!______!______!__________________!
               2 BYTES        2 BYTES
```

Just like 2, but also causes JUMP to start address specified.

The load records are used in the following way to optimize the disk reads (example):

```
                        ===================
Logical byte #'s       !  Type 1 record   !  Must be first in file
        !              !__________________!
        !              !                  !
        !              !                  !
        !              !  Type Ø record   !  Filler record
        !              !                  !
        !              !__________________!
        !              !  Type 2 record   !  Indicates "Load next
        !              !==================!  two sectors".
      NBPS             !  1ST Data Byte   !
                       !                  !
                       !==================!
     2*NBPS            !                  !
                       !                  !
                       !==================!
     3*NBPS            !  Type 3 record   !  Indicates "Load next
                       !__________________!  two sectors" and
                       !                  !  transfer control to
                       !                  !  start address when
                       !                  !  done.
                       !==================!
     4*NBPS            !                  !
                       !                  !
                       !==================!
```

Encrypted Files

```
 ______________________________________________
|  Ø5   |S# COUNT |        48 Random Bits      |
|_______|_________|____________________________|
|           1st Serial Number                  |
|______________________________________________|
|           2nd Serial Number                  |
|______________________________________________|
|           S#  Serial Number                  |
|______________________________________________|
```

ENCRYPTED OBJECT FILES

An encrypted file is one whose content is not in a directly usable form. Under SDOS, encrypted object files contain proprietary programs which are designed to run on only a limited number of CPUs. Some programs are proprietary to Software Dynamics; other programs are proprietary to other vendors. Software Dynamics provides a tool to allow vendors to encrypt their own object programs or suite of programs.

An encrypted program is decrypted by SDOS while loading into memory by use of an Encryption Key. The Encryption Key is a function of the serial numbers of the CPUs on which the program is authorized run, and a 48 bit "application suite" number embedded in the object file.

SDOS zeros the address space when loading an encrypted file whose Encryption Key is different than the Encryption Key of the last file loaded. This prevents "Trojan Horse" software from obtaining a snapshot of a previously-executed program. Only programs with the same encryption key may pass control (and non-zero data) to one another. This is a common requirement of an "application suite".

Encrypted object files have an un-encrypted 1st object record, followed by the rest of the file in an encrypted format. The encrypted portion of the file, once un-encrypted, is in standard SDOS load record format, with the exception that no skip records are allowed (decrypting skip records is simply a waste of time).

The first object record starts with a byte containing :Ø5, signifying this file is an encrypted object file. The SerialNumberCount (S#) specifies how many serial numbers for which this object file was encrypted. Following the SerialNumberCount are 6 bytes of Application Suite number (typically a random number chosen at time of encryption). Last are a series of 8 byte Serial Numbers on which this object file is authorized to run. These serial numbers are in a clear text form so they can be easily inspected by a utility program.

SDOS DISK FILE STRUCTURE

This section gives detailed information on the structure of the SDOS disk file system. Two concepts are critical to the understanding of the file system: Logical Sector Numbers and Logical Cluster Numbers. These concepts are detailed xbelow.

Definitions:

NBPS	Number of bytes/sector (2^n,n=1..15). Must be power of 2!! NBPS is limited to 128*32=4096 by directory search routine. Minimum size is 128 bytes (see BOOT sector).
NSPT	Number of sectors/track
NTPC	Number of tracks/cylinder
NCYL	Number of cylinders/drive
NLSN	Number of (logical) sectors on a disk (= NSPT*NTPC*NCYL)

Note: Number of bytes/cluster < 2^16 for 6800/6809 implementation.

LOGICAL SECTOR NUMBERS (LSNs)

LSN's are imaginary sequence numbering applied to physical disk sectors on a disk cartridge or floppy diskette. The reason for using them is that Logical Sector Numbers can be mapped onto any disk removing any structure that the disk drive might arbitrarily impose from the knowledge and concern of SDOS; i.e., the distinction between tracks, cylinders, and sectors ceases to be of concern to the SDOS file system.

The only requirements placed by SDOS on LSN's is that they begin with Ø and increase sequentially; further, track Ø, sector Ø, cylinder Ø (usually) maps into LSN Ø. This is because most hardware interfaces can read in this physical disk block as a means for booting the system, so SDOS reserves LSN Ø for this block.

A useful method for choosing the LSN number for a disk block on physical cylinder C, track T, and sector S is:

$$LSN(C,T,S)=S+NSPT*(T+NTPC*C)$$

where NSPT and NTPC are the number of Sectors per Track and the number of Tracks per Cylinder, respectively; where Ø<=S<NSPT, Ø=<T<NTPC, and Ø<=C<NCYL (NCYL= number of cylinders). This has the advantage of allowing SDOS to allocate new blocks to a file by use of their LSN's, attempting to minimize LSN distance (which minimizes Cylinder, Track, and Sector distance, in that order. The name NLSN refers to the number of logical sector numbers for a disk and is equal to NSPT*NTPC*NCYL. There are physical disk read and write routines in the I/O package which are required to convert LSN's into the corresponding values of S, T and C. Each LSN occupies 3 bytes (maximum of $(2^{24})-1$ LSN's).

CLUSTERS (LCNs)

SDOS allocates disk space in units of "clusters" (not sectors!). A cluster is simply a set of sectors whose LSN's are contiguous, and whose lowest LSN is a multiple of the cluster size (an arbitrary constant for a particular diskette or disk pack). Data placed in a cluster is generally related in some fashion.

Each cluster is assigned a logical cluster number (LCN). An (LCN) is the number given to a cluster of sectors. Every LSN is in a cluster whose LCN is given by:

```
LCN(LSN)=INT(LSN/NSPC)
```

where NSPC is the number of disk sectors per cluster (defined for the disk).

The total number of clusters on a disk is given by:

```
NSPC=INT(NLSN/NSPC)
```

The special cluster number :FFFF is reserved, and means "no cluster allocated" or "no such cluster". This is the value to which unallocated clusters specified in cluster headers are set.

The advantage of this clustering technique is that it saves space and time. Space savings are effected on the disk because each file does not need to explicitly record all the sectors it contains. This means less disk space used keeping track of disk space.

Time savings are effected when SDOS is reading sequentially through a file, because (NSPC-1)/NSPC*100% (for NSPC=4, 75%) of the time, SDOS knows the next LSN which is required without having to do any disk reads to collect this information. The disadvantage is a small loss in efficiency of disk storage (i.e., each file wastes NSPC/2 disk sectors on the average, instead of 1/2 disk sector average).

The cluster size is chosen to either minimize average waste of disk sectors in files, or to minimize the seek time between disk sectors in a cluster, subject to several constraints.

The first constraint is that all legal LCN's are limited to the range 0-65534 decimal (65535 is reserved; 2 bytes inside SDOS), i.e., INT((NLSN-1)/NSPC)<65535.

The second constraint is that one cluster should have enough space to contain all the LCN's defined for a disk, i.e., NSPC*NBPS/2 >= INT(NLSN/NSPC) where NBPS is the number of bytes per sector. This constraint allows SDOS to use a single cluster to record all the clusters of a file. This constraint can be violated, but the result is that a single file might not be able to use the entire disk. SDOS will complain if the Header Cluster of a file overflows when allocating space to a file.

The third constraint is that 1<=NSPC<=255. This is purely an implementation restriction and must be followed.

Assuming a file with 2^31=2.1x10^9 bytes, NBPS=512, NSPC=255, we have 2.1x10^9/512=4.2x10^6 sectors in file; 4.2x10^6/255=16449 clusters in file. The header cluster has room for 255*512/2=65280 clusters, which covers such a file easily.

To minimize average wasted space in disk files, NSPC should be chosen to be as small as possible within the constraints specified. This may leave some disk sectors (with high LSNs) unused by SDOS if NLSN is not a multiple of NSPC, but the total wastage here is again only 1/2*NSPC sectors average, and if one has 100 files on a disk, this is insignificant in comparison with the total savings. A final note: if the number of sectors per cylinder is not a multiple of NSPC, some time inefficiency will occur when reading sequentially through a cluster because some clusters will cross track or cylinder boundaries. This inefficiency will be small if the average file size is much greater than NSPT*NSPC.

If the average file size is smaller than NSPT*NTPC, some time savings can be gained by making NSPC a divisor of NTPC - this will generally prevent part of file (cluster) from overlapping cylinder boundaries, and will therefore save seek time.

A sample calculation of NSPC:

> Assume we have 77 cylinders (NCYL=77), 1 track/cylinder (NTPC=1), 16 sectors/track (NSPT=16), 256 bytes/sector (NBPS=256) (so NLSN=NSPT*NTPC*NCYL=16*1*76=1232). Let LSN(C,T,S)=S+16*(T+1*C). Since we have only one track (track #0), the formula simplifies:
>
> LSN(C,S)=S+16C
>
> For any NSPC>=1 then NLSN/NSPC < 65536, satisfying constraint 1.
> Constraint 2 implies:
> NSPC*256/2>=INT(1232/NSPC)
> NSPC*128>=INT(1232/NSPC)
> which is true for any NSPC>=4
> If we choose NSPC=4, constraint 3 is also satisfied.

To minimize average wasted space, we choose NSPC=4. On a disk with 100 files, an average of 100*4/2=200 disk sectors are wasted. With NSPC=3, with 100 files wastes an average of 100*3/2=150 sectors, and prevents files from containing more than 1152 sectors (i.e., a particular file can only cover 93% of the disk).

DISK FILE STRUCTURE

A File under SDOS is a mechanism for storing logically related information. From the point of view of an application program, a disk file is a very large array of 8 bit bytes which can be read/written sequentially, can be positioned for later read/writes, and can be automatically extended (at the end) to add more information. These files can be up to 2^31 bytes (2.1 billion bytes) in size, physical disk size being the real limitation.

This view of files is implemented by device drivers. The operations that a device driver considers legal and the actual operation performed are dependent on each device driver (see Device Drivers). There are two kinds of drivers: non-disk file and disk file.

The disk file driver is a component of SDOS proper. It handles files by breaking them down into two layers: clusters and sectors. Sectors are the physical unit of transfer to/from the disk drive. Clusters are a logical grouping of sectors used to minimize the amount of information required to record where all the sectors of a file are located.

Each file has a special cluster of sectors known as the Header Cluster. The Header Cluster contains the logical cluster numbers of all (data) clusters contained in the file. These numbers are placed in the Header Cluster in such a way as to indicate the relative (byte) position of the target cluster in the file.

A special cluster number of hex :FFFF means "no data cluster allocated" to this place in the file. This allows sparse files to be built with very little wasted space.

```
                           __________
                           !        !  First Cluster Pointer
                           !        !  Points to Headercluster
                           !        !
                           V        !
Header      -       ______________  !
Cluster     !   2B  !     ----------!
            !   2B  !     ------------------
            !   2B  !     ---------------  !  Remaining Cluster Pointers
            !       !     .      !      !  !  Point to Data Clusters
            !       !     .      !      !  !
            !       !____________!      !  !
            !       !      ----------   !  !
            !       !            !  !   !  !   Data Cluster #Ø
    NSPC*NBPS       !            !  !   !  !   ___________
            !       !____________!  !   !  !-->!          !
            !       !            !  !   !      !          !
            !       !            !  !   !      !__________!
            !       !            !  !   !
            !       !            !  !   !
            -       !____________!  !   !      Data Cluster #1
                                    !   !      ___________
                                    !   !----->!    Ø     !
                                    !          !__________!
                                    !          !    1     !
                                    !          !__________!
                                    !          !    .     !
                                    !          !    .     !
                                    !          !    .     !
                                    !          !__________!
                                    !          !  NSPC-1  !
                                    !          !__________!
                                    !
                                    !
                                    !          Data Cluster #N
                                    !          ___________
                                    !--------->!          !
                                               !          !
                                               !__________!
```

The first two bytes in the header cluster are reserved to contain the cluster number of the header cluster itself (this simplifies the space allocation routines). Succeeding pairs of bytes contain the logical cluster numbers of the Øth data clusters, 1st data cluster, etc.

When a file is first allocated, all the pointers (except the first) in the first sector of the header cluster are initialized as :FFFF (no data cluster allocated). The other sectors in the header cluster are left as garbage.

A special 1 byte counter (stored in the directory), DIR:HCIC (header cluster initialized count) tells SDOS how many of the sectors in the header cluster are initialized.

If a data byte is in logical byte number "LBN" in a file, then SDOS can access that byte (in at most two disk reads) by the following process (definition):

First, compute:

```
NBPC := NSPC*NBPS         (COMPUTE # BYTES/CLUSTER)
RDCN := INT(LBN/NBPC)     (COMPUTE THE RELATIVE DATA
                          CLUSTER NUMBER)
HSLSN := INT[(RDCN+1)*2]/NBPS + HCLCN*NSPC
```

where HSLSN = header sector logical sector number
and HCLCN = header cluster logical cluster number.

This computes the LSN of desired sector in the Header Cluster. The "+1" is because the first cluster pointer is the pointer to the header cluster. The "*2" is because each cluster number occupies two bytes.

Note: HCSIC may indicate that this sector (HSLSN) has not been initialized!!

Next:

read HSLSN into memory in HBBUF (header buffer)
DBLCN := @((((RDCN+1)*2)MOD NBPS)+.HBBUF)
this computes the LCN of the data cluster containing the byte.

"@" means use the value to the right as a memory address and fetch 16 bits. ".HBBUF" means the address of the header buffer. Note: DBLCN may be :FFFF (undefined)!!

Finally:

DBLSN := DBLCN*NSPC+INT((LBN MOD NSPC*NBPS)/NBPS)
Read DBLSN into memory; desired byte is found at displacement RBN := LBN MOD NBPS

SDOS FILE STRUCTURE

```
              !       .       !
              !       .       !
              !       .       !
              !_______________!
              !  DIRECTORY    !
              !    ENTRY      !
              !               !
              !    LCN -----------
              !_______________!    !
              !      .        !    !
              !      .        !    !
              !      .        !    !
                                  /
                                 /
                                /
                               /
                -------------/
                !
                !
                !
     ___        !
     ! !        !  Header Cluster - First Cluster of File
     ! _V____V_   _______   _______       _______
     !-----   !  !      !  !      !      !      !  )(Some)
        !_____!  !      !  !      !      !      !  )Sections
  !-------    !  !      !  !      !      !      !  )Initialized
  !     !_____!  !      !  !      !      !      !  )To "No
  !  !-----   !  !      !  !      !......!      !  )Cluster
  !  !  !_____!  !      !  !      !      !      !  )Allocated"
  !  !  ! 3RD !  !      !  !      !      !      !  )At File
  !  !  !_____!  !      !  !      !      !      !  )Creation
  !  !  !  .  !  !      !  !      !      !      !  )Time
  !  !  !  .  !  !      !  !      !      !      !
  !  !  !_____!  !______!  !______!      !______!
  !  !
  !  !-------> 2nd Data Cluster
  !                                 ______________________
  !---------> 1st Data Cluster     ! A Cluster Number Of !
                                   !        :FFFF        !
                                   !  Means "No Cluster  !
                                   !     Allocated"      !
                                   !_____________________!
```

```
     __________
     !        !
     !        V   Header Cluster
     !
     !    __________
     !--------     !
     ---------     !
     !    !        !
     !    !    ------------> 2nd Data Cluster
     !    !    ------------> Etc.
     !    !________!
     !
     !
     !   1st Data Cluster
     !
     !    -------  -------  -------     -------
     !    !     !  !     !  !     !     !     !
     !----->    !  !     !  !     !     !     !
          !     !  !     !  !     !     !     !
          !     !  !     !  !     !.....!     !
          !     !  !     !  !     !     !     !
          !     !  !     !  !     !     !     !
          !     !  !     !  !     !     !     !
          !     !  !     !  !     !     !     !
          -------  -------  -------     -------
```

Logical sector number of 1st sector in cluster =
cluster number * cluster size in sectors.
Succeeding sector numbers are base sector number + index into cluster.

DIRECTORY.SYS STRUCTURE

The directory stores the name and location of the header cluster for all files on the disk (SDOS allows no magic disk files which are not in the directory; even system files are mentioned in the directory).

Each Directory entry is 32 bytes and contains the following information:

DIR:FILENAME

The file name can be any left-justified sequence of letters (uppercase only), digits Ø through 9, $ or ".". It may not begin with a "." or a digit. The name is blank filled to 16 bytes. Two file names are considered equivalent if they match byte for byte. SDOS automatically folds lowercase characters in disk file names into uppercase. Bit 7 of all bytes must be zero.

DIR:HLCN

The Header Logical Cluster Number specifies the location of the Header Cluster for the file if DIR:HCSIC > Ø. If DIR:HCSIC = Ø, then DIR:HLCN is actually the cluster number of the 1st data cluster.

DIR:HCSIC

The Header Cluster Sector Initialized Count tells SDOS how many sectors of the header cluster have been initialized properly (see File Structure) and need not concern any but the systems programmer. DIR:HCSIC is updated whenever a new header cluster sector is initialized. If DIR:HCSIC is zero, and DIR:NCLUSTERS > Ø, then the file is contiguously allocated on the disk, with the first data cluster being in DIR:HLCN, contiguous for DIR:NCLUSTERS.

DIR:NCLUSTERS

DIR:NCLUSTERS is the number of clusters allocated. This count is needed as a very sparse file may have an enormous logical file size, and yet have a very small actual disk allocation. SDOS updates DIR:NCLUSTERS only when a file is closed. If DIR:NCLUSTERS is zero, this directory entry is not valid and is available for use by a new file (name).

DIR:FILESIZE

DIR:FILESIZE is the apparent size of the file in bytes, and is equal to the position of the last data byte written in the file, +1. The filesize is completely independent of sector or cluster size.

DIR:PROTECTION

DIR:PROTECTION contains file protection bits. The protection bits prevent inadvertant or undesired references to file. The currently defined bits are:

7	<NOT DEFINED>
6	PROTECT:WRITE
5	<NOT DEFINED>
4	<NOT DEFINED>
3	<NOT DEFINED>
2	<NOT DEFINED>
1	<NOT DEFINED>
Ø	PROTECT:BACKUP

PROTECT:WRITE

The PROTECT:WRITE bit prevents DELETE, RENAME, and CREATE commands on a file with the corresponding name. This is used by SDOS to prevent accidental erasures of critical system files, and may be used by the user to protect his critical files.

PROTECT:BACKUP

This bit prevents SDOSDISKBACKUP from backing up a file if the CHANGED option is specified. It is reset whenever a file is first created, or when a file is modified in any way (SYSCALL:WRITEA, SYSCALL:WRITEB, CC:SETFILESIZE, etc.). It is set by SDOSDISKBACKUP whenever a file has been backed up using the CHANGED option.

DIR:CREATIONDATE

DIR:CREATIONDATE contains the creation date of the file in the form DDMMYY. DD is one byte containing the day number in BCD; MM is one byte of BCD month; and YY is the year number modulo 1ØØ in one BCD byte.

SDOSDISKINIT generally places the first data cluster of the directory at INT(NLCN/2) (the middle of the logical disk) in an attempt to decrease seek-to-directory time. This also causes SDOS to extend the directory in the middle of the disk if need be. Note: This LCN must be non-zero! (Otherwise, the directory and the boot cluster collide.)

SDOS locates the directory initially by reading BOOT:DIRLSN from the BOOT.SYS file. BOOT:DIRLSN gives the LSN of the directory sector containing the DIRECTORY.SYS entry. The directory entry for DIRECTORY.SYS is located in the first 32 bytes of the sector. This requirement also forces the sector size to be at least 32 bytes (the first entry must be contained entirely in the first directory sector), and to be a multiple of 32!

All other filenames in the directory are added to it according to the following procedure:

> The directory is searched by initially hashing the desired name to choose a directory sector, and then searching circularly through the directory for the desired name. The hashing scheme tends to make lookups of existing names very quick, as long as the directory is 80% or less loaded. The circular search guarantees that even if the directory size changes, files will still be found.
>
> The directory is automatically expanded by SDOS if it is full and a new filename needs to be added. This automatic expansion invalidates all the previous hashes, but since new (or renamed) files will get hashed to the correct place, after the system has been used with the expanded directory awhile, lookups will speed up again. Renaming a file rehashes it, so renaming all files will rehash them all.

The directory size is kept in the DIR:FILESIZE entry of the DIRECTORY.SYS entry, and is always a multiple of the cluster size (NBPC).

As a convenience to the hashing algorithm, a limit of 65536 directory sectors is imposed on the DIRECTORY.SYS file.

```
                Directory Entry
                       .
                       .
                       .
               !       .        !
               !       .        !
               !       .        !
               !       .        !
               !       .        !
               !================!
               !                ! 16 Bytes,
               !    DIR:NAME    ! Legal Names Contain
               !                ! A-Z, Ø-9, $, "."
               !----------------!
               !    DIR:HLCN    ! 2B, Header Cluster Number
               !                !
               !                !
               !----------------!
               !   DIR:HCSIC    ! 1B Header Cluster Sector
               !                ! Initialized Count
               !----------------!
               !  DIR:NCLUSTERS ! 2B Number of Data
               !                ! Clusters in File
               !----------------!
               !  DIR:FILESIZE  ! 4B, Number of Bytes in
               !                ! The File
               !----------------!
               ! DIR:PROTECTION ! 1B, Protection Bits
               !                !
               !----------------!
               !DIR:CREATIONDATE! 3B
               !----------------!
               !     UNUSED     ! 1B
               !================!
               !       .        !
               !       .        !
               !       .        !
               !                !
               !                !
               !                !
               !                !
```

File System Sketch

```
                   DIRECTORY.SYS
                     file body
                   _____________
                   !           !<-------|
LCN Ø, LSN Ø       !    ...    !        |  DIRECTORY.SYS
                   !___________!        |  Header Cluster
____________       !           !     ___|____________________
! BOOT:    !       ! DIRECTORY !     !  |  !    !    !    !
! DIRLSN *-------> ! .SYS   #--------->!  #  !    ! ...!    !
!_________!        !___________!     !_____!____!____!____!
  ^                !           !      usually, LCN=@(Directory)-1
  |                !    ...    !
  |   BOOT.SYS     !___________!
  |   Header       !           !
__|__________      !           !
! # ! ... !  !<-----# BOOT     !
!___!_____!__!     !  .SYS     !
                   !___________!        DISKMAP.SYS file body
                   !           !        "Busy Cluster Map"
                   !    ...    !          (1 cluster)
SDOS.SYS           !___________!
Header Cluster     !           !     ______________________
---------------    ! SDOS.SYS  !     !     !    !    !    !
! # ! ... !  !<--------#       !     !     !    ! ...!    !
!_|_!_____!__!     !___________!     !_____!____!____!____!
  |                !           !        ^
  | SDOS.SYS       !    ...    !        |
  | file body      !___________!        |   DISKMAP.SYS
  V                !           !        |   Header Cluster
___________        !           !     ___|__________________
! LCN #1  !        ! DISKMAP   !     !  |  !    !    !    !
! LCN #2  !        ! .SYS #--------->!  #  !    ! ...!    !
! LCN #3  !        !___________!     !_____!____!____!____!
!   ...   !        !           !
! LCN #n  !        !    ...    !
!_________!        !___________!
```

Notes: * represents pointer to (logical) sector
represents pointer to (logical) cluster

SDOS hashes to 1st directory entry, and does linear circular search thereafter.

Physical Placement of Files on Disk

```
       LCN ~~ NLCN/2
              |     Logical Sector Ø
              |             |
              |             V
      _______|___________________________
     /        |         !  BOOT.SYS     /\
    /         |         !              /  \
   /          |          _____________/ S  \ <--- Logical Cluster #1
  /           |                       \  D  \ <--- LCN 2, 3, etc.
 /          __V______________          \  O  \
/          ! DISKMAP.SYS !              \  S  \
           ---------------               !  .  !
!                                        !  S  !
!                /-------\               !  Y  !
!                !       !       ___     !  S  !
!                !       !      ! S !    !_____!
!                \_______/      ! Y !          !
!                               ! S !          !
 \                             / . /  <---------- LCN ~~ NLCN/2
  \          ______________   / Y /           /
   \        ! D I R E C T O R    /           /
    \       -----------------/             /
     \                                    /
      \__________________________________/
```

THE BOOT.SYS FILE

BOOT.SYS is a file which owns LSN 0 (the boot sector).

The BOOT.SYS file contains three things:

1) a disk identification (32 bytes of text blank padded).
2) the appropriate DISKINFO (tuning parameters) for this disk
3) a "simple" program to read SDOS off the disk and into memory as a means of booting.

Items 1 and 2 are stored in fixed places in LSN 0 and occupy the first 64 bytes. This sets a minimum sector size requirement of 64 + 1 --> 128 (sector sizes must be a power of two!). Other LSN's in the BOOT.SYS file are simply wasted or used to store an extended bootstrap program if needed.

The form of the boot sector must be as follows:

```
           ________________
          !                !
16 BYTES  !     JUMP  ----------- These Bytes Are
          !________________!     ! Used For Any
          !                !     ! Purpose By Boot
15 BYTES  !     DISK       !     ! or Boot ROM
          !     INFO       !     ! Routine
          !________________!     !
 1 BYTE   !DISKINFOCKSUM   ! <------ = :FF - Sum of
          !________________!     !      Bytes in
          !                !     !      DISKINFO Part
          !                !     !
          !                !     !
32 BYTES  !     DISK       !     !
          !      ID        !     !
          !                !     !
          !                !     !
          !________________!     !
          !                !     !
          !             <---------
          !    BOOT        !
REST OF   !   ROUTINE      !
SECTOR    !                !
          !                !
          !________________!
```

This ensures that the disk identification string is easily locatable, and that it does not prevent the boot routine from executing.

Normally, LSN 0 is read into memory at $100 by a ROM boot routine, and control is passed to location $100. The boot sector reads in the rest of BOOT.SYS if necessary.

The boot routine then reads the contents of the SDOS file into memory at the appropriate place, and transfer control to the starting point.

BOOT:FILESYSTEMVERSION is a single byte containing a file system version and revision number in the left and right nibbles, respectively. This document describes file system version 1.1 (note: SDOS revision numbers do not necessarily match file system revision numbers!).

BOOT:NSPC is a single byte which specifies the cluster size of clusters on this disk (Ø<BOOT:NSPC<=255).

BOOT:MINALLOC is two bytes which specify the minimum number of data clusters to allocate to a disk file when it is created on this disk. Ø is not legal.

BOOT:MIDALLOC is two bytes which specify the minimum number of clusters to be allocated to a file being extended. BOOT:MIDALLOC must be >= 1.

BOOT:MAPALGORITHM is 16 bits which are used in a disk-sector driver dependent way to tune rotational and seek latency times to a minimum.

Commonly, the upper 8 bits are used as "spiralling", or the number of sectors each cylinder should be offset from the next (cylinder) to tune seeks for sequential reads; the lower byte tunes the physical spacing between adjacent logical sector numbers (also measured in units of sector times). SDOS can usually only read every other sector, best case.

When using the "common" mapalgorithm interpretation to map LSNs into physical CYLINDER, TRACK, and SECTOR (assuming CYLINDERs and TRACKs increase sequentially from Ø, and physical sector Ø on all TRACKs are aligned) the following formulas apply:

```
REM PSUEDO-BASIC TO COMPUTE PHYSICAL CYLINDER, TRACK, SECTOR
CYLINDER= INT(LSN/(NSPT*NTPC))
TRACK= INT((LSN-CYLINDER*NSPT*NTPC)/NSPT)
SECTOR= ((CYLINDER*SPIRAL)+MAP[LSN MOD NSPT]) MOD NSPT
```

where MAP[i]= (i*SPACING) MOD NSPT if SPACING is relatively prime to NSPT, and is generally computed as:

```
    MAP[Ø]:= Ø \ !RULE!
    K= SPACING
    FOR i= 1 TO NSPT-1
100 FOR J= Ø TO i-1
    IF K= MAP[J] THEN K=(K+1) MOD NSPT\ GOTO 1ØØ
    NEXT J
    MAP[i]= K
    K= (K+SPACING) MOD NSPT
    NEXT i
```

On hardware systems where formatting a disk is used to effect this tuning, the Mapalgorithm is by convention always set to "1".

BOOT:CREATIONDATE is the date that this disk was SDOSDISKINITed, and consists of 3 BCD bytes: day, month, and year MOD 1ØØ, respectively.

BOOT:DIRLSN is the Logical Sector Number of the DIRECTORY.SYS sector that contains the DIRECTORY.SYS directory entry in the first 32 bytes.

BOOT:CHECKSUM contains :FF-[sum of the 15 bytes between (and including) BOOT:FILESYSTEMVERSION] modulo 256, and is used to check the validity of the disk.

BOOT:DISKID contains 32 ASCII characters blank filled, used solely as a disk identification. This ID is displayed by the FILES command. It can be used (read) by an application for the purpose of verifying the disk before the application uses it.

The boot routine is used to read the contents of SDOS.SYS into memory. Ususally, the boot routine does not fit entirely into the remainder of the BOOT sector; the rest of the boot routine is stored in memory image format in the remaining sectors of LCN Ø. Listings of sample boot routines can be obtained from the distributor of SDOS or from Software Dynamics.

SERIALNUMBER.SYS

SERIALNUMBER.SYS is a file required to be on an SDOS boot disk in order that SDOS may successfully boot. The file is encrypted, and contains several things:

A first-time-only conversation with the purchaser of SDOS;

The serial number of the computer for which the particular version of SDOS was sold; and

The name of the end-user, or organization.

SDOS, after initializing operation of the system, goes and hunts for SERIALNUMBER.SYS. If this file does not exist, SDOS displays, and hangs up with a "No SERIALNUMBER.SYS file" error. If the file does exist, it is CHAINed to, causing implicit decryption. The SDOS decrypting loader will refuse to load SERIALNUMBER.SYS if the serial number encoded into it does not match that of the ROM included in the system hardware; this causes a "Wrong Serial Number" message to be printed, and operation of SDOS is aborted. If SERIALNUMBER.SYS is not encrypted, an error message will likewise be generated and SDOS will not run. Otherwise, the module is loaded and executed. If this is not the first time SERIALNUMBER.SYS has been loaded, then SERIALNUMBER.SYS first prints the ROM serial number, and the name of the end-user; further operation of SDOS is normal.

The name of the end-user is supplied by the end-user when the copy is first run by him; i.e., if the end-user name is blank. SERIALNUMBER.SYS asks the name and then waits for the operator to enter a corresponding code number that he must obtain from Software Dynamics. This code number is generated by Software Dynamics from the serial number of the computer and the string entered by the user (this may be obtained from Software Dynamics well in advance of system installation, in order to minimize delays). An invalid response is so indicated, and the end-user name is NOT updated. A correct response causes SERIALNUMBER.SYS to change the end-user name to the supplied string. Once set, the SERIALNUMBER.SYS file can never be changed again. A response of <CR> is taken as a signal that the user does not wish to set the name yet (this may be a demo copy, or the user may not have yet obtained the corresponding code number from Software Dynamics); in this case, after a 30 minute delay, SDOS will operate normally.

SDOS.SYS

SDOS.SYS is an SDOS load record format file containing the memory resident part of the operating system. It is used by the boot procedure to load a copy of the system from the disk into memory.

To simplify the boot process, certain restrictions are made on the file structure of SDOS.SYS.

The data LCNs of SDOS must be numbered 1, 2, 3,... etc., i.e., a contiguously allocated file. This guarantees sequentially increasing LSNs which makes the boot routine's job (of computing LSNs) extremely simple. The header LCN of SDOS (if it has one) may be anywhere on the disk; the boot routine need not look at it (many boot routines never bother reading the SDOS header clusters). Normally, the SDOSDISKINIT program assigns a very high LCN to the header cluster of SDOS.SYS.

The start address of SDOS.SYS is defined to be SYSCALL$ (:FB).

When debugging a (newly SYSGENed) I/O package, a convenient trick is to modify (using BMP, the Binary Maintenance Program) the first load record (actually the start record) in the SDOS.SYS file so the SDOS start address is the entry point to the ROM debugger instead of :FB. With this change made, "booting" will cause SDOS to get loaded, and the debugger will then gain control. Patches may be made and breakpoints established, and then SDOS can be started by causing a jump to :FB. When debugging is completed, the first load record should be restored to its initial state.

DISKMAP.SYS

The DISKMAP.SYS file is used to keep track of clusters allocated to disk files for that disk. Each disk cartridge or floppy diskette has its own DISKMAP.SYS.

The file has one bit per cluster on the disk on which the file resides. An "on" bit indicates the cluster is allocated (or contains a bad sector). An "off" bit indicates a free cluster, available for allocation to a file. SDOS assumes that the entire disk map can be contained in a single cluster, so if constraint 2 of LCNs is violated (see section on CLUSTERS), one needs to make sure that NBPS*8*NSPC>=INT(NLSN/NSPC) (otherwise the diskmap doesn't fit into a single cluster). If constraint 2 is satisfied, so is this condition (the 8 is the number of bits per byte).

Each byte of DISKMAP.SYS represents 8 clusters, such that bit number n (starting with Ø, counting from the right) represents an LCN such that (LCN mod 8) = n. Bytes at logically higher byte addresses within DISKMAP.SYS represent groups of LCNs with higher values, so that if LBN (logical byte number), BITN (bit number) represent a particular bit in the DISKMAP.SYS, then that bit corresponds to LCN=LBN*8+BITN (logical cluster number).

Cluster space allocation is done by taking the previously allocated cluster number (to a file) as the starting point of a search for a free cluster. Searches toward logical cluster number Ø and NLCN-1 are both made in an attempt to minimize the distance between the cluster number (allocated to the preceding cluster in the file) and the new. Furthermore, an old cluster number of :FFFF causes allocation starting at a random place within the map.

The allocator prefers a forward search, and will not bother with a backwards search if it can get a distance of 1 between the old cluster number and the new.

DISK CLUSTER ALLOCATION MAP

```
    MSB represents Logical Cluster #7
                    |
                    |    LSB corresponds to Logical Cluster #Ø
                    |                    !
                  __V__________________V__
        BYTE #Ø   !                      !<--Logical Clusters #Ø-7
             #1   !                      !<--#8-15
             #2   !                      !<--#16-23
            .     !                      !<--#24-31
            .     !                      !
            .     !                      !
            .     !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
                  !                      !
           ^      !                      !
           |      !                      !
           |      !                      !
     One Cluster  !           /-------------- Bit # (NLCN-1) mod 8
           |      !___________V__        !    (max legal cluster number)
           |      !//////////!   !       !
           v      !//////////!___!_______!
                  !//////////////////////!
                  !//////////////////////!
                  !//////////////////////!
   NOTE: Sector   !//////////////////////!
   Containing Bit !//////////////////////!
   For NLCN-1 is  !///////ILLEGAL////////!
   Null Filled With !/////CLUSTER////////!
   1's to Make it !//////NUMBERS/////////!
   Appear that the !/////////////////////!
   Illegal LCNs   !//////////////////////!
   are allocated. !//////////////////////!
                  !//////////////////////!
                  !----------------------!
```

A "1" bit --> cluster is busy
A "Ø" bit --> cluster is free (available).

ERRORMSGS.SYS (ERROR MESSAGE FILE) FORMAT:

The ERRORMSGS.SYS file is used by SDOS to convert 16 bit error codes into English text messages for the operator.

The file must exist on the default disk, and the default disk must be mounted, in order for SDOS to use the file (otherwise SDOS merely prints "Error nnnnn").

The file is organized into two parts, and is sparse.

The first part of the file converts 16 bit error codes into string pointers into the file. The second part of the file contains the raw error message text.

The 16 bit error code is multiplied by 3, and used as a byte index on the file to fetch a 3 byte relative index into the file. The 3 byte index points to an ASCII error message string, ending with a CR (:ØD) character. The SDOS error routines do not print the CR explicitly but use it only to decide where the end of the error message is (see SYSCALL:DISPERROR). A 3 byte index value of zero means "no message defined".

The first 65536*3=1966Ø8 bytes of the file are reserved for this lookup; since the number of error messages actually defined out of the 65536 possible is very small, this region of the file is sparsely allocated. The first text message starts in byte number 65536*3 of the file. This section of the file is dense.

New messages are added to the file by merely appending them to the end, and adjusting the 3 byte pointer corresponding to the error code to point to the old end of file.

The program SDOSERRORMAINT is used to maintain the contents of this file. The file ERRORMSGBUILD.DO is a DO file used to initially construct this file.

ERROR MESSAGE NUMBER ASSIGNMENTS:

Ø	No Error
1	Operator Requested Attention
2-99	BASIC Compiler Runtime Errors
1ØØ-199	Errors Related to System Processors, Etc.
2ØØ-299	EDITor Errors
3ØØ-999	Application System Dependent Errors
1ØØØ-1999	SDOS / I/O Errors
2ØØØ-65535	Reserved

BUILDING A TURN-KEY APPLICATION SYSTEM

In many circumstances, the full generality of an SDOS development system is not needed; a simple menu-driven application program selector plus the applications is sufficient. This is useful in an office environment because it reduces the training required of the office personnel.

Only two things need be done to build a turn-key application system:

1) The boot process needs to be made automatic. This procedure is hardware dependent and is not described further here.

2) The DEFAULTPROGRAM on an otherwise bootable SDOS disk needs to be replaced by the menu-display program. This program may contain the entire application, or it may CHAIN to other segments at the appropriate time. The other segments, on completion, will EXIT, which causes DEFAULTPROGRAM (the menu-display program) to be reloaded, and the cycle repeats.

Note that the application program must set the time and date itself by doing a WRITEB to the CLOCK$ device.

System development can still continue on a turn-key system if the menu program has a way of chaining to SDOSCOMMANDS, or if a regular development disk is inserted (just the boot part is automatic).

If DEFAULTPROGRAM is replaced by a compiled BASIC 1.4 program, the 1.4 program must be combined with a runtime package.

```
          *       SDOS 1.1 DEFINITIONS FILE (AS OF 8/16/82)
          *
 0011     SDOSVERSION     EQU     $11      1.1 IN HEX

 0000             IFUND   LISTDEFS
                  FIN

 0001             IF      LISTDEFS
 0001             ELSE
                  FIN

          *
          *
          *       The Definitions are broken into 3 parts:
          *               A) THOSE NEEDED TO ASSEMBLE SDOS PROPER OR SYSTEM PROGRAMS
          *               B) THOSE NEEDED TO BUILD AN I/O PACKAGE (A SUBSET OF "A")
          *               C) THOSE NEEDED BY EVERYDAY USER PROGRAMS (A SUBSET OF "B")
          *

 0001             IFUND   SYSTEMDEFS
 0000     SYSTEMDEFS      EQU     0        DON'T WANT SYSTEM DEFINITIONS
                  FIN

 0001             IFUND   IOPKDEFS
 0000     IOPKDEFS        EQU     0        DON'T WANT I/O PACKAGE DEFINITIONS
                  FIN

 00FB     SYSCALL$        EQU     $FB      JMP TO SDOS; RESERVED SYSCALL ENTRY POINT
          *       CONTENTS OF ($FC,$FD) POINTS TO END OF USER RAM
          *       CONTENTS OF ($FE,$FF) ARE SACRED; THEY BELONG TO THE ROM
```

```
                        36: *       SYSCALL$ OPCODE DEFINITIONS
                        37: *
 ØØØØ                   38:         ORG     Ø
ØØØØ ØØØ1               39: SYSCALL:OPEN    RMB     1       OPEN FILE
ØØØ1 ØØØ1               40: SYSCALL:CREATE  RMB     1       CREATE A NEW FILE
ØØØ2 ØØØ1               41: SYSCALL:CLOSE   RMB     1       CLOSE A FILE
ØØØ3 ØØØ1               42: SYSCALL:RENAME  RMB     1       RENAME A FILE
ØØØ4 ØØØ1               43: SYSCALL:DELETE  RMB     1       DELETE A FILE
ØØØ5 ØØØ1               44: SYSCALL:LOAD    RMB     1       LOAD AN OVERLAY
ØØØ6 ØØØ1               45: SYSCALL:CHAIN   RMB     1       CHAIN TO A FILE
ØØØ7 ØØØ1               46: SYSCALL:CREATELOG       RMB     1       CREATE THE LOG FILE
ØØØ8 ØØØ1               47: SYSCALL:CLOSELOG        RMB     1       CLOSE THE LOG FILE
ØØØ9 ØØØ1               48: SYSCALL:DISKDEFAULT     RMB     1       SELECT DEFAULT DISK DEVICE
ØØØA ØØØ1               49: SYSCALL:READA   RMB     1       READ ASCII BYTES FROM A FILE
ØØØB ØØØ1               5Ø: SYSCALL:READB   RMB     1       READ BINARY BYTES FROM A FILE
ØØØC ØØØ1               51: SYSCALL:WRITEA  RMB     1       WRITE ASCII BYTES TO A FILE
ØØØD ØØØ1               52: SYSCALL:WRITEB  RMB     1       WRITE BINARY BYTES TO A FILE
ØØØE ØØØ1               53: SYSCALL:CONTROL RMB     1       PERFORM A CONTROL OPERATION ON A FILE/DEVICE
ØØØF ØØØ1               54: SYSCALL:STATUS  RMB     1       READ FILE/DEVICE STATUS
ØØ1Ø ØØØ1               55: SYSCALL:WAITDONE        RMB     1       WAIT FOR I/O ON CHANNEL TO COMPLETE
ØØ11 ØØØ1               56: SYSCALL:EXIT    RMB     1       GIVE CONTROL BACK TO THE OPERATING SYSTEM
ØØ12 ØØØ1               57: SYSCALL:ERROREXIT       RMB     1       EXIT TO SYSTEM WITH ERROR CODE
ØØ13 ØØØ1               58: SYSCALL:SETERROR        RMB     1       REPORT AN ERROR TO THE SYSTEM
ØØ14 ØØØ1               59: SYSCALL:GETERROR        RMB     1       READ BACK THE LAST ERROR CODE
ØØ15 ØØØ1               6Ø: SYSCALL:DISPERROR       RMB     1       DISPLAY ERROR MESSAGE CORRESPONDING TO LAS
ØØ16 ØØØ1               61: SYSCALL:KILLPROOF       RMB     1       PREVENT USER PROGRAM FROM BEING KILLED
ØØ17 ØØØ1               62: SYSCALL:KILLENABLE      RMB     1       ALLOW USER PROGRAM TO BE KILLED
ØØ18 ØØØ1               63: SYSCALL:DEBUG   RMB     1       CALL SYSTEM DEBUGGER
ØØ19 ØØØ1               64: SYSCALL:ATTNCHECK       RMB     1       OPERATOR ATTENTION REQUEST CHECK
ØØ1A ØØØ1               65: SYSCALL:ISCONSOLE       RMB     1       CHECK FOR CHANNEL Ø INPUT DEVICE = CONSOLE
ØØ1B ØØØ1               66: SYSCALL:INTERLOCK       RMB     1       PERFORM INTERLOCK FUNCTIONS ON OBJECTS
ØØ1C ØØØ1               67: SYSCALL:DELAY   RMB     1       DELAY FOR n 1/6Øths OF A SECOND
ØØ1D ØØØ1               68: SYSCALL:READLUN RMB     1       CONVERT LOGICAL UNIT NUMBER TO DEVICE NAME
ØØ1E ØØØ1               69: SYSCALL:GETSERIALNUMBER RMB     1       GET PROCESSOR SERIAL NUMBER
ØØ1F ØØØ1               7Ø: SYSCALL:JOBCONTROL      RMB     1       CREATE/TEST/DESTROY OTHER JOBS
```

```
                  *
                  *        SYSCALL BLOCK DISPLACEMENTS
                  *
 ØØØØ                      ORG      Ø
ØØØØ ØØØ1         SCBLK:OPCODE     RMB      1        PRIMARY SYSCALL FUNCTION (OPEN, READ, ETC.)
ØØØ1 ØØØ1         SCBLK:WLEN       RMB      1        WAIT FLAG BIT (Ø=WAIT) AND SYSCALL BLOCK LENGTH (Ø..
ØØØ2 ØØØ2         SCBLK:PARAMS     RMB      2        PARAMETER BYTES TO OPCODE (SECONDARY OPCODE, CHANNEL
ØØØ4 ØØØ2         SCBLK:WRBUF      RMB      2        POINTER TO WRITE DATA BUFFER
ØØØ6 ØØØ2         SCBLK:WRLEN      RMB      2        NUMBER OF BYTES IN WRITE DATA BUFFER
ØØØ8 ØØØ2         SCBLK:RPLEN      RMB      2        LENGTH OF REPLY (RESULT OF SYSCALL)
ØØØA ØØØ2         SCBLK:RDBUF      RMB      2        POINTER TO READ DATA BUFFER (WHERE RESULT GOES)
ØØØC ØØØ2         SCBLK:RDLEN      RMB      2        CEILING ON SIZE OF REPLY (READ DATA BUFFER)
ØØØE ØØØØ         SCBLK:DATA       RMB      Ø        OTHER PARAMETERS FOR SYSCALL; UP TO 127-12 BYTES
ØØØE ØØØØ         SCBLK:END        RMB      Ø        END OF SYSCALL BLOCK; ASSERT SCBLK:WLEN[1..7]=SCBLK:
                  *
                  *        SYSCALL PARAMETER LIST DEFINITIONS
                  *
 ØØØ2             OPEN:CHANNEL     EQU      SCBLK:PARAMS     CHANNEL NUMBER
 ØØØ6             OPEN:LENGTH      EQU      SCBLK:WRLEN      FILE NAME LENGTH
 ØØØ4             OPEN:NAMEP       EQU      SCBLK:WRBUF      POINTER TO FILE NAME
 ØØØE             OPEN:SCLEN       EQU      SCBLK:DATA       OPEN SYSCALL BLOCK LENGTH
                  *
 ØØØ2             CREATE:CHANNEL   EQU      SCBLK:PARAMS     CHANNEL NUMBER
 ØØØ6             CREATE:LENGTH    EQU      SCBLK:WRLEN      FILE NAME LENGTH
 ØØØ4             CREATE:NAMEP     EQU      SCBLK:WRBUF      POINTER TO FILE NAME
 ØØØE             CREATE:SCLEN     EQU      SCBLK:DATA       CREATE SYSCALL BLOCK LENGTH
 ØØØE             CREATE:FILESIZE EQU       SCBLK:DATA       4 BYTE FILE SIZE INITIAL ALLOCATION
 ØØ12             CREATE:FILESIZESCLEN     EQU      CREATE:FILESIZE+4       END OF CREATE BLOCK WITH FIL
                  *
 ØØØ2             CLOSE:CHANNEL    EQU      SCBLK:PARAMS     CHANNEL NUMBER
 ØØØ3             CLOSE:SCLEN      EQU      SCBLK:PARAMS+1   CLOSE SYSCALL BLOCK LENGTH
                  *
 ØØØ2             RENAME:CHANNEL   EQU      SCBLK:PARAMS     CHANNEL NUMBER
 ØØØ6             RENAME:LENGTH    EQU      SCBLK:WRLEN      NEW FILE NAME LENGTH
 ØØØ4             RENAME:NAMEP     EQU      SCBLK:WRBUF      POINTER TO NEW FILE NAME
 ØØØE             RENAME:SCLEN     EQU      SCBLK:DATA       RENAME SYSCALL BLOCK LENGTH
                  *
 ØØØ6             DELETE:LENGTH    EQU      SCBLK:WRLEN      FILE NAME LENGTH
 ØØØ4             DELETE:NAMEP     EQU      SCBLK:WRBUF      POINTER TO NAME
 ØØØE             DELETE:SCLEN     EQU      SCBLK:DATA       DELETE SYSCALL BLOCK LENGTH
```

```
                *
ØØØ6            LOAD:LENGTH     EQU     SCBLK:WRLEN     LENGTH OF FILE NAME
ØØØ4            LOAD:NAMEP      EQU     SCBLK:WRBUF     POINTER TO FILE NAME
ØØØE            LOAD:SCLEN      EQU     SCBLK:DATA      LOAD SYSCALL BLOCK LENGTH
                *
ØØØ6            CHAIN:LENGTH    EQU     SCBLK:WRLEN     LENGTH OF FILE NAME
ØØØ4            CHAIN:NAMEP     EQU     SCBLK:WRBUF     POINTER TO FILE NAME
ØØØE            CHAIN:SCLEN     EQU     SCBLK:DATA      CHAIN SYSCALL BLOCK LENGTH
                *
ØØØ6            CREATELOG:LENGTH        EQU     SCBLK:WRLEN     LENGTH OF FILE NAME
ØØØ4            CREATELOG:NAMEP EQU     SCBLK:WRBUF     POINTER TO FILE NAME
ØØØE            CREATELOG:SCLEN EQU     SCBLK:DATA      CREATELOG SYSCALL BLOCK LENGTH
                *
ØØØ2            CLOSELOG:SCLEN  EQU     SCBLK:PARAMS    CLOSELOG SYSCALL BLOCK LENGTH
                *
ØØØ6            DISKDEFAULT:LENGTH      EQU     SCBLK:WRLEN     FILE NAME LENGTH
ØØØ4            DISKDEFAULT:NAMEP       EQU     SCBLK:WRBUF     POINTER TO FILE NAME
ØØØE            DISKDEFAULT:SCLEN       EQU     SCBLK:DATA      DISKDEFAULT SYSCALL BLOCK LENGTH
                *
ØØØ2            READA:CHANNEL   EQU     SCBLK:PARAMS    CHANNEL NUMBER
ØØØ3            READA:LMFLAG    EQU     SCBLK:PARAMS+1  LINE MODE FLAG BYTE
ØØØA            READA:BUFFERP   EQU     SCBLK:RDBUF     BUFFER POINTER
ØØØC            READA:MAXCOUNT  EQU     SCBLK:RDLEN     BYTE COUNT
ØØØ8            READA:ACTUALCOUNT       EQU     SCBLK:RPLEN     ACTUAL NUMBER OF BYTES TRANSFERRE
ØØØE            READA:SCLEN     EQU     SCBLK:DATA      READA SYSCALL BLOCK LENGTH
ØØØE            RW:POSITION     EQU     SCBLK:DATA      READ/WRITE IMPLICIT FILE POSITION
ØØ12            RWPOSITION:SCLEN        EQU     RW:POSITION+4   END OF R/W SYSCALL WITH IMPLICIT
                *
ØØØ2            READB:CHANNEL   EQU     SCBLK:PARAMS    CHANNEL NUMBER
ØØØA            READB:BUFFERP   EQU     SCBLK:RDBUF     BUFFER POINTER
ØØØC            READB:MAXCOUNT  EQU     SCBLK:RDLEN     BYTE COUNT
ØØØ8            READB:ACTUALCOUNT       EQU     SCBLK:RPLEN     ACTUAL NUMBER OF BYTES TRANSFERRE
ØØØE            READB:SCLEN     EQU     SCBLK:DATA      READB SYSCALL BLOCK LENGTH
                *
ØØØ2            WRITEA:CHANNEL  EQU     SCBLK:PARAMS    CHANNEL NUMBER
ØØØ4            WRITEA:BUFFERP  EQU     SCBLK:WRBUF     BUFFER POINTER
ØØØ6            WRITEA:COUNT    EQU     SCBLK:WRLEN     BYTE COUNT
ØØØ8            WRITEA:SCLEN    EQU     SCBLK:RPLEN     WRITEA SYSCALL BLOCK LENGTH
                *
ØØØ2            WRITEB:CHANNEL  EQU     SCBLK:PARAMS    CHANNEL NUMBER
```

```
 0004           WRITEB:BUFFERP  EQU     SCBLK:WRBUF      BUFFER POINTER
 0006           WRITEB:COUNT    EQU     SCBLK:WRLEN      BYTE COUNTER
 0008           WRITEB:SCLEN    EQU     SCBLK:RPLEN      WRITEB SYSCALL BLOCK LENGTH
                *
 0002           CONTROL:CHANNEL EQU     SCBLK:PARAMS     CHANNEL NUMBER
 0003           CONTROL:CODE    EQU     SCBLK:PARAMS+1   CONTROL CODE
 0004           CONTROL:SCLEN   EQU     SCBLK:WRBUF      CONTROL SYSCALL BLOCK MINIMUM LENGTH
 0000           CONTROL:DATA    EQU     0       DISPLACEMENT INTO WRITE BUFFER FOR CONTROL DATA
                *
 0002           STATUS:CHANNEL  EQU     SCBLK:PARAMS     CHANNEL NUMBER
 0003           STATUS:CODE     EQU     SCBLK:PARAMS+1   STATUS SELECTOR CODE
 000A           STATUS:BUFFERP  EQU     SCBLK:RDBUF      POINTER TO STATUS TARGET BUFFER
 000C           STATUS:MAXCOUNT EQU     SCBLK:RDLEN      SIZE OF STATUS READ-BACK BUFFER
 0008           STATUS:ACTUALCOUNT      EQU     SCBLK:RPLEN      ACTUAL # STATUS BYTES READ
 000E           STATUS:SCLEN    EQU     SCBLK:DATA       STATUS SYSCALL BLOCK MINIMUM LENGTH
 0000           STATUS:DATA     EQU     0       DISPLACEMENT INTO READ BUFFER FOR READ-BACK STATUS
                *
 0002           WAITDONE:CHANNEL        EQU     SCBLK:PARAMS     CHANNEL NUMBER
 0003           WAITDONE:SCLEN  EQU     SCBLK:PARAMS+1   WAITDONE SYSCALL BLOCK LENGTH
                *
 0002           EXIT:SCLEN      EQU     SCBLK:PARAMS     EXIT SYSCALL BLOCK LENGTH
                *
 0002           ERROREXIT:CODE  EQU     SCBLK:PARAMS     ERROR CODE NUMBER
 0004           ERROREXIT:SCLEN EQU     SCBLK:WRBUF      ERROREXIT SYSCALL BLOCK LENGTH
                *
 0002           SETERROR:CODE   EQU     SCBLK:PARAMS     ERROR CODE NUMBER
 0004           SETERROR:SCLEN  EQU     SCBLK:WRBUF      SETERROR SYSCALL BLOCK LENGTH
                *
 000A           GETERROR:BUFFERP        EQU     SCBLK:RDBUF      POINTER TO ERROR READ-BACK AREA
 000C           GETERROR:MAXCOUNT       EQU     SCBLK:RDLEN      SHOULD BE 2
 0008           GETERROR:ACTUALCOUNT    EQU     SCBLK:RPLEN      SHOULD BE RETURNED AS 2
 000E           GETERROR:SCLEN  EQU     SCBLK:DATA       GETERROR SYSCALL BLOCK LENGTH
                *
 0002           DISPERROR:SCLEN EQU     SCBLK:PARAMS     DISPERROR SYSCALL BLOCK LENGTH
                *
 0002           KILLPROOF:SCLEN EQU     SCBLK:PARAMS     KILLPROOF SYSCALL BLOCK LENGTH
                *
 0002           KILLENABLE:SCLEN        EQU     SCBLK:PARAMS     KILLENABLE SYSCALL BLOCK LENGTH
                *
 0002           DEBUG:SCLEN     EQU     SCBLK:PARAMS     DEBUG SYSCALL BLOCK LENGTH
```

```
                *
 0002           ATTNCHECK:SCLEN EQU     SCBLK:PARAMS    ATTNCHECK SYSCALL BLOCK LENGTH
                *
 0002           ISCONSOLE:SCLEN EQU     SCBLK:PARAMS    ISCONSOLE SYSCALL BLOCK LENGTH
                *
 0002           INTERLOCK:FUNCTION      EQU     SCBLK:PARAMS    INTERLOCK FUNCTION
 0004           INTERLOCK:BUFFERP       EQU     SCBLK:WRBUF     POINTER TO OBJECT
 0006           INTERLOCK:COUNT EQU     SCBLK:WRLEN     LENGTH OF OBJECT
 0008           INTERLOCK:SCLEN EQU     SCBLK:RPLEN     INTERLOCK SYSCALL BLOCK LENGTH
                *
 0002           DELAY:PERIOD    EQU     SCBLK:PARAMS    DELAY PERIOD
 0004           DELAY:SCLEN     EQU     SCBLK:WRBUF     DELAY SYSCALL BLOCK LENGTH
                *
                *READLUNNAME:LUN        EQU     SCBLK:PARAMS    LOGICAL UNIT NUMBER
                *READLUNNAME:BUFFERP    EQU     SCBLK:RDBUF     WHERE TO READ NAME BACK
                *READLUNNAME:MAXCOUNT   EQU     SCBLK:RDLEN     MAXIMUM LENGTH OF REPLY
                *READLUNNAME:ACTUALCOUNT        EQU     SCBLK:RPLEN     ACTUAL NAME LENGTH
 000E           READLUNNAME:SCLEN       EQU     SCBLK:DATA      READLUN SYSCALL BLOCK LENGTH
                *
 000A           GETSERIALNUMBER:BUFFERP EQU     SCBLK:RDBUF     POINTER TO SERIAL NUMBER REPLY BU
 000C           GETSERIALNUMBER:MAXCOUNT        EQU     SCBLK:RDLEN     SIZE OF BUFFER
 0008           GETSERIALNUMBER:ACTUALCOUNT     EQU     SCBLK:RPLEN     SIZE OF REPLY
 000E           GETSERIALNUMBER:SCLEN   EQU     SCBLK:DATA      MINIMUM SIZE OF BLOCK
                *
 0002           JOBCONTROL:FUNCTION     EQU     SCBLK:PARAMS    JOB CONTROL FUNCTION
 0004           JOBCONTROL:BUFFERP      EQU     SCBLK:WRBUF     POINTER TO JOB CAPABILITY
 0006           JOBCONTROL:COUNT        EQU     SCBLK:WRLEN     SIZE OF CAPABILITY
 000C           JOBCONTROL:MAXCOUNT     EQU     SCBLK:RDLEN     MAXIMUM LENGTH OF REPLY
 0008           JOBCONTROL:ACTUALCOUNT  EQU     SCBLK:RPLEN     ACTUAL SIZE OF CAPABILITY
 0008           JOBCONTROL:SCLEN        EQU     SCBLK:WRLEN+2   JOBCONTROL SYSCALL MINIMUM BLOCK
```

```
 0003      LSN:SIZE        EQU     3       # BYTES OCCUPIED BY AN LSN
 0002      LCN:SIZE        EQU     2       # BYTES OCCUPIED BY AN LCN
           *
           *       STANDARD STATUS SYSCALL SUB-CODES
           *
 0000              ORG     0
0000 0001  SC:GETPOS       RMB     1       GET BYTE POSITION
0001 0001  SC:GETCOL       RMB     1       GET COLUMN COUNT
0002 0001  SC:GETEOF       RMB     1       GET EOF FLAG
0003 0001  SC:GETFILESIZE  RMB     1       GET FILE SIZE
0004 0001  SC:GETTYPE      RMB     1       GET DEVICE TYPE AND CHARACTERISTICS
0005 0001  SC:GETPARAMS    RMB     1       GET DEVICE SPECIFIC PARAMETERS

 0010      SC:DEVICESPECIFICOP     EQU     $10     BASE FOR DEVICE SPECIFIC STATUS CODES
           *
           *       STANDARD CONTROL SYSCALL SUB-CODES
           *
 0000              ORG     0
0000 0001  CC:POSITION     RMB     1       POSITION TO THIS PLACE IN THE FILE
0001 0001  CC:DUMPBUFFERS  RMB     1       DUMP BUFFERS TO THE DEVICE (MAINLY FOR DISK)

 0010      CC:DEVICESPECIFICOP     EQU     $10     BASE FOR DEVICE-SPECIFIC CONTROL CODES
           *
           *       STANDARD INTERLOCK SYSCALL SUB-CODES
           *
 0000              ORG     0
0000 0001  IC:CREATE       RMB     1       CREATE AN OBJECT IDENTIFIER
0001 0001  IC:DESTROY      RMB     1       DESTROY AN OBJECT IDENTIFIER
0002 0001  IC:RESET        RMB     1       RESET OBJECT REFERENCE LIST
0003 0001  IC:LOCK RMB     1       LOCK AN OBJECT OR BLOCK UNTIL AVAILABLE
0004 0001  IC:RELEASE      RMB     1       RELEASE A LOCKED OBJECT
0005 0001  IC:TEST RMB     1       LOCK AN OBJECT OR ERROR IF UNAVAILABLE
```

```
                  *       VIRTUAL TERMINAL SPECIFIC STATUS REQUESTS
                  *
                  *
 0010                     ORG     SC:DEVICESPECIFICOP
0010 0001         SC:GETPROFILE   RMB     1       GET CURRENT DEVICE PROFILE NAME
0011 0001         SC:GETACTCOL    RMB     1       GET ACTIVATION COLUMN

 002C                     ORG     SC:DEVICESPECIFICOP+$1C (DON'T ASK!!!)
002C 0001         SC:GETLINEFLAGS RMB     1       GET LINE FLAGS
                  *
                  *       DISK FILE SPECIFIC STATUS REQUESTS
                  *
 0010                     ORG     SC:DEVICESPECIFICOP
0010 0001         SC:GETFILEDATE  RMB     1       READ BACK CREATION DATE OF FILE IN CLOCK FORMAT
0011 0001         SC:GETFILEPROT  RMB     1       READ BACK FILE PROTECTION BYTE
                  *
                  *       DISK DEVICE STATUS REQUESTS
                  *
 0010                     ORG     SC:DEVICESPECIFICOP
0010 0001         SC:GETLASTBADLSN        RMB     1       READ BACK LSN THAT CAUSED DRIVER A PROBLEM
0011 0001         SC:GETERRORSTATS        RMB     1       GET DEVICE ERROR (HISTORY) STATISTICS
                  *
                  *       VIRTUAL TERMINAL SPECIFIC CONTROL OPERATIONS
                  *
 0010                     ORG     CC:DEVICESPECIFICOP
0010 0001         CC:ECHO RMB     1       TURN ECHO ON
0011 0001         CC:NOECHO       RMB     1       TURN ECHO OFF
0012 0001         CC:IDLES        RMB     1       SET TTY IDLES
0013 0001         CC:TABS RMB     1       SET TTY TABS
0014 0001         CC:SETACTBLOCK  RMB     1       DECLARE ACTIVATION SET
0015 0001         CC:CLRINPUT     RMB     1       CLEAR INPUT BUFFER
0016 0001         CC:CLROUTPUT    RMB     1       CLEAR OUTPUT BUFFER
0017 0001         CC:SETREADTIMEOUT       RMB     1       SET TIMEOUT PERIOD FOR READA
0018 0001         CC:SETPROFILE   RMB     1       DECLARE DEVICE PROFILE
0019 0001         CC:ALTERPROFILE RMB     1       ALTER MALLEABLE DEVICE PROFILE
001A 0001         CC:WRITEEDITLINE        RMB     1       PUT LINE IN TYPE-AHEAD BUFFER
001B 0001         CC:SETFIELDSIZE RMB     1       DECLARE WIDTH OF INPUT FIELD
001C 0001         CC:SETPARAMS    RMB     1       DECLARE DEVICE WIDTH AND DEPTH
001D 0001         CC:ACTIVATIONCK RMB     1       CHECK FOR READA DATA READY
001E 0001         CC:WRAP RMB     1       ALLOW FORE- AND BACK-WRAP
```

```
001F 0001       CC:NOWRAP        RMB     1       DISALLOW FORE- AND BACK-WRAP
0020 0001       CC:COLORING      RMB     1       DECLARE AND SET FOREGROUND COLORING
0021 0001       CC:BACKGROUND    RMB     1       DECLARE AND SET BACKGROUND COLORING
0022 0001       CC:KILLPROOF     RMB     1       KILLPROOF VT DEVICE
0023 0001       CC:KILLENABLE    RMB     1       KILLENABLE VT DEVICE
                *
                *       DISK FILE SPECIFIC CONTROL OPERATIONS
                *
  0010                  ORG     CC:DEVICESPECIFICOP
0010 0001       CC:SETFILEDATE   RMB     1       SET CREATION DATE OF FILE (USE CLOCK FORMAT)
0011 0001       CC:SETFILEPROT   RMB     1       SET FILE PROTECTION BYTE
0012 0001       CC:SETFILESIZE   RMB     1       SET SIZE OF FILE
0013 0001       CC:POSITIONTOEND        RMB     1       POSITION TO END OF FILE
                *
                *       DISK DEVICE SPECIFIC CONTROL OPS
                *
  0010                  ORG     CC:DEVICESPECIFICOP
0010 0001       CC:UNLOCKDISK    RMB     1       UNLOCK THE DISK DEVICE FOR WRITING
0011 0001       CC:DISMOUNTDISK  RMB     1       DISMOUNT THE DISK
0012 0001       CC:SETMAPALGORITHM      RMB     1       SET MAP ALGORITHM NUMBER FOR DRIVE
0013 0001       CC:MULTISECTORREAD      RMB     1       READ MULTIPLE SECTORS
0014 0001       CC:MULTISECTORWRITE     RMB     1       WRITE MULTIPLE, ACCORDING TO SYSCALL EXTENSI
0015 0001       CC:FORMAT        RMB     1       FORMAT DISK
0016 0001       CC:WAITDONE      RMB     1       WAIT FOR CONTROLLER OPERATION COMPLETE
```

```
                *       RETURNED STATUS DISPLACEMENTS
                *
 0000                   ORG     0
0000 0004       STATUS:DIST     RMB     4       POSITION IN DISK FILE
                *
 0000                   ORG     0
0000 0001       STATUS:COLUMN   RMB     1       COLUMN NUMBER
                *
 0000                   ORG     0
0000 0001       STATUS:EOFFLAG  RMB     1       END OF FILE FLAG
                *
 0000                   ORG     0
0000 0001       STATUS:DEVTYPE  RMB     1       DEVICE TYPE DATA FOR DIRECTORIED DISK
                *
 0000                   ORG     0
0000 0002       STATUS:NBPS     RMB     2       NUMBER OF BYTES PER SECTOR
0002 0000       STATUS:NSPC     RMB     0       NUMBER OF SECTORS PER CLUSTER FOR DISK FILE
0002 0002       STATUS:NSPT     RMB     2       NUMBER OF SECTORS PER TRACK
0004 0002       STATUS:NTPC     RMB     2       NUMBER OF TRACKS PER CYLINDER
0006 0002       STATUS:NCYL     RMB     2       NUMBER OF CYLINDERS
                *
 0000                   ORG     0
0000 0004       STATUS:FILESIZE RMB     4       SIZE OF DISK FILE IN BYTES
                *
 0000                   ORG     0
0000 0003       STATUS:LASTBADLSN       RMB     3       LSN OF LAST BAD SECTOR ON DISK
                *
                *       SC:GETFILEDATE REPLY BUFFER
                *
 0000                   ORG     0
0000 0003       STATUS:DATETICKS        RMB     3       24 BITS OF TICKS SINCE MIDNITE
0003 0001       STATUS:DATEDAY  RMB     1       BCD VALUE OF DAY (1..31)
0004 0001       STATUS:DATEMONTH        RMB     1       BCD VALUE OF MONTH (1..12)
0005 0001       STATUS:DATEYEAR RMB     1       BCD VALUE OF YEAR MOD 100 (00.99)
                *
                *       SC:GETFILEPROT REPLY BUFFER
                *
 0000                   ORG     0
0000 0001       STATUS:PROT     RMB     1       PROTECTION BYTE FROM FILE
```

```
                361: *       SC:GETERRORSTATS REPLY BUFFER
                362: *
  ØØØØ          363:         ORG     Ø
ØØØØ ØØØ2       364: STATUS:SEEKERRCNT       RMB     2       # SEEK ERRORS SINCE MOUNT
ØØØ2 ØØØ2       365: STATUS:SEEKERRSTS       RMB     2       16 BITS OF LAST "SEEK" STATUS IN ERROR
ØØØ4 ØØØ2       366: STATUS:WRITEERRCNT      RMB     2       # WRITE ERRORS SINCE MOUNT
ØØØ6 ØØØ2       367: STATUS:WRITEERRSTS      RMB     2       16 BITS OF LAST "WRITE" STATUS IN ERROR
ØØØ8 ØØØ2       368: STATUS:READERRCNT       RMB     2       # READ ERRORS SINCE MOUNT
ØØØA ØØØ2       369: STATUS:READERRSTS       RMB     2       16 BITS OF LAST "READ" STATUS IN ERROR
ØØØC ØØØ3       37Ø: STATUS:OPSCOUNT RMB     3       24 BITS OF # DRIVER OPERATIONS SINCE MOUNT
ØØØF ØØØ3       371: STATUS:ERRLSN   RMB     LSN:SIZE        LSN CAUSING ANY SOFT OR HARD ERROR
```

```
                 *        CC:POSITION WRITE BUFFER
                 *
 0000                     ORG     0
0000 0004        CONTROL:DIST     RMB     4       VALUE OF POSITIONING COMMANDS
                 *
                 *        CC:SETFILEDATE WRITE BUFFER
                 *
 0000                     ORG     0
0000 0003        CONTROL:DATETICKS        RMB     3       24 BITS OF TICKS SINCE MIDNITE
0003 0001        CONTROL:DATEDAY RMB      1       BCD VALUE OF DAY (1..31)
0004 0001        CONTROL:DATEMONTH        RMB     1       BCD VALUE OF MONTH (1..12)
0005 0001        CONTROL:DATEYEAR         RMB     1       BCD VALUE OF YEAR MOD 100 (00..99)
                 *
                 *        CC:SETFILEPROT WRITE BUFFER
                 *
 0000                     ORG     0
0000 0001        CONTROL:PROT     RMB     1       PROTECTION BYTE FOR FILE
                 *
                 *        CC:SETMAPALGORITHM WRITE BUFFER
                 *
 0000                     ORG     0
0000 0002        CONTROL:MAPALGORITHM     RMB     2       PARAMETER BLOCK FOR SET MAP ALGORITHM CALL
                 *
                 *        JOB CONTROL SUB-CODES
                 *
 0000                     ORG     0
0000 0001        JC:CREATE        RMB     1       CREATE A NEW JOB
0001 0001        JC:TESTDONE      RMB     1       TEST TO SEE IF A JOB IS DONE
0002 0001        JC:DESTROY       RMB     1       DESTROY A JOB
```

```
                *
                *       SYSTEM-DEFINED ERROR CODES
                *
 0000                   ORG     0
0000 0001       ERR:NONE        RMB     1       CODE 0 --> NO ERROR
0001 0001       ERR:ATTENTION   RMB     1       OPERATOR REQUESTED ATTENTION

 0064                   ORG     100
0064 0001       ERR:FATALCOMPILE        RMB     1       COMPILATION OR ASSEMBLY HAD FATAL ERRORS
0065 0001       ERR:WARNINGCOMPILE      RMB     1       COMPILATION OR ASSEMBLY HAD NON-FATAL ERRORS
0066 0001       ERR:BADCMDFORMAT        RMB     1       BAD COMMAND FORMAT (SYNTAX ERROR!)
0067 0001       ERR:CANTGOTO    RMB     1       CAN'T DO GOTO FROM CONSOLE:
0068 0001       ERR:ABNORMALSTOP        RMB     1       PROGRAM TERMINATED ABNORMALLY
0069 0001       ERR:NOTENUFMEM  RMB     1       NOT ENOUGH MEMORY TO EXECUTE COMMAND
                *
                *       SDOS ERROR CODES
                *
                *
                *       ERROR CODES FOR SDOS ARE RESERVED BETWEEN 1000-1999
                *
 03E8                   ORG     1000
03E8 0001       ERR:BOOTCKSUMFAIL       RMB     1       BOOT SECTOR DISKINFO CHECK SUM FAILED
03E9 0001       ERR:EOFHIT      RMB     1       END OF FILE HIT
03EA 0001       ERR:FILEISOPEN  RMB     1       A FILE IS OPEN DURING DISMOUNT REQUEST
03EB 0001       ERR:NODEBUGGER  RMB     1       NO DEBUGGER TO CALL!
03EC 0001       ERR:BADPOSITION RMB     1       BAD POSITIONING REQUEST
03ED 0001       ERR:NBPCTOOBIG  RMB     1       NUMBER OF BYTES PER CLUSTER >= 65536
03EE 0001       ERR:NODISKMAP   RMB     1       NO DISK MAP, CAN'T ALLOC OR FREE
03EF 0001       ERR:NOMATCHFCB  RMB     1       NO MATCHING FILE CONTROL BLOCK FOUND
03F0 0001       ERR:NODEFAULTPROGRAM    RMB     1       NO "DEFAULTPROGRAM" ON THIS DISK
03F1 0001               RMB     1       **** UNUSED ****
03F2 0001       ERR:FILEWRTPROT RMB     1       FILE IS WRITE PROTECTED
03F3 0001       ERR:FILENOTFOUND        RMB     1       FILE NOT FOUND
03F4 0001       ERR:ILLEGALLCN  RMB     1       LCN OUT OF RANGE
03F5 0001       ERR:BADFNAMESIZE        RMB     1       LENGTH OF FILE NAME > 16 CHARACTERS
03F6 0001       ERR:NEWFILEEXISTS       RMB     1       NEW FILE ALREADY EXISTS!
03F7 0001       ERR:NODISKSPACE RMB     1       DISK SPACE EXHAUSTED
03F8 0001       ERR:LCNWASNTALLOCATED   RMB     1       LCN ENCOUNTERED BY FREECLUSTERS WASN'T ALLOC
03F9 0001       ERR:NOFREEFCBS  RMB     1       RAN OUT OF FCBS (*SYSTEM*)
03FA 0001       ERR:WRONGFILESYSTEM     RMB     1       FILE SYSTEM INCOMPATIBLE WITH THIS VERSION O
```

```
Ø3FB ØØØ1        443: ERR:FILEINCREATE        RMB      1       FILE IS BEING CREATED
Ø3FC ØØØ1        444: ERR:DISKMOUNTED RMB     1       DISK IS MOUNTED, CAN'T CHANGE MAPALGORITHM
Ø3FD ØØØ1        445: ERR:CANTOPENMUSTCREATE  RMB      1       MUST CREATE TO OPEN OUTPUT ONLY DEVICE
Ø3FE ØØØ1        446: ERR:NOERRORMSGS RMB     1       NO $ERRORMESSAGES FILE ON DRIVE Ø
Ø3FF ØØØ1        447: ERR:BADFILENAME RMB     1       FILENAME DOESN'T START WITH A-Z OR $
Ø4ØØ ØØØ1        448: ERR:ILLFILESIZE RMB     1       ILLEGAL FILE SIZE SPECIFICATION (SYNTAX OR OVFLOW)
Ø4Ø1 ØØØ1        449: ERR:HCSICTOOSMALL       RMB      1       HEADER CLUSTER NOT INITZED FOR RDCN FETCH
Ø4Ø2 ØØØ1        45Ø: ERR:NOTENOUGHPOOL       RMB      1       NOT ENOUGH DISKBUFFER POOL (*SYSTEM*)
Ø4Ø3 ØØØ1        451: ERR:PWRFAILDISKF        RMB      1       DISK FILE HANDLERS DON'T IMPLEMENT POWER F
Ø4Ø4 ØØØ1        452: ERR:NOTALOADFILE        RMB      1       CAN'T LOAD THAT - WRONG FORMAT
Ø4Ø5 ØØØ1        453: ERR:BADFILEVERSION      RMB      1       FILE VERSION NUMBER HAS NO DIGITS OR IS >2
Ø4Ø6 ØØØ1        454: ERR:CHTOOBIG    RMB     1       CHANNEL # IS TOO BIG
Ø4Ø7 ØØØ1        455: ERR:CHBUSY      RMB     1       CHANNEL IS ALREADY OPEN
Ø4Ø8 ØØØ1        456: ERR:CLOSED      RMB     1       CHANNEL IS ALREADY CLOSED
Ø4Ø9 ØØØ1        457: ERR:ILLEGALSYSCALL      RMB      1       ILLEGAL SYSCALL #
Ø4ØA ØØØ1        458: ERR:ILLDEVICEOP RMB     1       ILLEGAL DEVICE OPERATION
Ø4ØB ØØØ1        459: ERR:RENAMEDEVICE        RMB      1       CAN'T RENAME TO DIFFERENT DEVICE
Ø4ØC ØØØ1        46Ø: ERR:BADLOADRECORD       RMB      1       LOAD RECORD FORMAT ERROR
Ø4ØD ØØØ1        461: ERR:NOTENOUGHROOM       RMB      1       PROGRAM TOO BIG TO LOAD
Ø4ØE ØØØ1        462: ERR:ILLLSN      RMB     1       ILLEGAL LSN PASSED TO PHYSICAL DISK DRIVERS
Ø4ØF ØØØ1        463: ERR:DIRECTORYDAMAGED    RMB      1       DIRECTORY.SYS DIRECTORY ENTRY IS DAMAGED
Ø41Ø ØØØ1        464: ERR:IBUFOVERFLOW        RMB      1       INPUT BUFFER OVERFLOW IN THE DRIVERS
Ø411 ØØØ1        465: ERR:PROGRAMKILLED       RMB      1       PROGRAM KILLED BY OPERATOR
Ø412 ØØØ1        466: ERR:DEVICETIMEDOUT      RMB      1       DEVICE TIMED OUT
Ø413 ØØØ1        467: ERR:SECTORSIZE2 RMB     1       SECTORSIZE IS NOT A POWER OF 2!
Ø414 ØØØ1        468: ERR:SYSTEMCROAKED       RMB      1       ...WHILE DOING AN EXIT OR CHAIN (*SYSTEM*)
Ø415 ØØØ1        469: ERR:DISKREAD    RMB     1       DISK READ ERROR
Ø416 ØØØ1        47Ø: ERR:DISKWRITE   RMB     1       DISK WRITE ERROR
Ø417 ØØØ1        471: ERR:DISKSEEK    RMB     1       DISK SEEK ERROR
Ø418 ØØØ1        472: ERR:DSKWRTPROT  RMB     1       DISK IS WRITE PROTECTED
Ø419 ØØØ1        473: ERR:DISKWRITELOCKED     RMB      1       DISK DEVICE IS SOFTWARE WRITE LOCKED
Ø41A ØØØ1        474: ERR:SDOSCKSUM   RMB     1       SDOS GOT A KNIFE IN THE RIBS!
Ø41B ØØØ1        475: ERR:NLSNGE224   RMB     1       NLSN >= 2^24, ILLEGAL
Ø41C ØØØ1        476: ERR:CLUSTERSIZELIMITSFILE       RMB     1       CLUSTER SIZE IS TOO SMALL TO SUPPC
Ø41D ØØØ1        477: ERR:SYSCALLTOOSHORT     RMB      1       SYSCALL BLOCK IS TOO SMALL FOR SPECIFIED S
Ø41E ØØØ1        478: ERR:RDBUFTOOSMALL       RMB      1       READ BUFFER SPECIFIED BY SYSCALL IS TOO SH
Ø41F ØØØ1        479: ERR:WRBUFTOOSMALL       RMB      1       WRITE BUFFER SPECIFIED BY SYSCALL IS TOO S
Ø42Ø ØØØ1        48Ø: ERR:NOSUCHDEVICE        RMB      1       NO SUCH DEVICE IN THIS CONFIGURATION
Ø421 ØØØ1        481: ERR:DEVICEERRORED       RMB      1       DEVICE HARDWARE DID NOT RESPOND REASONABLY
Ø422 ØØØ1        482: ERR:MUSTBEDISK  RMB     1       MUST SELECT DISK DEVICE
```

```
0423 0001      ERR:NOTOPENTOCONSOLE    RMB     1       CHANNEL 0 IS NOT OPEN TO CONSOLE DEVICE
0424 0001      ERR:DEVICENOTREADY      RMB     1       DEVICE IS NOT READY
0425 0001      ERR:TIMENOTSET  RMB     1       TIME NOT SET TO NON-ZERO DAY/MONTH!
0426 0001      ERR:NOSUCHLUN   RMB     1       NO SUCH LOGICAL UNIT NUMBER
0427 0001      ERR:ZEROSTARTADDRESS    RMB     1       OBJECT FILE HAS NO (ZERO) START ADDRESS
0428 0001      ERR:NOSUCHPROGRAM       RMB     1       NO SUCH PROGRAM EXISTS (ERROR ISSUED BY LOAD
0429 0001      ERR:OLDFILEEXISTS       RMB     1       OLD FILE BY SAME NAME ALREADY EXISTS
042A 0001              RMB     1       *** UNUSED ***
042B 0001      ERR:ALLOC0CLUSTERS      RMB     1       "ALLOC" CALL WITH REQUEST FOR 0 CLUSTERS!
042C 0001      ERR:FILEALREADYDELETED  RMB     1       FILE WAS DELETED BUT NOT CLOSED BEFORE RENAM
042D 0001      ERR:PRINTERNOTREADY     RMB     1       PRINTER IS NOT READY
042E 0001      ERR:INPUTTIMEOUT        RMB     1       INPUT TIMED OUT, ABORTED
042F 0001      ERR:ENDCFMEDIUM RMB     1       END OF MEDIUM ON DEVICE
0430 0001      ERR:SELFTESTCKSUM       RMB     1       PROGRAM SELF-TEST CHECKSUM FAILED
0431 0001      ERR:NOTIMEOUTBLKS       RMB     1       ZERO TIME OUT BLOCKS IN I/O PKG NOT LEGAL
0432 0001      ERR:SERIALNOWRONG       RMB     1       THIS CPU HAS WRONG SERIAL NUMBER TO RUN PROG
0433 0001      ERR:NOSUCHKEY   RMB     1       NO SUCH KEY EXISTS IN INDEX
0434 0001      ERR:DUPLICATEKEY        RMB     1       KEY ALREADY EXISTS IN INDEX
0435 0001      ERR:BRANCHFACTORSIZE    RMB     1       KEY BRANCHING FACTOR IS TOO SMALL
0436 0001      ERR:SDOSNOTREGISTERED   RMB     1       THIS COPY OF SDOS NOT REGISTERED WITH SD YET
0437 0001      ERR:DECRYPTIONKEYSDONTMATCH     RMB     1       LAST FILE LOADED HAS DIFFERENT DECRY
               *
  076E         ERR:WRONGDISKTYPE       EQU     1902    WRONG DISK TECHNOLOGY (DENSITY, SIDES, ETC.)
               *
               *       VIRTUAL TERMINAL DRIVER ERROR CODES
               *
  0771                 ORG     1905
0771 0001      ERR:IOINPROGRESS        RMB     1       LAST REQUEST HAS NOT COMPLETED
0772 0001      ERR:BUSYFORANOTHERPROCESS       RMB     1       DCB OPEN TO ANOTHER PROCESS
0773 0001      ERR:ACTIVATIONNOTINBUFFER       RMB     1       RDBUF DOES NOT HOLD ACTIVATION
0774 0001      ERR:BADFIELDWIDTH       RMB     1       CRT SCREEN FEILD SPECIFICATION IS TOO WIDE
0775 0001      ERR:ACTIVATIONRECEIVED  RMB     1       ACTIV. REC'D PER CC:ACTIVATIONCK
0776 0001      ERR:TIMEDINPUTEXPIRED   RMB     1       TIMED INPUT PERIOD EXPIRED
0777 0001      ERR:PROFILENOTFOUND     RMB     1       DEVICE PROFILE NOT FOUND
0778 0001      ERR:PROFILENOTMALLEABLE RMB     1       DEVICE PROFILE NOT MALLEABLE
0779 0001              RMB     1       *** RESERVED ***
               *
  04B0                 ORG     1200    SDOS/MT ERROR CODES
04B0 0001      ERR:BADREADBUF  RMB     1       SYSCALL REPLY BUFFER NOT WITHIN USER SPACE
04B1 0001      ERR:BADWRITEBUF RMB     1       SYSCALL WRITE BUFFER NOT WITHIN USER SPACE
```

```
Ø4B2 ØØØ1        523: ERR:RDBUFTOOBIG RMB    1       SYSCALL REPLY BUFFER > 255 BYTES
Ø4B3 ØØØ1        524: ERR:WRBUFTOOBIG RMB    1       SYSCALL WRITE BUFFER > 255 BYTES
Ø4B4 ØØØ1        525: ERR:NOTENOUGHCHANNELS  RMB     1       AVAILABLE I/O CHANNELS EXHAUSTED
Ø4B5 ØØØ1        526: ERR:NOTUNDERTIMESHARE  RMB     1       FUNCTION NOT AVAILABLE UNDER SDOS/MT
Ø4B6 ØØØ1        527: ERR:MTNOROOM    RMB    1       NOT ENOUGH ROOM TO RUN SDOS/MT
Ø4B7 ØØØ1        528: ERR:MTBADCONFIG RMB    1       INCORRECT CONFIGURATION FOR SDOS/MT
Ø4B8 ØØØ1        529: ERR:ALREADYLOCKED      RMB     1       INTERLOCK OBJECT IS ALREADY LOCKED
Ø4B9 ØØØ1        530: ERR:NOSUCHOBJECT       RMB     1       BAD CAPABILITY GIVEN
Ø4BA ØØØ1        531: ERR:NOTLOCKED   RMB    1       INTERLOCK OBJECT IS NOT LOCKED
Ø4BB ØØØ1        532: ERR:OBJECTDESTROYED    RMB     1       INTERLOCK OBJECT DESTROYED WHILE WAITING
Ø4BC ØØØ1        533: ERR:LOCKRESET   RMB    1       INTERLOCK OBJECT WAS RESET WHILE WAITING FOR IT
Ø4BD ØØØ1        534: ERR:IMPLEMENTATIONLIMITREACHED  RMB     1       CAN'T HANDLE MORE INTERLOCK OBJEC
Ø4BE ØØØ1        535: ERR:ILLEGALINTERLOCKFUNCTION    RMB     1       ILLEGAL INTERLOCK FUNCTION REQUES
Ø4BF ØØØ1        536: ERR:MEMORYMGMTFAIL     RMB     1       SDOS/MT INTERNAL MEMORY MANAGEMENT FAILUR
Ø4CØ ØØØ1        537: ERR:NOMOREJOBS  RMB    1       ALL AVAILABLE JOBS ARE BUSY NOW
Ø4C1 ØØØ1        538: ERR:ILLEGALJOBCONTROL  RMB     1       ILLEGAL JOB CONTROL REQUEST
Ø4C2 ØØØ1        539: ERR:CAPABILITYFAILURE  RMB     1       CAPABILITY DOES NOT HAVE RIGHTS TO PERFOR
Ø4C3 ØØØ1        54Ø: ERR:JOBKILLED   RMB    1       THIS JOB HAS BEEN KILLED BY ANOTHER
Ø4C4 ØØØ1        541: ERR:JOBCOMPLETED       RMB     1       JOB SUCCESSFULLY COMPLETED
```

```
                    *       DEVICE TYPE DEFINITIONS
                    *
 0000                       ORG     0
0000 0001           DVTYP.FILE      RMB     1       FILE (MANAGED BY SDOS)
0001 0001           DVTYP.DISK      RMB     1       DISK DEVICE (MANAGED BY SDOS)
0002 0001           DVTYP.STAPE     RMB     1       SERIAL TAPE DEVICE
0003 0001           DVTYP.DTAPE     RMB     1       DIRECTORIED TAPE DEVICE
0004 0001           DVTYP.CONSOLE   RMB     1       CONSOLE (HUMAN'S INTERFACE)
0005 0001           DVTYP.PRINTER   RMB     1       LINE PRINTER DEVICE
0006 0001           DVTYP.SERIALOUT RMB     1       ILL-DEFINED
0007 0001           DVTYP.SERIALIN  RMB     1
0008 0001           DVTYP.PAROUT    RMB     1       PARALLEL OUT
0009 0001           DVTYP.PARIN     RMB     1       PARALLEL IN
000A 0001           DVTYP.DUMMY     RMB     1       BLACK HOLE FOR DATA BYTES
000B 0001           DVTYP.CLOCK     RMB     1       CLOCK DEVICE
                    *
                    *
                    *
                    *
                    *       DEVICE TYPE DATA DISPLACEMENTS
                    *
 0000                       ORG     0
0000 0001           DVTYP:TYPE      RMB     1       DEVICE TYPE
                    *
                    *       DEVICE-TYPE SPECIFIC DATA
                    *
 0000                       ORG     0       DISK DEVICE SPECIFIC DATA
0000 0002           DVDAT:NBPS      RMB     2       NUMBER OF BYTES PER SECTOR
0002 0002           DVDAT:NSPT      RMB     2       NUMBER OF SECTORS PER TRACK
0004 0002           DVDAT.NTPC      RMB     2       NUMBER OF TRACKS PER CYLINDER
0006 0002           DVDAT:NCYL      RMB     2       NUMBER OF CYLINDERS
                    *
 0000                       ORG     0       CONSOLE/PRINTER DEVICE SPECIFIC DATA
0000 0001           DVDAT:WIDTH     RMB     1       LINE WIDTH IN CHARACTERS
0001 0001           DVDAT:DEPTH     RMB     1       PAGE DEPTH (DEFAULT DEPTH FOR PRINTERS)
                    *                               (0=INFINITY)
                    *
 0002                       ORG     DVDAT:NBPS+2    (DISK) FILE DEVICE SPECIFIC DATA
0002 0001           DVDAT:NSPC      RMB     1       NUMBER OF SECTORS PER CLUSTER
```

```
                 ***********************************************************************
 0010            FILESYSTEMVERSION      EQU     $10     VERSION 1.0 OF FILESYSTEM FORMAT
                 **********************************************************************
                 *
                 *       USEFUL ERROR-HANDLING OPCODES
                 *
 0C39            OKRTS   EQU     $0C39   "CLC,RTS"
 0D39            ERRORRTS        EQU     $0D39   "SEC,RTS"
                 *
                 *       FUNNY VALUES TO MAKE DATA STORAGE ALLOCATION USES MORE CLEAR
                 *
 0000            IGNORED EQU     0       SO I CAN MARK PLACES AS IGNORED
 0000            CHANGED EQU     0       SO I CAN MARK PLACES AS CHANGED
                 *
                 *       LOCATIONS 0-7 ARE TREATED AS PART OF TASK'S CONTEXT
                 *       AND SAVED DURING A CONTEXT SWITCH
                 *
                 *       DEFINED TEMPORARIES
                 *       USED TO PUSH (X) ONTO STACK IN INTERRUPTABLE WAY
                 *       FOR USE BY TASK-LEVEL SUBROUTINES
                 *
 0000                    ORG     $0
0000 0002        TEMPX   RMB     2       ANY SUBROUTINE MAY STEP ON THIS!!!
 0000            TEMP    EQU     TEMPX   FOR CONVENIENCE
 0000            TEMPA   EQU     TEMPX   TEMP STORAGE FOR A REGISTER
 0001            TEMPB   EQU     TEMPX+1 TEMP STORAGE FOR B REGISTER
                 *
                 *       PROTECTION BITS FOR DIR:PROTECTION
                 *
 0040            PROT::WRITE     EQU     $40     PROTECT AGAINST WRITES
 0001            PROT::BACKUP    EQU     $1      PROTECT AGAINST BACKING UP
```

```
                73: *       ASCII CHARACTER SET
                74: *
 0000           75: ASCII:NULL      EQU   0      ^@ NULL
 0001           76: ASCII:SOH       EQU   1      ^A START OF HEADING
 0002           77: ASCII:STX       EQU   2      ^B START OF TEXT
 0003           78: ASCII:ETX       EQU   3      ^C END OF TEXT
 0004           79: ASCII:EOT       EQU   4      ^D END OF TRANSMISSION
 0005           80: ASCII:ENQ       EQU   5      ^E ENQUIRY (WRU- WHO ARE YOU)
 0006           81: ASCII:ACK       EQU   6      ^F ACKNOWLEDGE
 0007           82: ASCII:BEL       EQU   7      ^G BELL
 0008           83: ASCII:BS        EQU   8      ^H BACKSPACE
 0009           84: ASCII:HT        EQU   9      ^I HORIZONTAL TAB
 000A           85: ASCII:LF        EQU   $A     ^J LINE FEED
 000B           86: ASCII:VT        EQU   $B     ^K VERTICAL TAB
 000C           87: ASCII:FF        EQU   $C     ^L FORM FEED
 000D           88: ASCII:CR        EQU   $D     ^M CARRIAGE RETURN
 000E           89: ASCII:SO        EQU   $E     ^N SHIFT OUT
 000F           90: ASCII:SI        EQU   $F     ^O SHIFT IN
 0010           91: ASCII:DLE       EQU   $10    ^P DATA LINK ESCAPE
 0011           92: ASCII:DC1       EQU   $11    ^Q DEVICE CONTROL 1
 0012           93: ASCII:DC2       EQU   $12    ^R DEVICE CONTROL 2
 0013           94: ASCII:DC3       EQU   $13    ^S DEVICE CONTROL 3
 0014           95: ASCII:DC4       EQU   $14    ^T DEVICE CONTROL 4
 0015           96: ASCII:NAK       EQU   $15    ^U NEGATIVE ACKNOWLEDGE
 0016           97: ASCII:SYN       EQU   $16    ^V SYNCHRONOUS IDLE
 0017           98: ASCII:ETB       EQU   $17    ^W END OF TRANSMISSION BLOCK
 0018           99: ASCII:CAN       EQU   $18    ^X CANCEL
 0019          100: ASCII:EM        EQU   $19    ^Y END OF MEDIUM
 001A          101: ASCII:SUB       EQU   $1A    ^Z SUBSTITUTE
 001B          102: ASCII:ESC       EQU   $1B    ^[ ESCAPE
 001C          103: ASCII:FS        EQU   $1C    ^\ FILE SEPERATOR
 001D          104: ASCII:GS        EQU   $1D    ^] GROUP SEPERATOR
 001E          105: ASCII:RS        EQU   $1E    ^^ RECORD SEPERATOR
 001F          106: ASCII:US        EQU   $1F    ^_ UNIT SEPERATOR
 0020          107: ASCII:SPACE     EQU   $20    SPACE (WORD SEPERATOR)
 007F          108: ASCII:RUBOUT    EQU   $7F    DELETE (RUBOUT)
 007F          109: ASCII:MASK      EQU   $7F    TO MASK OFF ALL BUT 7 LEGAL ASCII BITS
```

```
                *                   Extensions to the Configuration Table

  ØØØ2          ::                  set    *
  ØØØ2                              org    cnfg:timeoutlist+2


ØØØ2 ØØØ2       cnfg:vtprofiles     rmb    2            head of profile chain
ØØØ4 ØØØ2       cnfg:vtdebug        rmb    2            interrupt level ep to debugger
ØØØ6 ØØØ2       cnfg:mtprims        rmb    2            -> MT primitives vector

  ØØØ8          cnfg:vtsize         equ    *
  ØØØ2                              org    ::

                *                   VT User calls

                *                   Control calls

  ØØ3Ø                                              org   cc:devicespecificop+$2Ø
ØØ3Ø ØØØ1       cc:writeanowait                     rmb   1      write ascii, do not block
ØØ31 ØØØ1       cc:settimeshare                     rmb   1      set the timeshare flag
ØØ32 ØØØ1       cc:setexception                     rmb   1      set/clear exception flags
ØØ33 ØØØ1       cc:writebnowait                     rmb   1      write binary, do not block
ØØ34 ØØØ1       cc:stoptimeshare                    rmb   1      disable timesharing

                *                   Status calls

  ØØ3Ø                                              org   sc:devicespecificop+$2Ø
ØØ3Ø ØØØ1       sc:attentionck                      rmb   1      check for attention  (s/u)
ØØ31 ØØØ1       sc:statusck                         rmb   1      check for change of status (s/u)
ØØ32 ØØØ1       sc:gettimeshare                     rmb   1      check for SDOS/MT running (MT)
ØØ33 ØØØ1       sc:allstatus                        rmb   1      check for change of status on any
  ØØØ1                              ifund   sc:getlineflagshint ; kluge around SDOS11DEFS
ØØ34 ØØØ1       sc:getlineflagshint                 rmb   1      returns lineflags w/o clearing
  ØØØ1                              else
                                    fin
ØØ35 ØØØ1       sc:getfreecount                     rmb   1      returns dcb:tlroom
ØØ36 ØØØ1       sc:getdatacount                     rmb   1      returns dcb:tldata

  ØØFF          sysdependent                        equ   $ff    system dependent
```

```
  00FF          358: lineflags                              equ   sysdependent
```

```
                  *                             Errors specific to SDOS/MT

  0002            ::                      set     *
  04CE                                    org     1230

04CE 0001         err:sdosmtalreadyrunning        rmb     1       SDOS/MT is already running
04CF 0001         err:statushaschanged            rmb     1       port status has changed since las
04D0 0001         err:sdosmtprimsmissing          rmb     1       SDOS/MT primitives not defined in

  0002                                    org     ::

                          END      ;<<Supplied By ASM>>
```
 *** End of Source File Encountered.

INDEX