ATARI GEMDOS
–
REFERENCE MANUAL


April 4, 1986




TABLE OF CONTENTS

## INTRODUCTION

```
 _____
| THIS IS A PRELIMINARY DOCUMENT|
| AND IT DOES NOT CLAIM TO      |
| PERFECTLY DESCRIBE REALITY    |
| (or even GEMDOS).   PLEASE    |
| REPORT BUGS AND TYPOS TO      |
|       ATARI.   THANKS!        |
 _____
```

This is the Atari GEMDOS User's Manual.  It describes the internals and use of GEMDOS on the Atari ST.  This manual is divided into three parts; a tutorial and introduction for beginning users, a reference manual for application writers, and appendices for GEMDOS wizards.

The GEMDOS Tutorial is a gentle introduction to the basics of GEMDOS.  Its intention is to get beginning users started as quickly as possible.  It gives example programs, designed to exercise most of GEMDOS, which combine into a simple commandline interface, or "shell".  The tutorial also covers common pitfalls and useful shortcuts.

The GEMDOS Reference Manual is the application-writer's bible.  It covers GEMDOS' calling conventions, file and handle manipulation, process execution, and every GEMDOS call.

The Appendices contain nitty-gritty details and hints for those who have to push GEMDOS to the limit.  They are for application writers (and the merely curious) who have "need to know" about obscurities in the system.

To use this manual effectively readers should be familiar with C and 68000 assembly language.  Familiarity with MSDOS, Unix[1], and the standard C runtime library will also help.

_____

[1] Unix ist ein eingetragenes Warenzeichen der Bell Laboratories.

CALLING CONVENTIONS
        GEMDOS uses the Alcyon (or Digital Research) C calling
conventions.   Note   that   these conventions may differ from
other 68000 C compilers.  If you are using   another   C   com-
piler  it  might   not   be   possible to call GEMDOS directly;
please check your compiler's documentation   for   compatibil-
ity.

        Arguments are pushed on the stack, in reverse order  of
their  declaration.   The   GEMDOS   function number is pushed
last, as a WORD.  To do the call to GEMDOS,   a   68000   "TRAP
#1"  instruction is executed.  The trap can be made with the
68000 in user or supervisor mode.

NOTE
                Applications running in  supervisor  mode  may  be
                forced  back into user mode after making a GEM AES
                call.

                        Stack Snapshot
                (Just Before a GEMDOS Trap)


| stack | contents |
|-------|----------|
| (sp)  | WORD function number |
| 2(sp) | argument 1 |
| X(sp) | argument 2 |
| Y(sp) | argument 3 |
| . | . |
| . | ... and so on ... |
| . | . |


        Results are returned in D0.  Registers D0-D2 and   A0-A2
can   be   modified;   registers D3-D7 and A3-A7 will always be
preserved.  The caller is responsible for popping the   argu-
ments (including the function number) off of the stack after
the call.

        The Alcyon C compiler does not generate  TRAP   instruc-
tions,  so  most  applications use a small assembly-language
binding.  It typically looks like:

```
        text
*+
*   GEMDOS binding for Alcyon C
*
*   NOTE:
*     This binding is NOT re-entrant, and cannot
*     be shared by foreground and interrupt code.
*
*-
        .globl  _gemdos
_gemdos:
        move.l  (sp)+,t1sav      ; save ret addr
        trap    #1               ; call GEMDOS
        move.l  t1sav,-(sp)      ; restore ret addr
        rts                      ; do "real" return

      bss
t1sav:  ds.l    1                ; saved ret addr
```

## FILENAMES

A filename consists of a drive specification followed by a pathname and a simple filename. A drive specification consists of a single letter, A through P, followed by a colon; if the specification is missing, the default drive is used. A pathname consists of a list of simple filenames separated with backslashes. If the pathname starts with a backslash it is anchored in the root directory, otherwise it is anchored in the current directory. If the pathname is missing, the current directory is used. A simple filename consists of one to eight characters, optionally followed by a period and zero to three more characters.

Legal characters in filenames and pathnames include the alphabet (A-Z), digits (0-9), and most punctuation. Periods, colons, backslashes, slashes, question-marks, asterisks, control characters (including NULs), and characters greater than 0x7f may never appear in filenames. Lowercase letters are converted to uppercase.

A full file specification may not exceed 125 characters.

```
 _____
| Legal Characters in Filenames |
|       letters A-Z, a-z        |
|         numbers 0-9           |
|       _ (underscore)          |
|    ! @ # $ % ^ & ( )          |
|    + - = ~ ` ; ' " ,          |
|        < > | [ ] { }          |
|_____|
```

In a pathname, "." refers to the current directory and ".." refers to the current directory's parent directory. Thus, the paths:

                    "..\..\foo"
and
        ".\.\.\.\.\.\..\.\.\..\.\foo"

refer to the same file two directories up from the current one. (There is no parent directory at the root.)

There are three character devices. Only the calls Fread(), Fwrite() Fopen(), Fcreate(), and Fclose(), and the standard I/O functions work on them:

| name | handle | device |
|---|---|---|
| CON:, con: | 0x0ffff (-1) | system console |
| AUX:, aux: | 0x0fffe (-2) | RS232 port |
| PRN:, prn: | 0x0fffd (-3) | printer port |

An Fopen() or Fcreate() call on one of the character devices will return a character device handle. The handle is WORD negative, but not LONG negative.

## FILE OPERATIONS

GEMDOS places no restrictions on what a file may contain. Most applications assume that text files contain lines separated with carriage-return linefeeds, with a control-Z indicating the end of file. The format of executable files is documented in the Appendix.

The GEMDOS calls Fcreate() and Fopen() return small, positive 16-bit integers, called handles, that refer to open files. A file may be opened for reading only, for writing only, or for reading and writing. Closing the file relinquishes the handle, allowing the handle to be re-used.

There are three kinds of handles. Standard handles range from 0 to 5, and may refer to character devices or files. Non-standard handles start at 6, and refer only to files. Character handles refer only to character devices; the handle numbers range from Oxfffd to Oxffff. which are WORD negative, but not LONG negative.

When a process does a Pexec() call the child process inherits the parent's standard handles. Handle 0 is often referred to as "standard input" or "standard output"; normally it is connected to the console, CON:. With Fdup() and Fforce() calls it is possible to redirect a process's standard I/O to or from a file or another character device.

When a media change occurs, all files open on the disk that was removed are forced closed by GEMDOS.

## BUGS

There is no concept of "standard error" output.

PROCESSES
        Although GEMDOS does not support multitasking, it is
possible to execute processes in a subroutine-like manner.
A process may "call" another with Pexec(); the child process
will terminate with a WORD return code.

        A process owns any files it opens and any memory it
allocates.  Open files are closed and memory is deallocated
when the process terminates.

        Before a process is actually terminated GEMDOS will
call extended vector 0x102.  This allows applications to
make a "last ditch" effort to recover from error conditions,
or to deinstall themselves.

        The memory model used by GEMDOS is similar to MSDOS's.
A process runs in the TPA (Transient Program Area).  The
first 0x100 bytes of the TPA is the process's basepage,
which contains process-specific information.

                    Basepage Structure

| offset | name | description |
|--------|------|-------------|
| 0x00 | p_lowtpa | -> base of TPA |
| 0x04 | p_hitpa | -> end of TPA |
| 0x08 | p_tbase | base of text segment |
| 0x0c | p_tlen | size of text segment |
| 0x10 | p_dbase | base of data segment |
| 0x14 | p_dlen | size of data segment |
| 0x18 | p_bbase | size of BSS segment |
| 0x1c | p_blen | base of BSS segment |
| 0x20 | p_dta | Disk Transfer Address (DTA) |
| 0x24 | p_parent | -> parent's basepage |
| 0x28 | (reserved) | |
| 0x2c | p_env | -> enviroment string |
| 0x80 | p_cmdlin | commandline image |

        `p_lowtpa' points to the basepage (to itself).
`p_hitpa' points to the TPA's limit, to the first unusable
location.  `p_tbase', `p_tlen' and so on contain the start-
ing addresses and sizes of the text, data and BSS segments.
`p_parent' points to the process's parent process's
basepage.  `p_env' points to the enviroment string [see
Pexec()].

        The first byte of the commandline image contains the
number of characters in the commandline.  The second through
Nth bytes contain the image.  The image is not guaranteed to
be null-terminated.

An application receives control at the starting address
of its text segment. The second longword on the stack,
4(sp), will contain a pointer to the process's basepage.
Normally all free memory is allocated to a new process; if
the process is going to use Malloc() or Pexec() then it must
relocate its stack and call Mshrink() to release memory back
to the system. The stack segment starts near the highest
TPA location and grows toward the BSS.

## EXTENDED VECTORS

The 68000 uses vectors 0x02 through 0xff, corresponding to absolute locations 0x0000 through 0x03fc. GEMDOS adds eight _logical_ vectors, numbered 0x100 through 0x107. The absolute locations of the logical vectors is undefined; it is up to the BIOS to allocate storage for them.

### Logical Vector Assignments

| vector | use |
|---|---|
| 0x100 | timer tick |
| 0x101 | critical error handler |
| 0x102 | terminate (^C) handler |
| 0x103 - 0x107 | reserved for future use |

### 0x100 Timer Tick

This vector is called periodically (at 50hz) by the BIOS to maintain the system's date/time-of-day clock and do housekeeping. The first word on the stack, 4(sp), contains the number of milliseconds from the last timer tick interrupt.

To intercept the timer vector, use the BIOS call to get and set the vector. Each handler should execute its own code first, and then follow the old vector. Interrupt handlers should be short and sweet; dawdling here will affect system performance.

All registers (except SP and USP) are modified by GEMDOS. The BIOS takes responsibility for saving registers D0-D7/A0-A6; therefore handlers chained to this interrupt do not have to save and restore registers.

### 0x101 Critical Error Handler

The Critical Error Handler is called by the BIOS to handle certain errors (rwabs() disk errors and media change requests.) It allows the application to handle the errors as it sees fit.

The first word on the stack, 4(sp), is an error number. Depending on the error, other arguments may also be on the stack. The critical error handler should preserve registers D3-D7/A3-A6. When the handler returns, D0 contains a result code:

| value in DO.L | meaning |
|---|---|
| 0x00010000 | retry |
| 0x00000000 | pretend there wasn't an error (ignore) |
| 0xffffffXX | abort with an error |

       The default critical error handler simply  returns
   -1.

0x102 Terminate (^C) Handler
       Before a process is  actually  terminated,  GEMDOS
   calls  the  term'nate  vector.  If the terminate vector
   points to an RTS (the default case), the  process  will
   be  terminated.  If the application does not wish to be
   terminated it should do a longjump (or its  equivalent)
   to an appropriate handler.

## ERROR NUMBERS

All error numbers are negative.  Two ranges  of  errors  are
defined;  BIOS errors range from -1 to -31 and GEMDOS errors
range from -32 to -127.

### BIOS Error Codes

| name | number | description |
|------|--------|-------------|
| E_OK | 0 | OK (no error) |
| ERROR | -1 | Error |
| EDRVNR | -2 | Drive not ready |
| EUNCMD | -3 | Unknown command |
| E_CRC | -4 | CRC error |
| EBADRQ | -5 | Bad request |
| E_SEEK | -6 | Seek error |
| EMEDIA | -7 | Unknown media |
| ESECNF | -8 | Sector not found |
| EPAPER | -9 | Out of paper |
| EWRITF | -10 | Write fault |
| EREADF | -11 | Read fault |
|  | -12 | (unused) |
| EWRPRO | -13 | Write on write-protected media |
| E_CHNG | -14 | Media change detected |
| EUNDEV | -15 | Unknown device |
| EBADSF | -16 | Bad sectors on format |
| EOTHER | -17 | Insert other disk (request) |

`EOTHER' is really a request from the  BIOS  to  insert
another  disk  in drive A:.  The "virtual" disk number (0 or
1) is at 6(sp).  This feature is used to  fake  GEMDOS  into
thinking that a single drive system really has two drives.

GEMDOS Error Codes
(numbers in parenthesis
are MSDOS-equivalent error#s)

| name | number | description |
|------|--------|-------------|
| EINVFN | -32 (1) | Invalid function number |
| EFILNF | -33 (2) | File not found |
| EPTHNF | -34 (3) | Path not found |
| ENHNDL | -35 (4) | Handle pool exhausted |
| EACCDN | -36 (5) | Access denied |
| EIHNDL | -37 (6) | Invalid handle |
| ENSMEM | -39 (8) | Insufficient memory |
| EIMBA | -40 (9) | Invalid memory block address |
| EDRIVE | -46 (15) | Invalid drive specification |
| ENMFIL | -47 (18) | No more files |
| ERANGE | -64 | Range error |
| EINTRN | -65 | GEMDOS internal error |
| EPLFMT | -66 | Invalid executable file format |
| EGSBF | -67 | Memory block growth failure |

```
GEMDOS FUNCTIONS BY NUMBER
0x00 Pterm0 - Terminate Process
0x01 Cconin - Read character from Standard Input
0x02 Cconout - Write Character to Standard Output
0x03 Cauxin - Read Character from Standard AUX:
0x04 Cauxout - Write Character to Standard AUX:
0x05 Cprnout - Write Character to Standard PRN:
0x06 Crawio - Raw I/O to Standard Input/Output
0x07 Crawcin - Raw Input from Standard Input
0x08 Cnecin - Read Character from Standard Input, No Echo
0x09 Cconws - Write String to Standard Output
0x0A Cconrs - Read Edited String from Standard Input
0x0B Cconis - Check Status of Standard Input
0x0E Dsetdrv - Set Default Drive
0x10 Cconos - Check Status of Standard Output
0x11 Cprnos - Check Status of Standard PRN:
0x12 Cauxis - Check Status of Standard AUX: Input
0x13 Cauxos - Check Status of Standard AUX: Output
0x19 Dgetdrv - Get Default Drive
0x1A Fsetdta - Set DTA (Disk Transfer Address)
0x20 Super - Get/Set/Inquire Supervisor Mode
0x2A Tgetdate - Get Date
0x2B Tsetdate - Set Date
0x2C Tgettime - Get Time
0x2D Tsettime - Set Time
0x2F Fgetdta - Get DTA (Disk Transfer Address)
0x30 Sversion - Get Version Number
0x31 Ptermres - Terminate and Stay Resident
0x36 Dfree - Get Drive Free Space
0x39 Dcreate - Create Directory
0x3A Ddelete - Delete Directory
0x3B Dsetpath - Set Current Directory
0x3C Fcreate - Create File
0x3D Fopen - Open File
0x3E Fclose - Close File
0x3F Fread - Read From File
0x40 Fwrite - Write To File
0x41 Fdelete - Delete File
0x42 Fseek - Seek File Pointer
0x43 Fattrib - Get/Set File Attributes
0x45 Fdup - Duplicate File Handle
0x46 Fforce - Force File Handle
0x47 Dgetpath - Get Current Directory
0x48 Malloc - Allocate Memory
0x49 Mfree - Release Memory
0x4A Mshrink - Shrink Size of Allocated Block
0x4B Pexec - Load/Execute Process
0x4C Pterm - Terminate Process
0x4E Fsfirst - Search First
0x4F Fsnext - Search Next
0x56 Frename - Rename File
0x57 Fdatime - Get/Set File Timestamp
```

GEMDOS FUNCTIONS BY NAME
0x03 Cauxin - Read Character from Standard AUX:
0x12 Cauxis - Check Status of Standard AUX: Input
0x13 Cauxos - Check Status of Standard AUX: Output
0x04 Cauxout - Write Character to Standard AUX:
0x01 Cconin - Read character from Standard Input
0x0B Cconis - Check Status of Standard Input
0x10 Cconos - Check Status of Standard Output
0x02 Cconout - Write Character to Standard Output
0x0A Cconrs - Read Edited String from Standard Input
0x09 Cconws - Write String to Standard Output
0x08 Cnecin - Read Character from Standard Input, No Echo
0x11 Cprnos - Check Status of Standard PRN:
0x05 Cprnout - Write Character to Standard PRN:
0x07 Crawcin - Raw Input from Standard Input
0x06 Crawio - Raw I/O to Standard Input/Output
0x39 Dcreate - Create Directory
0x3A Ddelete - Delete Directory
0x36 Dfree - Get Drive Free Space
0x19 Dgetdrv - Get Default Drive
0x47 Dgetpath - Get Current Directory
0x0E Dsetdrv - Set Default Drive
0x3B Dsetpath - Set Current Directory
0x43 Fattrib - Get/Set File Attributes
0x3E Fclose - Close File
0x3C Fcreate - Create File
0x57 Fdatime - Get/Set File Timestamp
0x41 Fdelete - Delete File
0x45 Fdup - Duplicate File Handle
0x46 Fforce - Force File Handle.
0x2F Fgetdta - Get DTA (Disk Transfer Address)
0x3D Fopen - Open File
0x3F Fread - Read From File
0x56 Frename - Rename File
0x42 Fseek - Seek File Pointer
0x1A Fsetdta - Set DTA (Disk Transfer Address)
0x4E Fsfirst - Search First
0x4F Fsnext - Search Next
0x40 Fwrite - Write To File
0x48 Malloc - Allocate Memory
0x49 Mfree - Release Memory
0x4A Mshrink - Shrink Size of Allocated Block
0x4B Pexec - Load/Execute Process
0x4C Pterm - Terminate Process
0x00 Pterm0 - Terminate Process
0x31 Ptermres - Terminate and Stay Resident
0x20 Super - Get/Set/Inquire Supervisor Mode
0x30 Sversion - Get Version Number
0x2A Tgetdate - Get Date
0x2C Tgettime - Get Time
0x2B Tsetdate - Set Date
0x2D Tsettime - Set Time

```
 _____
| 0x00 Pterm0 - Terminate Process |
|_____|
```

void Pterm0()

     Terminate this process, closing all files it opened and releasing any memory it allocated. Return an exit code of 0x0000 to the parent process.

```
 _____
| 0x01 Cconin - Read character from Standard Input |
|_____|
```

LONG Cconin()

     **Read character** from the standard input (handle 0). If the standard input device is the console, the longword returned in D0 contains both the ASCII and the console scancode:

| 31..24 | 23..16 | 15..8 | 7..0 |
|--------|--------|-------|------|
| 0x00 or shift bits | scancode or 0x00 | ·0x00 | ASCII char |

     The function keys (F1 through F10, HELP, UNDO, etc.) return the ASCII code 0x00, with appropriate scancode values; see the GEM/VDI manual for keyboard scancode assignments. The ST BIOS is capable of placing the keyboard shift-key status in bits 24..31; see the BIOS Programmer's Guide for further details.

BUGS
Does not return any indication of end of file.
Control-C is not recognized.
There is no way to tell if standard input is a character device or a file.
There should be some way to type all possible 256 codes from the keyboard.

---
| 0x02 Cconout - Write Character to Standard Output |

        void Cconout(c)
        WORD c;

        Write the character `c' to the standard output  (handle
0).   The  high  eight  bits of `c' are reserved and must be
zero.   Tabs are not expanded.

---
| 0x03 Cauxin - Read Character from Standard AUX: |

        WORD Cauxin()

        Read character from handle 1 (normally the serial port,
AUX:).

BUGS
This function causes RS232 flow-control  to  fail;  applica-
tions  should  use  the BIOS character device calls to avoid
losing received characters.

---
| 0x04 Cauxout - Write Character to Standard AUX: |

        void Cauxout(c)
        WORD c;

        Write `c' to standard  handle  1  (normally  AUX:,  the
serial  port).   The high eight bits of `c' are reserved and
must be zero.  Tabs are not expanded.

BUGS
        This function causes RS232 flow-control to fail; appli-
cations  should use the BIOS character device calls to avoid
losing transmitted characters.

---
```
|0x05 Cprnout - Write Character to Standard PRN:|
```
---

        void Cprnout(c)
        WORD c;

        Write `c' to  handle  2  (normally  PRN:,  the  printer
port).   The high eight bits of `c' are reserved and must be
zero.  Tabs are not expanded


---
```
|0x06 Crawio - Raw I/O to Standard Input/Output|
```
---

        LONG Crawio(w)
        WORD w;

        If `w' is not 0x00FF, write it to the standard  output.
Tabs are not expanded

        Otherwise, if` `w' equals 0x00ff, read a character from
the  standard  input.   0x0000 is returned if no character is
available.

BUGS
Because of the way this function is defined,  `0xff'  cannot
be written to the standard output with this function.
Cannot distinguish between 0x00 and the end of the file.


---
```
|0x07 Crawcin - Raw Input from Standard Input|
```
---

        LONG Crawcin()

        Read a character from the standard  input  (handle  0).
If  the input device is CON: no control character processing
is done and the character is not echoed.

BUGS
No end of file indication.

---
| 0x08 Cnecin – Read Character from Standard Input, No Echo |
---

LONG Cnecin()

Read character from the standard input. If the input device is CON:, no echoing is done, although control characters are interpreted.

---
| 0x09 Cconws – Write String to Standard Output |
---

void Cconws(str)
char *str;

Write a null-terminated string. starting at `str`. to the standard output.

---
| 0x0A Cconrs – Read Edited String from Standard Input |
---

void Cconrs(buf)
char *buf;

Read string from the standard input, handling common line editing characters. The editing characters are:

| Char | Function |
|---|---|
| <return>, ^J | End the line |
| ^H, <rub> | Kill last character |
| ^U, ^X | Kill entire line |
| ^R | Retype line |
| ^C | Terminate the process |

The first character of `buf' indicates the size of the data part of the buffer. On return, the second byte of `buf' is set to the number of characters read, and locations `buf+2' through 'buf+2+buf[1]' contain the characters.

The string is **not** guaranteed to be null-terminated.

BUGS
Hangs on end-of-file.

---
| 0x0B Cconis - Check Status of Standard Input |
---

    WORD Cconis()

        Return 0xFFFF if a character is available on the  stan-
    dard input, 0x0000 otherwise.


---
| 0x0E Dsetdrv - Set Default Drive |
---

    LONG Dsetdrv(drv)
    WORD drv;

        Set the default drive to the  zero-based  drive  number
    `drv' (ranging from 0 to 15, A: to P:).  Return a bit-string
    of known drives (bit 0 = A, bit 1 = B, etc.)

        A "known drive" is one on which a  directory  has  been
    used.

    BUGS
        GEMDOS only supports 16 drives  (bits  0  through  15).
    Future systems will support 32 drives.


---
| 0x10 Cconos - Check Status of Standard Output |
---

    WORD Cconos()

        Return 0xFFFF if the console  is  ready  to  receive  a
    character.  Return 0x0000 if the console is NOT ready.

    BUGS
    CON: and files are always ready, so why check?


---
| 0x11 Cprnos - Check Status of Standard PRN: |
---

    WORD Cprnos()

        Return 0xFFFF if PRN: is ready to receive a  character,
    0x0000 if it isn't.

---

| 0x12 Cauxis - Check Status of Standard AUX: Input |

WORD Cauxis()

     Return 0xFFFF if a character is available on AUX: (handle 1), 0x0000 if not.

---

| 0x13 Cauxos - Check Status of Standard AUX: Output |

WORD Cauxos()

     Return 0xFFFF if AUX: (standard handle 1) is ready to accept a character, 0x0000 if not.

---

| 0x19 Dgetdrv - Get Default Drive |

WORD Dgetdrv()

     Return the current drive number, 0 through 15.

---

| 0x1A Fsetdta - Set DTA (Disk Transfer Address) |

void Fsetdta(addr)
char *addr;

     Set the DTA to `addr'.  (The DTA is used only by the functions Fsfirst() and Fsnext().)

---

| 0x20 Super - Get/Set/Inquire Supervisor Mode |

LONG Super(stack)
WORD *stack;

If `stack' is -1L (0xFFFFFFFF)  return  0x0000  if  the
processor  is in user mode, or 0x0001 if the processor is in
supervisor mode.

Otherwise, if the processor is  in  user  mode,  return
with  the  processor switched to supervisor mode.  If `stack'
is NULL (0x00000000) then the supervisor stack will  be  the
same  as  the user stack before the call.  Otherwise the su-
pervisor stack will be set to `stack'.

If the processor is in supervisor mode, return with the
processor switched back to user mode.  `stack' should be the
value of the supervisor stack that was returned by the first
call to the function.

NOTE
        The original supervisor stack value  MUST  be  re-
        stored before the process terminates.  Failure to do so
        will result in a system crash.

---

| 0x2A Tgetdate - Get Date |

WORD Tgetdate()

Return the current date, in DOS format:
```
15                      9  8           5  4           0
```

| year since 1980<br>0..119 | month<br>1..12 | day<br>1..31 |
|---|---|---|

RETURNS
        Bits:
        0..4 contain the day, ranging 1..31.
        5..8 contain the month ranging 1..12.
        9..15 contain the year (since 1980) ranging 0..119.

---

### 0x2B Tsetdate - Set Date

---

```
WORD Tsetdate(date)
WORD date;
```

Set the current date to `date', which is in the  format
described in Tgetdate().

**RETURNS**

    0 on valid date;
    ERROR on an obviously screwed-up date.

**BUGS**

    GEMDOS is not picky  about  date  parameters;  for  in-
stance, it likes Feb 31st ....

    GEMDOS does NOT let the BIOS know  that  the  date  has
been changed.

---

### 0x2C Tgettime - Get Time

---

```
WORD Tgettime()
```

Return  the  current  time  in  DOS  format:

| 15 | 11 | 10 | 5 | 4 | 0 |
|---|---|---|---|---|---|

| hour 0..23 | minute 0..59 | second 0..29 |
|---|---|---|

**RETURNS**

    Bits 0..4 contain the second divided by 2, 0..29.
    Bits 5..10 contain the minute, 0..59.
    Bits 11..15 contain the hour, 0..23.

---

```
¦0x2D Tsettime - Set Time¦
```

    WORD Tsettime(time)
    WORD time;

         Set the current time to `time', which is in the  format
    described in Tgettime().

    RETURNS
         0 if GEMDOS liked the time;
         ERROR if it didn't.

    BUGS
         GEMDOS does NOT let the BIOS know  that  the  time  has
    been changed.

---

```
¦0x2F Fgetdta - Get DTA (Disk Transfer Address)¦
```

    LONG Fgetdta()

         Returns the value of the current DTA, a pointer used by
    the functions Fsfirst() and Fsnext().

---

```
¦0x30 Sversion - Get Version Number¦
```

    WORD Sversion()

         Return GEMDOS's version number (in  byte-reversed  for-
    mat).   The high byte contains the minor version number, the
    low byte contains the major version number.

    NOTE
         The 5/29/85 (first disk-based) and the 11/20/85  (first
    ROM-based) release of GEMDOS had the version number 0x1300.

    GEMDOS version numbers and TOS versions numbers are not  one
    and  the  same.   See the ST BIOS REFERENCE MANUAL for about
    TOS version numbers.

---

| 0x31 Ptermres - Terminate and Stay Resident |
|---|

```
    void Ptermres(keepcnt, retcode)
    LONG keepcnt;
    WORD retcode;
```

Terminate the current process, keeping some of it in memory. `keepcnt' is the amount of the memory belonging to the process to keep, including and starting at the 256-byte basepage. `retcode' is the exit code that is returned to the parent process.

Memory the process has allocated (in addition to the TPA) will NOT be released.

Ptermres() will never return.

BUGS
Open files are closed as part of termination.

---

| 0x36 Dfree - Get Drive Free Space |
|---|

```
    void Dfree()
    LONG *buf;
    WORD driveno;
```

Get disk allocation information about the drive `driveno' and store it into four longwords starting at `buf':

| | |
|---|---|
| buf + 0 | # of free clusters |
| buf + 4 | total # of clusters |
| buf + 8 | sector size (in bytes) |
| buf + 12 | cluster size (in sectors) |

BUGS
Incredibly slow (5-10 seconds) on a hard disk.

```
┌─────────────────────────────────────┐
│0x39 Dcreate - Create Directory│
└─────────────────────────────────────┘
```

        WORD Dcreate(pathname)
        char *pathname;

        Create a directory.  `pathname' points to a null-
    terminated  string specifying the pathname of the new direc-
    tory.

        RETURNS
            0 on success;
            ERROR or appropriate error number on failure.

```
┌─────────────────────────────────────┐
│0x3A Ddelete - Delete Directory│
└─────────────────────────────────────┘
```

        WORD Ddelete(pathname)
        char *pathname;

        Delete a directory (it must be empty,  except  for  the
    special  directories  "." and "..").  `pathname' points to a
    null-terminated string specifying the pathname of the direc-
    tory to remove.

        RETURNS
            0 on success;
            ERROR or appropriate error number on failure.

```
┌──────────────────────────────────────────┐
│0x3B Dsetpath - Set Current Directory│
└──────────────────────────────────────────┘
```

        WORD Dsetpath(path)
        char *path;

        Set the current to `path',  a  null-terminated  string.
    If  the path begins with a drive letter and a colon, set the
    current directory on the specified drive.

        A current directory is kept for each drive in the  sys-
    tem.

        RETURNS
            0 for success;
            ERROR or an appropriate error number.

---
|0x3C Fcreate - Create File|
---

```
     WORD Fcreate(fname, attribs)
     char *fname;
     WORD attribs;
```

    Create a file `fname' and return a write-only non-standard handle to it. The attribute word is stored in the directory entry; its bit assignments are:

| mask | description |
|------|-------------|
| 0x01 | file set to read-only |
| 0x02 | file hidden from directory search |
| 0x04 | file set to "system" |
| 0x08 | file contains 11-byte volume label |

RETURNS
    a positive number, a handle, or:
    ERROR or an appropriate error number.

BUGS
    Useless feature department: If the `read-only' bit is set, a write-only handle is returned, and the handle can't be written to.

    Ideally, only one volume label is permitted in the volume's root directory. GEMDOS doesn't enforce this, though, which could cause confusion.

---
| 0x3D Fopen − Open File |
---

```
WORD Fopen(fname, mode)
char *fname;
WORD mode;
```

Open the `fname' according to `mode', and return a
non-standard handle to it.  The open mode can be:

| mode | description |
|------|-------------|
| 0 | read only |
| 1 | write only |
| 2 | read or write |

RETURNS
a positive number, a handle, or:
a negative error number.

---
| 0x3E Fclose − Close File |
---

```
WORD Fclose(handle)
WORD handle;
```

Close the file associated with the handle.

RETURNS
0 on success;
ERROR or an appropriate error number.

---
| 0x3F Fread − Read From File |
---

```
LONG Fread(handle, count, buffer)
WORD handle;
LONG count;
char *buffer;
```

Read from a file.  From the file referred to  by  `han-
dle' read `count' bytes into memory starting at `buffer'.

RETURNS
the number of bytes actually read, or:
0 on end of file, or:
a negative error number.

---
|0x40 Fwrite - Write To File|
---

```
      LONG Fwrite(handle, count, buffer)
      WORD handle;
      LONG count;
      char *buffer;
```

          Write to a file.   Write `count' bytes  from  memory,
starting at `buffer', to the file referred to by `handle'.

RETURNS
          the number of bytes actually written, or:
          a negative error number.

---
|0x41 Fdelete - Delete File|
---

```
      WORD Fdelete(fname)
      char *fname;
```

          Delete the file `fname'.

RETURNS
          0, success, or:
          a negative error number.

---
| 0x42 Fseek - Seek File Pointer |
---

```
LONG Fseek(offset, handle, seekmode)
LONG offset;
WORD handle;
WORD seekmode;
```

Set the current position within the file associated with `handle`. `offset` is a signed number; positive values move toward the end of the file, and negative values move toward its beginning. `seekmode` can be:

| seekmode | Moves `offset' bytes ... |
|----------|--------------------------|
| 0 | from beginning of file |
| 1 | relative to current position |
| 2 | from end of file |

RETURNS
   The current, absolute position in the file.

---
| 0x43 Fattrib - Get/Set File Attributes |
---

```
WORD Fattrib(fname, wflag, attribs)
char *fname;
WORD wflag;
WORD attribs;
```

Get or set a file's attribute bits. `fname' points to a null-terminated pathname. If `wflag' is 1, set the file's attributes from `attribs' (no return value). If `wflag' is 0, return the file's attributes.

The attribute bits are:

| mask | description |
|------|-------------|
| 0x01 | file is read-only |
| 0x02 | file hidden from directory search |
| 0x04 | file set to "system" |
| 0x08 | file contains 11-byte volume label |
| 0x10 | file is a subdirectory |
| 0x20 | file has been written to and closed. |

BUGS
   The "archive" bit, 0x20, doesn't seem to work as advertised.

---
¦0x45 Fdup - Duplicate File Handle¦
---

WORD Fdup(handle)
WORD handle;

The handle `handle' must be a standard  handle  (0..5);
Fdup()  returns a non-standard handle (greater than or equal
to 6) that refers to the same file.

RETURNS
a handle, or:
EIHNDL - nⵟt a standard handle
ENHNDL - no more standard handles available

---
¦0x46 Fforce - Force File Handle¦
---

Fforce(stdh, nonstdh)
WORD stdh;
WORD nonstdh;

Force the standard handle `stdh' to point to  the  same
file or device as the non-standard handle `nonstdh.'

RETURNS
OK, or:
EIHNDL - ınvalid handle

---
¦0x47 Dgetpath - Get Current Directory¦
---

void Dgetpath(buf, driveno)
char *buf;
WORD driveno;

The current directory for the specified drive `driveno'
is copied into `buf'.  The drive number is 1-based: 0 speci-
fies the default drive. 1 specifies A:, and so on.

BUGS
The maximum size of a pathname is not  limited  by  the
system; it is up to the application to provide enough buffer
space.  128 bytes should be enough for 8 or 9 levels of sub-
directories.

```
|0x48 Malloc - Allocate Memory|
```

        LONG Malloc(amount)
        LONG amount;

        If `amount' is -1L ($FFFFFFFF) return the size  of  the
largest free block in the system.

        Otherwise, if `amount' is not -1L, attempt to  allocate
`amount' bytes for the current process.  Return a pointer to
the beginning of the block or NULL if there is no free block
large enough to meet the request.

BUGS
WARNING
        A process may not have,  at  any  time,  more  than  20
blocks of Malloc()'d memory.  Exceeding this limit may crip-
ple GEMDOS.  [It is OK to do many Malloc() calls if they are
followed  by  matching  Mfree() calls; the limit of 20 is to
the number of fragments a process may generate.]


```
|0x49 Mfree - Release Memory|
```

        WORD Mfree(saddr)
        LONG saddr;

        Free the block of memory starting at `saddr'; the block
must be one that was returned by Malloc().

RETURNS
        0 if the release was successful, or:
        ERROR or an appropriate error number.

---

| 0x4A Mshrink - Shrink Size of Allocated Block |
---

```
WORD Mshrink(0, block, newsiz)
(WORD) 0;
LONG block;
LONG newsiz;
```

Shrink the size of an allocated block of memory; `block' points to a process basepage or a piece of memory allocated by Malloc(), `newsiz' is the new size of the block.

The first argument must be a WORD of zero.


RETURNS
     0 if the size adjustment was successful, or:
     EIMBA - invalid memory block address
     EGSBF - setblock failure due to growth restrictions

BUGS
     A block can only be shrunk; `newsiz' must be less than or equal to the current block size.

---
| 0x4B Pexec - Load/Execute Process |
---

```
WORD Pexec(mode, ptr1, ptr2, ptr3)
WORD mode;
char *ptr1;
char *ptr2;
char *ptr3;
```

This function wears several hats, according to the flag `mode':

| mode | ptr1 | ptr2 | ptr3 |
|---|---|---|---|
| 0 = load & go | file to exec | command tail | enviroment string |
| 3 = load, no go | file to load | command tail | enviroment string |
| 4 = just go | basepage address | (unused) | (unused) |
| 5 = create basepage | (unused) | command tail | enviroment string |

The file to load or exec, `ptr1', and the command tail, `ptr2', are null-terminated pathnames. The enviroment string, `ptr3', is either NULL (0L), or a pointer to a string structure of the form:

```
"string1\0"
"string2\0"
... etc. ...
"stringN\0"
    "\0"
```

The enviroment string is any number of null-terminated strings, with an empty string (a single null) at the end. If `ptr3' is NULL, then the process inherits a copy of the parent's enviroment string.

Load-and-go (mode 0) will load the specified file, set-up its basepage, and execute it. Pexec()'s return value will be the child process's exit code (see Pterm0() and Pterm()).

Load-nogo will load the specified file, setup its basepage, and return a pointer to the basepage; the process is not executed.

Just-go is passed a pointer to a basepage. The process

starts   executing at the base of its text segment, as speci-
fied in the basepage.

Create-basepage will allocate the largest free block of
memory and create most of a basepage for it.  (Some entries,
most significantly the text/data/bss size and  base  values,
are  NOT  setup -- the caller is responsible for maintaining
them).

A child process inherits  the  parent's  standard  file
descriptors;  effectively  doing  an  Fdup() and an Fforce()
call on handles 0 through 5.

Since system resources are allocated when a basepage is
created,  the spawned process MUST be terminated in order to
release them.  This is especially important when using over-
lays;  see  the  [Pexec cookbook] for details on use of Pex-
ec().

---

| 0x4C Pterm - Terminate Process |

```
void Pterm(retcode)
WORD retcode;
```

Terminate the current process, closing all  open  files
and releasing any allocated memory.  Return `retcode' to the
parent process.

```
|Ox4E Fsfirst - Search First|
```

```
WORD Fsfirst(fspec, attribs)
char *fspec;
WORD attribs;
```

Search for the first occurrence of the file `fspec`.
The file specification may contain wildcards (`?' and `*')
in the simple filename, but not in the path specification.
`attrib' controls which files are returned by Fsfirst; its
format is described in the documentation on `Fattrib()'.

If `attrib' is zero, then only normal files are
searched for (no volume labels, hidden files, subdirectories
or system files are returned). If `attrib' is set for hid-
den or system files, they are included in the search set.
If `attrib' is set for volume labels, only volume labels are
returned.

When a file is found, a 44-byte structure is written to
the location pointed to by the DTA:

| offset | size | contents |
|--------|------|----------|
| 0-20 | | (reserved) |
| 21 | byte | file attribute bits |
| 22 | word | time stamp |
| 24 | word | date stamp |
| 26 | long | file size |
| 30 | 14 bytes | file name + extension |

The filename and extension is null-terminated, and con-
tains no spaces.

RETURNS
    0, if a file was found, or:
    EFILNF - file not found (no matches), or:
    an appropriate error number.

```
|0x4F Fsnext - Search Next|
```

        WORD Fsnext()

        Search for the next occurrence of a file.  (The first
occurrence  should  be  searched for with Fsfirst()).  Bytes
0-20 of the DTA must remain unmodified  from  the  Fsfirst()
call or the most recent Fsnext() call.

        RETURNS
                0 if a file was found, or:
                ENMFIL - no more files were found, or:
                an appropriate error number.

```
|0x56 Frename - Rename File|
```

        WORD Frename(0, oldname, newname)
        (WORD) 0;
        char *oldname;
        char *newname;

        Rename a file from `oldname' to `newname'.  The desti-
nation  file must not exist.  The new file may be in another
directory.

        The first argument must be a zero WORD.

        RETURNS
                EACCDN - destination file already exists;
                EPTHNF - `oldname' not found;
                ENSAME - `newname' not on save drive;
                or an appropriate error.

---

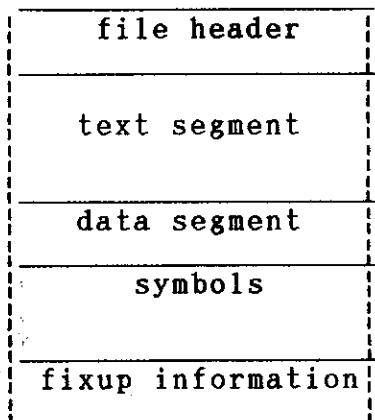| 0x57 Fdatime - Get/Set File Timestamp |

---

```
void Fdatime(handle, timeptr, wflag)
WORD handle;
LONG timeptr;
WORD wflag;
```

   The file is referred to by `handle`.  `timeptr`  points
to  two  words  containing  the DOS formatted timestamp (the
time word is first, the date word is second).  If `wflag` is
1,  set  the file's timestamp from `timeptr`, otherwise read
the file's timestamp into `timeptr`.

EXECUTABLE FILES
      An executable file consists of a header followed by
images for the text and data segments, zero or more symbol
table entries, a fixup offset, and zero or more fixup
records:

                    Executable File Parts


                ┌──────────────────────────┐
                │      file header         │
                ├──────────────────────────┤
                │                          │
                │      text segment        │
                │                          │
                ├──────────────────────────┤
                │      data segment        │
                ├──────────────────────────┤
                │        symbols           │
                │                          │
                ├──────────────────────────┤
                │   fixup information      │
                └──────────────────────────┘


      The file header contains a "magic" number (a  signature
to indicate that it is an executable file) and several long-
words containing size information:

                    Executable File Header

| Offset | Size | Description |
|--------|------|-------------|
| 0x00   | word | 0x601A (magic number) |
| 0x02   | long | Size of text segment |
| 0x06   | long | Size of data segment |
| 0x0A   | long | Size of BSS segment |
| 0x0E   | long | Size of symbol table |
| 0x12   | long | (reserved) |
| 0x16   | long | (reserved) |
| 0x1A   | long | (reserved) |
| 0x1E   |      | (start of text segment) |


      The text and data segment images immediately follow the
header.  The symbol table, if there is one, follows the data
segment.

      GEMDOS will "fix up" a longword in  the  text  or  data
segments by adding the base of the text segment to the value
already in the longword.  The  fixup  list  specifies  which
longwords need to be relocated.  The first item in the fixup
list is a longword specifying the offset of the first fixup;

the longword is NULL (OL) if there are no fixups. Single
bytes following the longword specify offsets to more fixups.
The longwords must start on word boundaries, or the system
will crash.

### Relocation Bytes
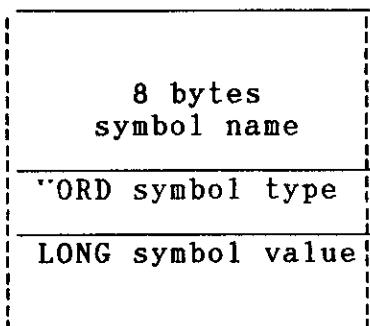
| Byte | Description |
|------|-------------|
| 0 | end of relocation information |
| 1 | advance 254 bytes, get next byte |
| 2, 4, .. 254 | fixup longword at location pointer |
| 3, 5, .. 255 | (odd numbers, reserved for future use) |

## SYMBOL TABLE

The symbol table consists of symbol-table entries, for-
matted as:

Symbol Table Entry

```
 _____
|                   |
|                   |
|      8 bytes      |
|    symbol name    |
|                   |
|_____|
| "ORD  symbol  type|
|_____|
| LONG  symbol value|
|                   |
|                   |
|_____|
```

<<<explain about symbol types here.  It's really pretty sim-
ple...>>>

Values for Symbol Types

| Type | Value |
|---|---|
| defined | 0x8000 |
| equated | 0x4000 |
| global | 0x2000 |
| equated register | 0x1000 |
| external reference | 0x0800 |
| data based relocatable | 0x0400 |
| text based relocatable | 0x0200 |
| BSS based relocatable | 0x0100 |

## VOLUME ORGANIZATION

    GEMDOS uses the first few sectors of a disk to indicate
where files are stored.   A volume usually contains five
parts; an optional boot sector, two identical FAT tables,  a
root directory, and a cluster area.

    When GEMDOS first accesses a  drive  (or  accesses  one
after a media change), it makes a `GETBPB' (Get BIOS Parame-
ter Block) BIOS call to determine how big these  areas  are,
and  where  they  are  stored on the disk.  GETBPB returns a
pointer to a nine-word structure.  From this structure, GEM-
DOS  can puzzle out where the various parts of the file sys-
tem are.

                   BIOS  Parameter Block (BPB)


| name   | value | function |
|--------|-------|----------|
| recsiz | 512   | physical sector size in bytes |
| clsiz  | 2     | cluster size in sectors |
| clsizb | 1024  | cluster size in bytes |
| rdlen  |       | root directory length in sectors |
| fsiz   |       | FAT size, in sectors |
| fatrec |       | sector# of 1st sector of 2nd FAT |
| datrec |       | sector# of 1st data sector |
| numcl  |       | number of data clusters on disk |
| bflags |       | flags |


    RECSIZ indicates the number of bytes per physi-
cal  sector;   this must be 512 with the current GEM-
DOS.  CLSIZ indicates the number  of  sectors  in  a
cluster;  this  must  be  2  in  the current GEMDOS.
CLSIZB is the number of bytes in  a  cluster,  which
must be 1024.

    RDLEN is the size of  the  root  directory,  in
sectors.   A  directory  entry uses 32 bytes, so the
number of root files available is RDLEN * 512 / 32.

    FSIZ is the size of each FAT in  sectors.   FA-
TREC is the starting sector number of the first sec-
tor of the /second/ FAT.

    DATREC is the starting  sector#  of  the  first
cluster.   NUMCL  is  the  number of clusters on the
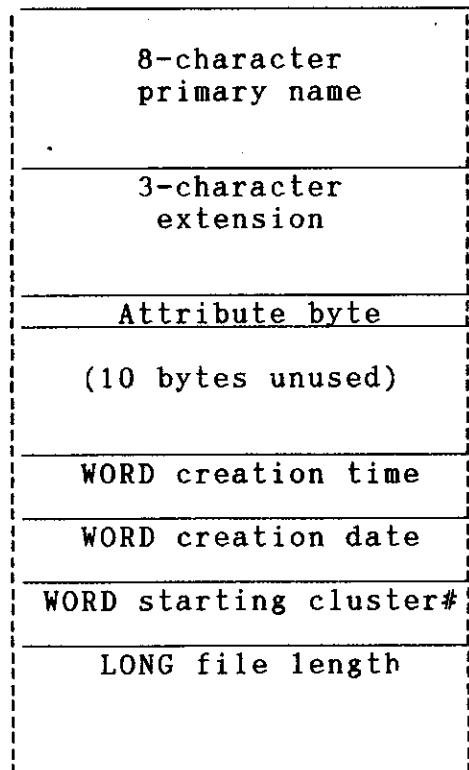device.

    BFLAGS was  supposed  to  be  a  bit-vector  of
flags.  Currently only bit 0 is being used; when set
it indicates that 16-bit  FAT  entries  (instead  of
12-bit ones) are to be used.

If there are boot sectors, they occupy logical  sectors
0  through  FATREC  -  FSIZ  -  1.  The second FAT starts at
FATREC, and the first FAT starts at FATREC - FSIZ.  The root
directory  starts  at  FATREC  + FSIZ, and the first cluster
starts at DATREC.  The cluster region is where the data  for
all files on the volume is kept.

DIRECTORY ENTRIES
     A directory entry contains a filename, some flags,  the
file's  creation  time  and  date,  the file's size, and the
file's starting cluster number.  The entry itself is  a  32-
byte structure that looks like:

Directory Entry

```
 _____
|                        |
|     8-character        |
|     primary name       |
|                        |
|_____|
|     3-character        |
|      extension         |
|                        |
|_____|
|     Attribute byte     |
|_____|
|   (10 bytes unused)    |
|                        |
|_____|
|   WORD creation time   |
|_____|
|   WORD creation date   |
|_____|
|  WORD starting cluster#|
|_____|
|    LONG file length    |
|                        |
|                        |
|_____|
```

     All WORDS and LONGS in the directory entry are in   8086
"byte reversed" format.

     When a file is deleted, the  first  byte  of  the  name
field is set to 0xe5.

A subdirectory is a file that contains directory entries. The first two entries in a subdirectory are always the special directories "." and "..".


FAT ENTRIES
The File Allocation Table (FAT) is used to allocate clusters and to link clusters together into files. FAT entries may be 12 or 16 bits. A file's directory entry contains the number of the first cluster in the file. Each cluster's associated FAT entry contains the number of the next cluster in the file, or a number that indicates end-of-file.

### 12-bit FAT Entries

| value | meaning |
|---|---|
| 0x000 | free cluster |
| 0x001 | (impossible) |
| 0x002 - 0xfef | next cluster number |
| 0xff0 - 0xff7 | bad sector |
| 0xff8 - 0xfff | end of file |

### 16-bit FAT Entries

| value | meaning |
|---|---|
| 0x0000 | free cluster |
| 0x0001 | (impossible) |
| 0x0002 - 0x7fff | next cluster number |
| 0x8000 - 0xffef | (impossible) |
| 0xfff0 - 0xfff7 | bad sector |
| 0xfff8 - 0xffff | end of file |

For a 12-bit FAT, obtain the next cluster in the file, NCL, given the current cluster number, CL, by:
    [1] (Multiply by 1.5)
        NCL = CL + CL / 2
    [2] Set NCL to the 16-bit word in the FAT indexed by NCL
        (it must be byte-swapped to 68000 format as well.)
        The word might not be on a 68000 word boundary.
    [3] (Extract the correct 12 bits.)
        If CL is odd, set NCL = NCL >> 4.

[4] (Mask off incorrect bits.)
    Set NCL = NCL & 0x0FFF.
[5] (Interpret the result.)
    If NCL is 0x0FF8 or higher, then  CL  was  the  last
    cluster  in the file.  If NCL is zero or in the range
    0x0FF0 to 0x0FF7 then there is a file system problem.
    Otherwise,  NCL  is the number of the next cluster in
    the file.

    For a 16-bit FAT, obtain the next cluster in the  file,
NCL, given the current cluster number, CL, by:
    [1] Set NCL to the 16-bit word in the FAT indexed by  CL.
        The word must ^e byte-swapped into 68000 format.
    [2] If NCL is 0xfff8 or higher,  then  CL  was  the  last
        cluster  in  the  file.   If NCL is 0 or in the range
        0x8000 to 0xfff7 then there is a file system problem.
        Otherwise,  NCL  is the number of the next cluster in
        the file.

    To convert from a cluster number, CL, to a logical sec-
tor number, LSN:
    [1] (Adjust for reserved FAT entries.)
        LSN = CL - 2
    [2] Multiply LSN by the number  of  sectors  per  cluster
        (CLSIZ).
    [3] Add the logical sector# of the first cluster  to  LSN
        (DATREC).