Amiga ROM Kernel Reference Manual

DOS

Thomas Richter

Contents

Chapter 1

Introduction

1.1 Purpose

The purpose of this manual is to provide a comprehensive documentation of the AmigaDOS subsystem of the Amiga Operation System. This subsystem is represented by the *dos.library*, and it provides services around files, file systems and stream-based input and output. While the Amiga ROM Kernel Reference Manuals [7] document major parts of the AmigaOs, they do not include a volume on AmigaDOS itself. This is due to the history of AmigaDOS which is nothing but a port of the TRIPOS to the Amiga, and thus its documentation became available as the AmigaDOS manual[1] separately. This book itself is, similar to AmigaDOS, based on the TRIPOS manual which has been augmented and updated to reflect the changes that were necessary to fit TRIPOS into AmigaOs. Unfortunately, the book is hard to obtain, and also leaves a lot to deserve.

Good third party documentation is available in the form of the Guru Book[10], though this source is out of print and even harder to obtain. It covers also other aspects of AmigaOs that go beyond AmigaDOS such that its focus is a bit different than this work.

This work attempts to fill this gap by providing a comprehensive and complete documentation of the AmigaDOS library and its subsystems in the style of the ROM Kernel Reference Manuals.

1.2 Language and Type Setting Conventions

The words *shall* and *shall not* indicate normative requirements software shall or shall not follow or in order to satisfy the interface requirements of AmigaOs. The words *should* and *should not* indicate best practise and recommendations that are advisable, but not strictly necessary to satisfy a particular interface. The word *may* provides a hint to a possible implementation strategy.

The word *must* indicates a logical consequence from existing requirements or conditions that follows necessarily without introducing a new restriction, such as in "if a is 2, $a + a$ *must* be 4".

> *Worth to remember!* Important aspects of the text are indicated with a bold vertical bar like this.

Terms are indicated in *italics*, e.g. the *dos.library* implements interface of *AmigaDOS*. Data structures and components of source code are printed `in courier` in fixed-width font, reassembling the output of a terminal, e.g.

```
typedef unsigned char UBYTE; /* an 8-bit unsigned integer */
typedef long LONG;           /* a 32-bit integer          */
```

Chapter 2

Elementary Data Types

2.1 The dos.library

AmigaDOS as part of the *Amiga Operating System* or short *AmigaOs* is represented by the ROM-based *dos.library*. This library is typically opened by the startup code of most compilers anyhow, and its base pointer is placed into `DOSBase` by this startup code:

```
struct DosLibrary *DOSBase;
```

Hence, in general, there is no need to open this library manually.

The structure *struct DosLibrary* is defined in `dos/dosextens.h`, but its layout and its members are usually not required and should rather not be accessed directly. Instead, the library provides accessor functions to read many objects contained within it.

If you do not link with compiler startup code, the base pointer of the *dos.library* can be obtained similar to that of any other library:

```
#include <proto/exec.h>
#include <proto/dos.h>
#include <exec/libraries.h>
#include <dos/dos.h>

...
if ((DOSBase = (struct DosLibrary *)(OpenLibrary(DOSNAME,47))) {
  ...
  CloseLibrary((struct Library *)DOSBase);
}
```

Unlike many other operating system, the *dos.library* does not manage disks or files itself, neither does it provide access to hardware interface components. It rather implements a *virtual file system* which forwards requests to its subsystems, called *handlers* or *file systems*, see 2.8.

2.2 Booleans

AmigaDOS uses a somewhat different convention for booleans, i.e. truth values defined in the file `dos/dos.h`:

Table 1: DOS Truth Values

Define	Value
DOSFALSE	0
DOSTRUE	-1

Note that the C language instead uses the value 1 for `TRUE`. Code that checks for zero or non-zero return codes will function normally, however code shall not compare to `TRUE` in boolean tests.

2.3 Pointers and BPTRs

AmigaDOS is a descendent of the *TRIPOS system* and as such originally implemented in the BCPL language. As of Kickstart 2.0, AmigaDOS was re-implemented in C and assembler, but this implementation had to preserve the existing interface based on BCPL conventions.

BCPL is a typeless language that structures the memory of its host system as an array of 32-bit elements enumerated contiguously from zero up. Rather than pointers, BCPL communicates the position of its data structures in the form of indices of the first 32-bit element of such structures. As each 32-bit group is assigned its own index, one can obtain this index by dividing the byte-address of an element by 4, or equivalently, by right-shifting the address by two bits. This has the consequence that (most) data structures passed into and out of the dos.library shall be aligned to 32-bit boundaries. Similarly, in order to obtain the byte-address of a BPCL structure, the index is multiplied by 4, or left-shifted by 2 bits.

> *Not on the Stack!* Since BPCL structures must have an address that is divisible by 4, you should not keep such structures on the stack as the average compiler will not ensure long word alignment for automatic objects. In the absense of a dedicated constructor function such as *AllocDosObject()*, a safe strategy is use the exec.library memory allocation functions such as *AllocMem()* or *AllocVec()* to obtain memory for holding them.

These indices are called *BCPL pointers* or short *BPTR*s, even though they are not pointers in the sense of the C language, but rather integer numbers as indices to an array of `LONG` (i.e. 32-bit) integers. In order to communicate this fact more clearly, the `dos/dos.h` include file defines the following data type:

```
typedef long  BPTR;                 /* Long word pointer */
```

Conversion from BCPL pointers to conventional C pointers and back are formed by the following macros, also defined in `dos/dos.h`:

```
/* Convert BPTR to typical C pointer */
#define BADDR(x)        ((APTR)((ULONG)(x) << 2))
/* Convert address into a BPTR */
#define MKBADDR(x)      (((LONG)(x)) >> 2)
```

Luckely, in most cases callers of the *dos.library* do not need to convert from and to BPTRs but can rather use such "pointers" as *opaque values* or *handles* representing some AmigaDOS objects.

It is certainly a burden to always allocate temporary BCPL objects from the heap, and doing so may also fragment the AmigaOs memory unnecessarily. However, allocation of automatic objects from the stack does not ensure long-word aligment in general. To work around this burden, one can use a trick and instead request from the compiler a somewhat longer object of automatic lifetime and align the requested object manually within the memory obtained this way. The following macro performs this trick:

```
#define D_S(type,name) char a_##name[sizeof(type)+3]; \
                       type *name = (type *)((ULONG)(a_##name+3) & ~3UL)
```

It is used as follows:

```
D_S (struct FileInfoBlock, fib);
```

At this point, fib is pointer to a properly aligned `struct FileInfoBlock`, e.g. this is equivalent to

```
struct FileInfoBlock _tmp;
struct FileInfoBlock *fib = &tmp;
```

except that the created pointer is properly aligned and can safely be passed into the *dos.library*.

Similar to the C language, a pointer to a non-existing element is expressed by the special pointer value 0. While this is called the `NULL` pointer in C, it is better to reserve another name for it in BPCL as its pointers are rather indices. The following convention is suggested to express an invalid *BPTR*:

```
#define ZERO 0L
```

Clearly, with the above convention, the BCPL `ZERO` pointer converts to the C `NULL` pointer and back, even though the two are conceptionally something different: The first being the index to the first element of the host memory array, the later the pointer to the first address.

2.4 C strings and BSTRs

While the C language defines *strings* as 0-terminated arrays of `char`, and AmigaOs in particular to 0-terminated arrays of `UBYTE`s, that is, unsigned characters, the BPCL language uses a different convention, namely that of a `UBYTE` array whose first element contains the size of the string to follow. They are not necessarily 0-terminated either. If BCPL strings are passed into BCPL functions, or are part of BCPL data structures, then typically in the form of a *BPTR* to the 32-bit element containing the size of the string its 8 most significant bits. The include file `dos/dos.h` provides its own data type for such strings:

```
typedef long  BSTR;                 /* Long word pointer to BCPL string  */
```

Luckely, functions of the *dos.library* take C strings as arguments and perform the conversion from C strings to their BCPL representation as *BSTR*s internally, such that one rarely gets in contact with this type of strings. They appear as part of some AmigaDOS structures to be discussed, and as part of the interface between the *dos.library* and its handlers, e.g. file systems. However, even though users of the *dos.library* rarely come in contact with *BSTR*s themselves, the *BCPL* convention has an important consequence, namely that (most) strings handled by the *dos.library* cannot be longer than 255 characters as this is the limit imposed by the BCPL convention.

> *Lengh-Limited Strings* Remember that most strings that are passed into the *dos.library* are internally converted to *BSTR*s and thus cannot exceed a length of 255 characters.

Unfortunately, even as of the latest version of *AmigaDos*, the *dos.library* is ill-prepared to take longer strings, and will likely fail or mis-interpret the string passed in. If longer strings are requried, e.g. as part of a *path*, it is (unfortunately) in the responsibility of the caller to take this path appart into components and iterate through the components manually, see also section ??.

2.5 Files

Files are streams of bytes together with a file pointer that identifies the next position to be read, or the next byte position to be filled. Files are explained in more detail in section 3.

2.6 Locks

Locks are access rights to a particular object on a file system. A locked object cannot be altered by any other process. Section 4.1.1 provides more details on locks.

2.7 Processes

AmigaDOS is a multi-tasking system operating on top of the *exec* kernel [7]. As such, it can operate multiple tasks at once, where the tasks are assigned to the CPU in a round-robin fashion. A *Process* is an extension of an AmigaOs *Task* that includes additional state information relevant to AmigaDOS, such as a *current directory Current Directory* it operates in, a *default file system*, a *console* it is connected to, and default input, output and error streams. Processes are explained in more detail in section **??**.

2.8 Handlers and File Systems

Handlers are special processes that manage files on a volume, or that input or output data to a physical device. AmigaDOS itself delegates all operations on files to such handlers. Handlers are introduced in section **??**.

File systems are special handlers that organize the contents of data carriers such as hard disks, floppies or CD-Roms in the form of files and directories, and provides access to such objects through the *dos.library*. File systems interpret paths (see 3.3) in order to locate objects such as files and directories on such data carriers.

Chapter 3

Files

3.1 What are Files?

Files are streams of sequences of bytes that can be read from and written to, along with a file pointer that points to the next byte to be read, or the next byte to be written or overwritten. Files may have an *End-of-File position*, beyond which the file pointer can not advance when reading bytes from it.

3.2 Interactive vs. non-Interactive Files

AmigaDOS knows two types of files: *Interactive* and *non-interactive* files.

Non-interactive files are stored on some persistent data carrier; unless modified by a process, the contents of such non-interactive files does not change. They also have a defined *file size*. The file size is the number of bytes between the start of the file and the end-of-file position, or short *EOF position*. This file size does not change unless some process writes to the file, which may or may not be the same process that reads from the file.

Examples for non-interactive files are data on a disk, such as a floppy or a harddisk. Such files have a name, possibly a path within a hierachical file system, and possibly multiple protection flags that define which type of actions can be applied to a file; such flags define whether the file can be read from, written to, and so on.

Interactive files depend on the interaction of the computer system with the outside world, and their contents can change due to such interaction. Interactive files may not define a clear end-of-file position, and an attempt to read from them or write to them may block an indefinite amount of time until triggered by an external event.

Examples for interactive files are the console, where reading from it depends on the user entering data in a console window and output corresponds to printing to the console; or the serial port, where read requests are satisfied by serial data arriving at the serial port and written bytes are transmitted out of the port. The parallel port is another example of an interactive file. Requests to read from it result in an error condition, while writing prints data on a printer connected to the port. Writing may block indefinitely if the printer runs out of paper or is turned off.

3.3 Paths and File Names

Files are identified by *paths*, which are strings from which AmigaDOS locates a process through which access to the file is managed. Such a process is called the *Handler* of the file, or, in case of

files of on a data carrier, also the *File System*. AmigaDOS itself does not operate on files directly, but delegates such work to its handler.

A *path* is broken up into two parts: An optional device or volume name terminated by a colon (":"), followed by string that identifies the file within the handler identified by the first part.

The first part, if present, is interpreted by AmigaDOS itself. It relates to the name of a handler (or file system) of the given name, or a known disk volume, or a logical volume of the name within the AmigaDOS *device list*. These concepts are presented in further detail in section 6.

The second, or only part is interpreted by the handler identfied by the first part.

3.3.1 Devices, Volumes and Assigns

The first part of a path, up to the colon, identifies the device, the volume or the assign a file is located in.

3.3.1.1 Devices

A *device name* identifies the handler or file system directly. Handlers are typically responsible for particular hardware units within the system, for example for the first floppy drive, or the second partition of a harddisk. For example, `df0` is the name of the handler responsible for the first floppy drive, regardless of which disk is inserted into it.

Table 2 lists all devices AmigaDOS mounts itself even without a boot volume available. They can be assumed present any time.

Table 2: System defined devices

Device Name	Description
`DF0`	First floppy drive
`PRT`	Printer
`PAR`	Parallel port
`SER`	Serial port
`CON`	Line-interactive console
`RAW`	Character based console
`PIPE`	Pipeline between processes
`RAM`	RAM-based file handler

If more than one floppy drives are connected to the system, they are named `DF1` through `DF3`. If a hard disk is present, then the device name(s) of the harddisk partitions depend on the contents of Rigid Disk Block, see ??. These names can be selected upon installation of the harddisks, e.g. through *HDToolBox*. The general convention is to name them `DH0` and following.

The following device names have a special meaning and do not belong to a particular device:

Table 3: System defined devices

Name	Description
`*`	the console of the current process
`CONSOLE`	the console of the current process
`NIL`	the data sink

The `NIL` device is a special device without a handler that is maintained by AmigaDOS itself. Any data written into it vanishes completely, and any attempt to read data from it results in an end-of-file condition.

The `*`, if used as complete path name without a trailing colon, is the current console of the process, if such a console exists. Any data output to the file named `*` will be printed on the console.

Any attempt to read from * will wait on the user to input data on the console, and will return such data.

> *Not a wildcard!* Unlike other operating systems, the asterisk * is *not* a wildcard under AmigaDOS. It rather identifies the current console of a process, or is used as escape character in . AmigaDOS shell scripts

The CONSOLE device is the default console of the process. Unlike *, but like any other device name, it shall be followed by a colon, and an optional job name. Such job names form *logical consoles* that are used by the shell for job control purposes.

> *Prefer the stars* The difference between * and CONSOLE is subtle, and the former should be preferred as it identifies the process as part of a particular shell job. An attempt to output to CONSOLE: may block the current process as it does not identify it properly as part of its job, but rather denotes the job started when creating the shell. Thus, in case of doubt, use the * without any colon if you mean the console.

Additional devices can be loaded into the system by the *Mount* command, see section ??.

3.3.1.2 Volumes

A *volume name* identifies a particular data carrier within a physical drive. For example, it may identify a particular floppy disk, regardless of the drive it is inserted it. For example, the volume name "Workbench3.2" relates always to the same floppy, regardless of whether it is inserted in the first DF0 or second DF1 drive.

3.3.1.3 Assigns

An *assign* or *logical volume* identifies a subset of a files within a file system under a unique name. Such assigns are created by the system or by the user helping to identify portions of the file system containing files that are of particular relevance for the system. For example, the assign C contains all commands of the boot shell, and the assign LIBS contains dynamically loadable system libraries. Such assigns can be changed or redirected, and by that the system can be instructed to take system resources from other parts of a file system, or entirely different file systems.

Assigns can be of three types: *Regular assigns*, *non-binding assigns* and *late assigns*. *Regular assigns* bind to a particular directory on a particular volume. If the *assign* is accessed, and the original volume the bound directory is not available, the system will ask to insert this particular volume, and no other volume will be accepted.

Regular assigns can also bind to multiple directories at once, in which case a particular file or directory within such a *multi-assign* is searched in all directories bound to the assign. A particular use case for this is the FONTS assign, containing all system-available fonts. Adding another directory to FONTS makes additional fonts available to the system without loosing the original ones.

Regular assigns have the drawback that the volume remains known to the system, and the corresponding volume icon will not vanish from the workbench.

Non-binding assigns avoid this problem by only storing the symbolic path the assign goes to; whenever the assign is accessed, any volume of the particular name containing the particular path will work. However, if this also implies that the target of the assign is not necessarily consistent, i.e. if the assign is accessed later on, another volume with different content will be accepted by the system.

Late assigns are a compromize between *regular assigns* and *non-binding assigns*. AmigaDOS initially only stores a target path for the assign, but when the assign is accessed the first time, the assign is converted to a *regular assign* and thus then binds to the particular directory of the particular volume that was inserted at the time of the first access.

Table 4 lists the assigns made by AmigaDOS automatically during bootstrap; except for the SYS assign, they all go to a directory of the same name on the boot volume. They are all *regular assigns*, except for ENVARC, which is *late assign*.

Table 4: System defined assigns

Assign Name	Description
C	Boot shell commands
L	AmigaDOS handlers and file systems
S	AmigaDOS Scripts
LIBS	AmigaOs libraries
DEVS	AmigaOs hardware drivers
FONTS	AmigaOs fonts
ENVARC	AmigaOs preferences (late)
SYS	The boot volume

In addition to the above table, the following assigns are handled by AmigaDOS internally and are not part of the *device list*, (see section 6):

Table 5: System defined assigns

Assign Name	Description
PROGDIR	Location of the executable

Thus, PROGDIR is the directory the currently executed binary was loaded from. Note that PROGDIR does not exist in case an executable file was not loaded from disk, probably because it was either taken from ROM or was made resident. More on resident executables is found in section ??.

Additional assigns can become necessary for a fully operational system, though these assigns are created through the *Startup-sequence*, a particular AmigaDOS script residing in the S assign which is executed by the boot shell. Table 6 lists some of them.

Table 6: Assigns made during bootstrap

Assign Name	Description
ENV	Storage for active preferences and global variables
T	Storage for temporary files
CLIPS	Storage for clipboard contents
KEYMAPS	Keymap layouts
PRINTERS	Printer drivers
REXX	ARexx scripts
LOCALE	Catalogs and localization

Additional assigns can always be made with the *Assign* command, see section ??.

3.3.2 Relative and Absolute Paths

As introduced in section 3.3, a path consists either of an device, volume or assign name followed by a colon followed by a second part, or the second part alone. If a device, volume or assign name is present, such a path is said to be an *absolute path* because it identifies a location within a logical or physical volume.

If no first part is present, or if it is empty, i.e. the colon is the first part of the path, AmigaDOS uses information from the calling process to identify a suitable handler. Details on this are provided in section ??. Such a path is called a *relative path*.

This second part is forwarded to the handler and is not interpreted by the *dos.library*. It is then within the responsibility of the handler to interpret this path and locate a file within the data carrier it manages, or to configure an interface to the outside world according to this path.

In general, the *dos.library* does not impose a particular syntax on how this second part looks like. However, several support functions of AmigaDOS implicitly define conventions file systems should follow to make these support functions workable and it is therefore advisable for file system implementors to follow these conventions.

3.3.3 Maximum Path Length

The *dos.library* does not enforce a limit on the size of file or directory names, except that the total length of a path including all of its components shall not be larger than 255 characters. This is because it is converted to a BSTR within the *dos.library*. How large a component name can be is a matter of the file system itself. The *Fast File System* includes variants that limit file names to 30, 56 or 106 characters.

File systems typically do not report an error if the maximum file name is exceeded; instead, the name is clamped to the maximum size without further notice, which may lead to undesired side effects. For example, a file system may clip or remove a trailing `.info` from a workbench icon file name without ever reporting this, resulting in unexpected side effects. The *icon.library* and *workbench.library* of AmigaOs take care to avoid such file names and double check created objects for correct names.

3.3.4 Flat vs. Hiearchical File Systems

?? A flat file system organizes files as a single list of all files available on a physical data carrier. For large amounts of files, such a representation is clearly burdensome as files may be hard to find and hard to identify.

For this reason, all file systems provided by AmigaOs are *hierarchical* and organize files in nested sets of *directories*, where each directory contains files or additional directories. The topmost directory of a volume forms the *root directory* of this volume.

While AmigaDOS itself does not enforce a particular convention, all file systems included in AmigaDOS follow the convention that a path consists of a sequence of zero or more directory names separated by a forwards slash ("/"), and a final file or directory name.

3.3.4.1 Locating Files or Directories

When attempting to locate a particular file or directory, the *dos.library* first checks whether an absolute path name is present. If so, it starts from the root directory on the device, physical or logical volume identified by the device or volume name and delegates the interpretation of the path to the handler.

Otherwise, it uses the *current directory* of the calling process to locate a handler responsible for the interpretation of the path name. If this current directory is `ZERO` (see section 2.3), it uses the *default file system* of the process, which by itself, defaults to the boot file system.

The second part of the path interpretation is up to the file system identified by the first step and is performed there, outside of the *dos.library*. If the path name includes a colon (":"), then locating a file starts from the root of the inserted volume. This also includes the special case of an absent device or volume name, though a present colon, i.e. ":" represents the root directory of the volume to which the current directory belongs.

The following paragraphs describe a recommended set of operations an AmigaDOS file system should follow. A path consists of a sequence of components separated by forward slashes ("/").

To locate a file, the file system should work iteratively through the path, component by component: A single isolated "/" without a preceeding component indicates the parent directory of the current directory. The parent directory of the root directory is the root directory itself.

Otherwise, a component followed by "/" instructs the handler to enter the directory of given by the component, and continue searching there.

Scanning terminates when the file system reaches the last component. The file or directory to find is then the given by the last component reached during the scan.

As scanning through directories starts with the current directory and stops when the end of the path has been reached, the empty string indicates the current directory.

> *No Dots Here* Unlike other operating systems, AmigaDOS does not use "." and ".." to indicate the current directory or the parent directory. Rather, the current directory is represented by the empty string, and the parent directory is represented by an isolated forwards slash without a preceeding component.

Thus, for example, ":S" is a file or directory named "S" in the root directory of the current directory of the process, and "//Top/Hi" is a file or directory named "Hi" two directories up from the current directory, in a directory named "Top".

3.3.5 Case Sensitivity

The *dos.library* does not define whether file names are case-sensitive or insensitive, except for the device or volume name which is case-insensitive. Most if not all AmigaDOS file-systems are also case-insensitve, or rather should. Some variants of the *Fast File System* do not handle case-insensitive comparisons correctly on non-ASCII characters, i.e. ISO-Latin code points whose most-significant bit is set, see section ?? for details. These variants should be avoided and the "international" variants should be preferred.

3.4 Opening Files

To read data from or write data to a file, it first needs to be opened by the `Open()` function:

```
file = Open( name, accessMode )
        D0           D1    D2

BPTR Open(STRPTR, LONG)
```

The `name` argument is the *path* the file to be opened, which is interpreted according to the rules given in section 3.3. The argument `accessMode` identifies how the file is opened. The function returns a `BPTR` to a *file handle* on success, or `ZERO` on failure. A secondary return code can be retrieved from `IoErr()` described in section ??. It is 0 on success, or an error code from `dos/dos.h` in case opening the file failed.

The access mode shall be one of the following, defined in `dos/dos.h`:

Table 7: Access Modes for Opening Files

Access Name	Description
`MODE_OLDFILE`	Shared access to existing files
`MODE_READWRITE`	Shared access to new or existing files
`MODE_NEWFILE`	Exclusive access to new files

Length Limited As this function needs to convert the path argument from a C string to a BSTR, path names longer than 255 characters are not supported and results are unpredictable if such names are passed into Open(). If such long path names cannot be avoided, it is the responsibility of the caller to split the path name accordingly and potentially walk through the directories manually if necessary. Note that this strategy may not be suitable for interactive files or handlers that follow conventions for the path name that are different from the conventions described in section 3.3.4.1.

The access mode `MODE_OLDFILE` attempts to find an existing file. If the file does not exist, the function fails. If the file exists, it can be read from or written from, though simultaneous access from multiple processes is possible and does not create an error condition. If multiple processes write to the same file simultaneously, the result is undefined and no particular order of the write operations is imposed.

The access mode `MODE_READWRITE` first attempts to find an existing file, but if the file does not exist, it will be created under the name given by the last component of the path. The function does not attempt to create directories within the path if they do not access. Once the file is opened, access to the file is shared, even if it has been just created. That is, multiple processes may then access it for reading or writing. If multiple processes write to the file simultenously, the order in which the writes are served is undefined and depends on the scheduling of the processes.

The access mode `MODE_NEWFILE` creates a new file, potentially erasing an already existing file of the same name if it already exists. The function does not attempt to create directories within the path if they do not exist. Access to the file is exclusive, that is, any attempt to access the file from a second process fails with an error condition.

No Wildcards The Open() function, similar to most *dos.library* functions, does not attempt to resolve wild cards. That is, any character potentially reassembling a wild card, such as "?" or "#" will taken as a literal and will be used as part of the file name. While these characters are valid, they should be avoided as they make such files hard to access from the Shell.

3.5 Closing Files

The `Close()` function writes all internally buffered data to disk and makes an exclusively opened file accessible to other processes again.

```
success = Close( file )
   D0             D1

BOOL Close(BPTR)
```

The `file` is a BPTR to `FileHandle` identifying the file. The return code indicates whether the file system could successfully close the file and write back any data. If the result code is `DOSFALSE`, an error code can be obtained through `IoErr()` described in section `??`. Otherwise, `IoErr()` will not be altered.

Unfortunately, not much can be done if closing a file fails and no general advise is possible how to handle this situation.

Attempting to close the `ZERO` file handle returns success immediately.

3.6 Types of Files and Handlers

As introduced in 3.2, AmigaDOS distinguishes between non-interactive files managed by file systems and interactive files that interact with the outside world. Typically, *file systems* create non-

interactive files; all other *handlers* create interactive or non-interactive files, depending on the nature of the handler.

3.6.1 Obtaining the Type from a File

A file can be either interactive, in which case attempts to read or write data to the file may block indefinitely, or non-interactive where the amount of available data is determined by file itself. The `IsInteractive()` function returns the nature of an already opened file.

```
status = IsInteractive( file )
 D0                      D1

BOOL IsInteractive(BPTR)
```

The `IsInteractive()` function returns `TRUE` in case the *file handle* passed in is interactive, or `FALSE` in case it corresponds to a non-interactive stream of bytes, potentially on a file system. This function cannot fail and does not alter `IoErr()`.

3.6.2 Obtaining the Type from a Path

A *handler* that manages physical data carriers and allows to access named files on such data carriers is a *file system*. The `IsFileSystem()` function determines the nature of a handler given a path (see 3.3) to a candidate handler.

```
result = IsFileSystem(name)
D0                     D1

BOOL IsFileSystem(STRPTR)
```

The `name` argument is a path that does not need to identify a physically existing object. Instead, it is used to identify a handler that would be responsible to such a hypothetical object regardless whether it exists or not.

It is of advisable to provide a path that identifies the handler uniquely, i.e. a string that is terminated by a colon (":").

The returned result is `DOSTRUE` in case the handler identified by the path is a file system, and as such allows access to multiple files on a physical data carrier. Otherwise, it returns `DOSFALSE`.

3.7 Unbuffered Input and Output

The functions described in this section read bytes from or write bytes to already opened files. These functions are *unbuffered*, that is, any request goes directly to the handler. Since a request performs necessarily a task switch from the caller to the handler managing the file, these functions are inefficient on small amounts of data and should be avoided. Instead, files should be read or written in larger chunks, either by buffering data manually, or by using the buffered I/O functions described in section ??.

3.7.1 Reading Data

The following function reads data from an opened file by directly invoking the handler for performing the read:

```
actualLength = Read( file, buffer, length )
  D0                   D1    D2      D3

LONG Read(BPTR, void *, LONG)
```

The `Read()` function reads `length` bytes from an opened file identified by the *file handle* `file` into the buffer pointed to by `buffer`. The buffer is a standard C pointer, not a BPTR.

The return code `actualLength` is the amount of bytes actually read, or -1 for an error condition. A secondary return code can be retrieved from `IoErr()` described in section ??. It is 0 on success, or an error code from `dos/dos.h` in case reading failed.

The amount of data read may be less data than requested by the `length` argument, either because the *EOF* position has been reached (see section 3.2) for non-interactive files, or because the interactive source is depleted. Note that for interactive files, the function may block indefinitely until data becomes available.

3.7.2 Testing for Availability of Data

An issue of the `Read`function is that it may block indefinitely on an interactive file if the user does not enter any data. The `WaitForChar()` tests for the availability of a character on an interactive file for limited amount of time and returns if no data becomes available.

```
status = WaitForChar( file, timeout )
  D0                    D1    D2

BOOL WaitForChar(BPTR, LONG)
```

This function waits for a maximum of `timeout` microseconds for the availability of input on `file`. If data is already available, or becomes available within this time, the function returns `DOSTRUE`. Otherwise, the function returns `DOSFALSE`.

A secondary return code can be obtained from `IoErr()`. If it is 0, the handler was able check the availablility of a byte from the given file. Otherwise, an error code from `dos/dos.h` indicates failure of the function.

This function requires an interactive file to operate, file systems will typically not implement this function as they do not block.

3.7.3 Writing Data

The following call writes an array of bytes unbuffered to a file, interacting directly with the corresponding handler:

```
returnedLength =  Write( file, buffer, length )
 D0                       D1    D2      D3

LONG Write (BPTR, void *, LONG)
```

The Write function writes `length` bytes in the buffer pointed to by `buffer` to the *file handle* given by the `file` argument. On success, it returns the number of bytes written as `returnedLength`, and advances the file pointer of the file by this amount. Note that this amount of bytes may even be 0 in case the file cannot absorb any more bytes. On error, -1 is returned.

A secondary return code can be retrieved from `IoErr()` described in section ??. It is 0 on success, or an error code from `dos/dos.h` in case writing failed.

For interactive files, this function may block indefinitely until the corresponding handler is able to take additional data.

3.7.4 Adjusting the File Pointer

The `Seek()` adjusts the file pointer of a non-interactive file such that subsequent reading or writing is performed from an alternative position of the file.

```
oldPosition = Seek( file, position, mode )
 D0                 D1    D2        D3

LONG Seek(BPTR, LONG, LONG)
```

This function adjusts the file pointer of `file` relative to the position determined by `mode` by `position` bytes. The value of `mode` shall be one of the following options, defined in `dos/dos.h`:

Table 8: Seek Modes

Mode Name	Description
`OFFSET_BEGINNING`	Seek relative to the start of the file
`OFFSET_CURRENT`	Seek relative to the current file position
`OFFSET_END`	Seek relative to the end of the file

> *Undefined on Interactive Files* The `Seek` function will typically indicate failure if applied to interactive files. Some handlers may assign this function, however, a particular meaning. See the handler definition for details.

If `mode` is `OFFSET_BEGINNING`, then the new file pointer is placed `position` bytes from the start of the file, i.e. the new file pointer is equal to `position`.

If `mode` is `OFFSET_CURRENT`, then `position` is added to the file pointer. That is, the file pointer is advanced if `position` is positive, or rewinded if `position` is negative.

If `mode` is `OFFSET_END`, then the end-of-file position is determined, and `position` is added to this position. This, in particular, requires that `position` should be negative.

The `Seek()` function returns the file pointer before its adjustment, or -1 in case of an error.

A secondary return code can be retrieved from `IoErr()` described in section ??. It is 0 on success, or an error code from `dos/dos.h` in case adjusting the file pointer failed.

> *Not 64bit safe* Unfortunately, it is not quite clear how `Seek` operates on files that are larger than 2GB, and it is file system dependent how such files could be handled. `OFFSET_BEGINNING` can probably only reach the first 2GB of a larger file as the file system may interpret negative values as an attempt to reach a file position upfront the start of the file and may return an error. Similarly, `OFFSET_END` may possibly only reach the last 2GB of the file. Any other position within the file may be reached by splitting the seek into chunks of at most 2GB and perform multiple `OFFSET_CURRENT` seeks. However, whether such a strategy suceeds is pretty much file system dependent. Note in particular that the return code of the function does not allow to distinguish between a file pointer just below the 4GB barrier and an error condition. A zero result code of `IoErr()` should be then used to learn whether a result of -1 indicates a file position of `0xffffffff` instead. Most AmigaDOS file systems may not be able to handle files larger than 2GB.

Even though `Seek()` is an unbuffered function, it is aware of a buffer and implicitly flushes the file system internal buffer. That is, it can be safely used by buffered and unbuffered functions.

3.7.5 Setting the Size of a File

The `SetFileSize()` function truncates or extends the size of an opened file to a given size. Not all handlers support this function.

```
newsize = SetFileSize(fh, offset, mode)
D0                    D1    D2     D3

LONG SetFileSize(BPTR, LONG, LONG)
```

This function extends or truncates the size of the file identified by the *file handle* `fh`; the target size is determined by the current file pointer, `offset` and the `mode`. Interpretation of `mode` and `offset` is similar to `Seek()`, except that the end-of-file position of the file is adjusted, and not the file pointer.

The `mode` shall be selected from to table 8. In particular, it is interpreted as follows:

If `mode` is `OFFSET_BEGINNING`, then the file size is set to the value of `offset`, irrespectible of the current file pointer.

If `mode` is `OFFSET_CURRENT`, then the new end-of-file position is set `offset` bytes relative to the current file pointer. That is, the file is truncated if `offset` is negative, and extended if `offset` is positive.

If `mode` is `OFFSET_END`, the new file size is given by the current file size plus `offset`. That is, the file is extended by `offset` bytes if positive, or truncated otherwise. The value of the current file pointer is irrelevant and ignored.

If the current file pointer of any *file handle* opened on the same file is, after a potential truncation, beyond the new end-of-file, it is clamped to the end-of-file. They remain unchanged otherwise.

If the file is enlarged, the values within the file beyond the previous end-of-file position are undetermined.

The return value `newsize` is the size of the file after the adjustment, i.e. the position of the end-of-file location.

> *Not 64bit safe* Similar to `Seek()`, `SetFileSize()` cannot be assumed to work properly if the (old or new) file size is larger than 2GB. What exactly happens if an attempt is made to adjust the file by more than 2GB depends on the file system performing the operation. A possible strategy to adjust the file size to a value above 2GB is to first seek to the closest position, potentially using multiple seeks of maximal size, and then perform one or multiple calls to `SetFileSize()` with the `mode` set to `OFFSET_CURRENT`. However, whether this strategy succeeds is file system dependent.

3.8 Buffered Input and Output

AmigaDOS also offers buffered input and output functions that stores data in an intermediate buffer. AmigaDOS then transfers data only in larger chunks between the buffer and the handler, minimizing the task switching overhead and offering better performance if data is to be read or written in smaller units.

> *Performance Improved* While buffered I/O functions of AmigaOs 3.1.4 and below were designed around single-byte functions and thus caused massive overhead in the buffered functions described in this section, the functions in this section were redesigned in AmigaOs 3.2 and now offer significantly better performance. Unfortunately, the default buffer size AmigaDOS uses is quite small and should be significantly increased by `SetVBuf()`. A suggested buffer size is 4096 bytes which corresponds to a disk block of modern hard drives.

3.8.1 Buffered Read

The `FRead()` function reads multiple equally-sized records from a file through a buffer, and returns the number of records retrieved.

```
count = FRead(fh, buf, blocklen, blocks)
 D0           D1  D2     D3        D4

LONG FRead(BPTR, STRPTR, ULONG, ULONG)
```

This function reads `blocks` records each of `blocklen` bytes from the file `fh` into the buffer `buf`. It returns the number of complete records retrieved from the file. If the file runs out of data, the last record may be incomplete.

From AmigaOs 3.2 onwards, `FRead()` first attempts to satisfy the request from the file handle internal buffer, but if the number of remaining bytes is larger than the buffer size, the handler will be invoked directly for "bursting" the data into the target buffer, bypassing the file buffer.

This function does not modify `IoErr()` in case the request can be satisfied completely from the file handle buffer. It neither returns −1 in case of an error. Callers should instead use `SetIoErr(0)` to clear the error state before calling this function, and then use `IoErr()` to learn if any error occured if the number of records read is smaller than the number of records requested.

3.8.2 Buffered Write

The `FWrite()` function writes multiple equally-sized records to a file through a buffer, and returns the number of records it could write.

```
count = FWrite(fh, buf, blocklen, blocks)
D0             D1  D2     D3        D4

LONG FWrite(BPTR, STRPTR, ULONG, ULONG)
```

This function write `blocks` records each of `blocklen` bytes from the buffer `buf` to the file `fh`. It returns the number of complete records written to the file. On an error, the last record written may be incomplete.

From AmigaOs 3.2 onwards, `FWrite()` first checks whether the file handle internal buffer is partially filled. If so, the file handle interal buffer is filled from `buf`. If any bytes remain to be written, and the number of bytes is larger than the internal buffer size, the handler will be invoked to write the data in a single block, bypassing the buffer. Otherwise, the data will be copied to the internal buffer.

This function does not modify `IoErr()` in case the request can be satisfied completely by using the file handle buffer. It neither returns −1 in case of an error. Callers should instead use `SetIoErr(0)` to clear the error state before calling this function, and then use `IoErr()` to learn if any error occured if the number of records written is smaller than the number of records passed in.

3.8.3 Adjusting the Buffer

The `SetVBuf()` function allows to adjust the internal buffer size for buffered input/output functions such as `FRead()` or `FWrite()`. It also sets the buffer mode. The default buffer size is 204 characters, which is too low for many applications.

```
error = SetVBuf(fh, buff, type, size)
D0              D1   D2    D3    D4

LONG SetVBuf(BPTR, STRPTR, LONG, LONG)
```

This function sets the internal buffer of the *file handle* `fh` to `size` bytes. Sizes smaller than 204 characters will be rounded up to 204. If `buff` is non-`NULL`, it is a pointer to a user-provided buffer that will be used for buffering. This buffer shall be aligned to a 32-bit boundary. A user provided buffer will not be released when the file is closed.

Otherwise, if `buff` is `NULL` AmigaDOS will allocate the buffer for you, and will also release it when the file is closed.

The `type` argument identifies the type of buffering according to Table 9; the modes there are defined in the include file `dos/stdio.h`.

Table 9: Buffer Modes

Buffer Name	Description
`BUF_LINE`	Buffer up to end of line
`BUF_FULL`	Buffer everything
`BUF_NONE`	No buffering

The buffer mode `BUF_LINE` automatically flushes the buffer when writing a line feed (0x0a), carriage return (0x0c) or ASCII NUL (0x00) character to the buffer, and the target file is interactive. Otherwise, the characters remain in the buffer until it either overflows or is flushed manually, see `Flush()`.

The buffer mode `BUF_FULL` buffers all characters until the buffer either overflows or is flushed.

The buffer mode `BUF_NONE` effectively disables the buffer and writes all characters to the target file immediately.

On reading, `BUF_LINE` and `BUF_FULL` are equivalent and fill the entire buffer from the file; `BUF_NONE` disables buffering.

The function returns non-zero on success, or 0 on error. Error conditions are either out-of-memory, an invalid buffer mode or an invalid file handle. Unfortunately, `IoErr()` is only set on an out-of-memory condition and remains otherwise unchanged.

3.8.4 Synchronize the File to the Buffer

The `Flush()` function flushes the internal buffer of a *file handle* and synchronizes the file pointer to the buffer position.

```
success = Flush(fh)
D0              D1

LONG Flush(BPTR)
```

Synchronizes the file pointer to the buffer, that is, if bytes were written to the buffer, writes out buffer content to file. If bytes were read from the file and non-read files remained in the buffer, such bytes are dropped and the function attempts to seek back to the position of the last read byte. This can fail for interactive files.

The return code is currently always `DOSTRUE` and thus cannot be used as an indication of error, even if not all bytes could be written, or if seeking failed. If error detection is desired, the caller should first use `SetIoErr(0)` to erase an error condition, then call flush, and then use `IoErr()` to check whether an error occurred.

> *Flush when switching between reading and writing* The `Flush()` function shall be called when switching from writing to a file to reading from the same file, or vice versa. The internal buffer logic is unfortunately not capable to handle this case correctly. Also, `Flush()` shall be called when switching from buffered to unbuffered input/output.

3.8.5 Write a Character buffered to a file

The `FPutC()` function writes a single character to a file, using the *file handle* internal buffer.

```
char = FPutC(fh, char)
D0           D1   D2

LONG FPutC(BPTR, LONG)
```

This function writes the single character `char` to the *file handle* `fh`. Depending on the buffer mode, the character and the type of file, the character may go to the buffer first, or may cause the buffer to be emptied. See `SetVBuf()` for details on buffer modes and conditions for implicit buffer flushes.

It returns the character written, or `ENDSTREAMCH` on an error. The latter constant is defined in `dos/stdio.h` and equals to -1.

This function does not touch `IoErr()` if the character only goes into the internal buffer.

3.8.6 Write a String Buffered to a File

The `FPuts()` function writes a `NUL`-terminated string to a file, using the *file handle* internal buffer.

```
error = FPuts(fh, str)
D0            D1  D2

LONG FPuts(BPTR, STRPTR)
```

This function writes the NUL-terminated (C-style) string `str` to the *file handle* `fh`. The terminating `NUL` character is not written.

Depending on the buffer mode, the string will first go into the buffer, or may be written out immediately. See `SetVBuf()` for details on buffer modes and conditions for implicit buffer flushes.

This function returns 0 on success, or `ENDSTREAMCH` on an error. The latter constant is defined in `dos/stdio.h` and equals to -1.

3.8.7 Read a Character from a File

The `FGetC()` function reads a single character from a file through the internal buffer of the *file handle*.

```
char = FGetC(fh)
D0           D1

LONG FGetC(BPTR)
```

This function attempts to read a single character from the *file handle* `fh` using the buffer of the handle. If characters are present in the buffer, the request is satisfied from the buffer first, then the function attempts to refill the buffer from the file and tries again.

The function returns the character read, or `ENDSTREAMCH` on an end-of-file condition or an error. The latter constant is defined in `dos/stdio.h` and equals to -1.

To distingish between the error and the end-of-file case, the caller should first reset the error condition with `SetIoErr(0)`, and then check `IoErr()` when the function returns with `ENDSTREAMCH`.

3.8.8 Read a Line from a File

The `FGets()` function reads a newline-terminated string from a file, using the *file handle* internal buffer.

```
buffer = FGets(fh, buf, len)
D0             D1  D2   D3

STRPTR FGets(BPTR, STRPTR, ULONG)
```

This function reads a line from the *file handle* into the buffer pointed to by `buf`, capable of holding `len` characters.

Reading terminates either if `len`-1 characters have been read, filling up the buffer completely; or a line-feed character is found, which is copied into the buffer; or if an end-of-file condition or an error condition is encountered. In either event, the string is `NUL` terminated.

The function returns `NULL` in case not even a single character could be read. Otherwise, the function returns the buffer passed in.

To distinguish between the error and end-of-file condition, the caller should first use `SetIoErr(0)`, and then test `IoErr()` in case the function returns `NULL`.

3.8.9 Revert a Single Byte Read

The `UnGetC()` function reverts a single byte read from a stream and makes this byte available for reading again.

```
value = UnGetC(fh, ch)
D0             D1  D2

LONG UnGetC(BPTR, LONG)
```

The character `ch` is pushed back into the *file handle* `fh` such that the next attempt to read a character from `fh` returns `ch`. If `ch` is -1, the last character read will be pushed back. If the last read operation indicated an error or end-of-file condition, `UnGetC(fh,-1)` pushes an end-of-file condition back.

This function returns non-zero on success or 0 if the character could not be pushed back. At most a single character can be pushed back after each read operation, an attempt to push back more characters can fail.

3.9 File Handle Documentation

So far, the *file handle* has been used as an opaque value bare any meaning. However, the `BPTR`, once converted to a regular pointer, is a pointer to `struct FileHandle`:

```
BPTR file = Open("S:Startup-Sequence,MODE_OLDFILE);
struct FileHandle *fh = BADDR(file);
```

In the following sections, this structure and its functions are documented.

3.9.1 The struct FileHandle

When opening a file via `Open()`, the *file handle* is allocated by the *dos.library* by going through `AllocDosObject()`, and then forwarded to the file system or handler for second-level initialization. It is documented in `dos/dosextens.h` as replicated here:

```
struct FileHandle {
   struct Message *fh_Link;
   struct MsgPort *fh_Port;
   struct MsgPort *fh_Type;
   BPTR fh_Buf;
   LONG fh_Pos;
   LONG fh_End;
   LONG fh_Funcs;
#define fh_Func1 fh_Funcs
   LONG fh_Func2;
   LONG fh_Func3;
   LONG fh_Args;
#define fh_Arg1 fh_Args
   LONG fh_Arg2;
};
```

`fh_Link` is actually not a pointer, but an AmigaDOS internal value that shall not be interpreted or touched, and of which one cannot make productive use.

`fh_Port` is similarly not a pointer, but a `LONG`. If it is non-zero, the file is interactive, otherwise it is a file system. `IsInteractive()` makes use of this member. The file system or handler shall initialize this value when opening a file and shall initialize it according to the nature of the handler.

`fh_Type` points to the `MsgPort` of the handler or file system that implements all input and output operations. Section ?? provides additional information on how handlers and file systems work. If this pointer is `NULL`, no handler is associated to the file handle. This is also the value AmigaDOS will deposit here when opening a file to the `NIL:` (pseudo-)device. Attempting to `Read()` from this handle results in an end-of-file situation, and calling `Write()` on such a handle does nothing, ignoring any data written.

`fh_Buf` is a `BPTR` to the file handle internal buffer all buffered I/O function documented in this section use.

`fh_Pos` is the next read or write position within this buffer.

`fh_End` is the size of the buffer in bytes.

`fh_Func1` is a function pointer that is called whenever the buffer is to be filled through the handler. Users shall not call this function itself, and the function prototype is intentionally not documented.

`fh_Func2` is a second function pointer that is called whenever the buffer is full and is to be written by the handler. Users shall not call this function itself, and the function prototype is intentially undocumented.

`fh_Func3` is a final function pointer that is called whenever the file handle is closed. This function then potentially writes the buffer content out when dirty, releases the buffer if it is system-allocated, and finally forwards the close request to the handler.

`fh_Arg1` is a file-system internal value the handler or file system uses to identify the file. The interpretation of this value is to the file system or handler, and the *dos.library* does not attempt to interpret it. The handler deposits the file identification here when opening a file, and the *dos.library* forwards it to the handler on `Read()` and `Write()`. See section ?? for details.

`fh_Arg2` is currently unused.

3.9.2 String Streams

It is sometimes useful to provide programs with (temporary) input not coming from a file system or handler directly, even though the program uses a file interface to access it. One solution to this problem is to deposit the input data on the `RAM` disk, then opening this file and providing it as input

to such a program. The drawback of this approach is that additional tests are necessary to ensure that the file name is unique, and to avoid that other than the intended program accesses it.

AmigaDOS uses the technique documented here itself, for example to provide the command to be executed by the `Run` command. There, the string stream contains the command to be run in background, which is then provided as input file to the shell. The `System()` function of the *dos.library* makes use of the same trick to feed the command to be executed as input file. Thus, even though the shell can only execute commands from a file, AmigaDOS can generate *file handles* that do not correspond to a handler, but to a string in memory containing the commands.

The shell itself is using the same technique to pass arguments to the commands it executes; it deposits the command arguments in the file handle buffer of the input stream where `ReadArgs()` collects them.

The idea is to allocate a `struct FileHandle` and initialize its buffer to contain the string within the file. For this `fh->Buf` needs to point to the buffer containing the string, and `fh->End` needs to be its size. The function pointers in the *file handle* remain 0 such as to avoid that the *dos.library* reads, writes or flushes the buffer. The `FileHandle` shall be allocated by `AllocVec()` as the *dos.library* releases the handle through `FreeVec()`.

The following program demonstrates this technique:

```
#include <exec/memory.h>
#include <dos/dos.h>
#include <dos/stdio.h>
#include <string.h>
#include <proto/dos.h>
#include <proto/exec.h>

int main(int argc,char **argv)
{
  const char *test = "Hello World!\n";
  const int len = strlen(test)+1;
  struct FileHandle *fh;
  BPTR file;

  fh   = AllocVec(sizeof(struct FileHandle) + len,MEMF_PUBLIC|MEMF_CLEAR);
  if (fh) {
    UBYTE *c = (UBYTE *)(fh + 1);
    file     = MKBADDR(fh);
    memcpy(c,test,len);
    fh->fh_Buf = MKBADDR(c);
    fh->fh_End = len;
    {
      BPTR out = Output();
      LONG ch;
      while((ch = FGetC(file)) >= 0) {
        FPutC(out,ch);
      }
    }
    Close(file);
  }
  return 0;
}
```

Here the buffer is allocated along with the file handle, and thus released along with it. Setting

MEMF_PUBLIC is of utter importance as it clears all function pointers, and in particular the fh_Link field to zero; the latter is an indication to the *dos.library* that this structure was not allocated through itself.

3.9.3 An FSkip() Implementation

Unlike most unbuffered functions, Seek() can be safely mixed with buffered input and output functions. However, this function is not very efficient, and seeking should be avoided if buffer manipulation is sufficient. Buffer manipulation has the advantage that small amounts of bytes can be skipped easily without going through the file system; skipping over larger amounts of bytes can be performed by a single function without requiring to read bytes.

The following function implements an FSkip() function that selects the most viable option and is more efficient that Seek() for buffered reads.

```
LONG FSkip(BPTR file,LONG skip)
{
  LONG res;
  struct FileHandle *fh = BADDR(file);
  if (fh->fh_Pos >= 0 && fh->fh_End > 0 && fh->fh_Func3) {
    LONG newpos = fh->fh_Pos + skip;
    if (newpos >= 0 && newpos < fh->fh_End) {
      fh->fh_Pos = newpos;
      return DOSTRUE;
    }
  }
  skip += fh->fh_Pos - fh->fh_End;
  fh->fh_Pos = -1;
  fh->fh_End = -1;
  if (Seek(fh,skip,OFFSET_CURRENT) != -1)
   return DOSTRUE;
  return DOSFALSE;
}
```

The first if-condition checks whether the buffer is actually present. Then, the new buffer position is computed. If it is within the buffer, the new buffer position is installed as the work is done.

Otherwise, the skip distance is adjusted by the buffer position. Initializing the buffer size and position to −1 ensures that the following Seek() does not attempt to call Flush() internally.

There is one particular catch, namely that the file needs to be initialized for reading immediately after opening the file, or the buffer will not be in the right state for the trick:

```
BPTR file = Open(filename,MODE_OLDFILE);
UnGetC(file,-1); /* initialize buffer */
```

This is only necessary if the first access to the file is an FSkip().

3.10 Formatted Output

The functions in this section print strings formatted to a file. Both files use the internal buffer of the *file handle*.

3.10.1 Print Formatted String using C-Syntax

The `VFPrintf()` function prints multiple datatypes using a format string that closely reessembles the syntax of the C syntax. `FPrintf()` is based on the same entry point of the *dos.library*, though the prototype for the C language is different and thus arguments are expected directly as function arguments instead of requiring them to be collected in an array upfront.

```
count = VFPrintf(fh, fmt, argv)
D0               D1  D2    D3

LONG VFPrintf(BPTR, STRPTR, LONG *)

count = FPrintf(fh, fmt, ...)

LONG FPrintf(BPTR, STRPTR, ...)
```

This function uses the `fmt` string to format an array of arguments pointed to by `argv` and outputs the result to the file `fh`. The syntax of the format string is identical to that of the exec function `RawDoFmt()`, and shares its problems. In particular, format strings indicating integer arguments such as `%d` and `%u` assume 16bit integers, independent of the integer model of the compiler. On compilers working with a 32bit integer models, the format modifier `l` should be used, e.g. `%ld` for signed and `%lu` for unsigned integers.

As `RawDoFmt()` is also patched by the *locale.library*, additional syntax elements from the `FormatString()` function of this library become available for `VFPrintf()` and `FPrintf()`.

The result `count` delivers the number of characters written to the file, or -1 for an error. In the latter case, `IoErr` provides an error code.

3.10.2 BCPL Style Formatted Print to a File

The `VFWritef()` function formats several arguments according to a format string similar to `VFPrintf()`, but uses the formatting syntax of the BCPL language. The main purpose of this function is to offer formatted output for legacy BCPL programs where this function appears as an entry of the BCPL *Global Vector*. New code should not use this function but rather depend on `VFPrintf()` which also gets enhanced by the *locale.library*.

The `FWritef()` uses the same entry point of the *dos.library*, though the compiler prototype imposes a different calling syntax where the objects to be formatted are directly delivered as function arguments rather requiring the caller to collect them in an array upfront.

```
count = VFWritef(fh, fmt, argv)
D0               D1  D2    D3

LONG VFWritef(BPTR, STRPTR, LONG *)

count = FWritef(fh, fmt, ...)

LONG FWritef(BPTR, STRPTR, ...)
```

This function formats the arguments from the array pointed to by `argv` according to the format string in `fmt` and writes the output to the file `fh`. The format string follows the syntax of the BCPL language. The following format identifiers are supported:

`%S` Write a `NUL` terminated string from the array to the output.

`%Tx` Writes a `NUL` terminated string left justified in a field whose width is given by the character `x`. The length indicator is always a single character; a digit from `0` to `9` indicates the field widths from 0 to 9 directly. Characters `A` to `Z` indicate field widths from 10 onwards.

`%C` Writes a single character whose ISO-Latin-1 code is given as a 32-bit integer on the `argv` array.

`%Ox` Writes an integer in octal to the output where `x` indicates the maximal field width. The field width is a single character that is encoded similarly to the `%T` format string.

`%Xx` Writes an integer in hexadecimal to the output in a field that is at most `x` characters long. `x` is a single character and encodes the width similar to that `%T` format string.

`%Ix` Writes a (signed) integer in decimal to the output in field that is at most `x` characters long. The field length is again indicated by a single character.

`%N` Writes a (signed) integer in decimal to the output without any length limitation.

`%Ux` Writes an unsigned integer in decimal to the output, limiting the field length to at most `x` characters, where `x` is encoded in a single character.

`%$` Ignores the next argument, i.e. skips over it.

This function is *not* patched by the *locale.library* and therefore is not localized or enhanced.

While the same function can also be found in the BPCL *Global Vector*, it there takes BSTRs instead of regular C strings for the format string and arguments of the `%S` and `%T` formats.

Chapter 4

Locks

Locks are access rights to objects, such as files or directories, on a file system. Once an object has been locked, it can no longer be deleted, or in case of files, it can no longer altered either. Depending on the file system, locks may also prevent other forms of changes of the object.

Locks come in two types: *Exclusive* and *shared locks*. Only a single exclusive lock can exist on a file system object at a time, and no other locks on an exclusively locked object can exist. An attempt to lock an exclusively locked object results in failure, and attempting to exclusively lock an object that is already shared locked will also fail.

Multiple *shared locks* can be kept on the same object at the same time, though once a shared lock has been obtained, any attempt to lock the same object exclusively fails.

One particular use case of *locks* is to serve as an identifier of a particular directory or file on a file system. Since paths are limited to 255 characters, see 3.3, locks are the preferred method of indicating a position within a file system. Even though paths are length limited, there is no restriction on the depth within the directory structure of a file system. The `ZERO` lock identifies the boot volume, also known as `SYS:`, see also section 3.3.1.3.

Locks are also the building stone of files; in fact, every file is internally represented by a lock on the corresponding object, even if the file system does not expose this lock to the caller.

As long as at least a single lock is held of an object on a particular volume, the file system will keep the volume within the *device list* of the *dos.library*, see section 6. This has, for example, the consequence that the workbench will continue to show an icon representing the volume in its window.

4.1 Obtaining and Releasing Locks

Locks can be obtained either explictly from a path, or can be derived from another lock or file. As locks block altering accesses to an object of a file system, locks need to be released as early as possible to allow other accesses to the locked object.

4.1.1 Obtaining a Lock from a Path

.

The `Lock()` function obtains a lock on an object given a path to the object. The path can be either absolute, or relative (see section 3.3) to the current directory of the calling process.

```
lock  = Lock( name, accessMode )
D0            D1        D2
```

```
BPTR Lock(STRPTR, LONG)
```

This function locks the object identified by `name`, which is the path to the object. The type of the lock is identified by `accessMode`. This mode shall be one of the two following modes, defined in `dos/dos.h`:

Table 10: Lock Access Modes

Access Mode	Description
`SHARED_LOCK`	Lock allowing shared access from multiple sources
`ACCESS_READ`	Synonym of the above, identical to `SHARED_LOCK`
`EXCLUSIVE_LOCK`	Exclusive lock, only allowing a single lock on the object
`ACCESS_WRITE`	Synonym of the above, identical to `EXCLUSIVE_LOCK`

The access mode `SHARED_LOCK` or `ACCESS_READ` allows multiple shared locks on the same object. This type of lock should be preferred. The access mode `EXCLUSIVE_LOCK` or `ACCESS_WRITE` only allows a single, exclusive lock on the same object.

The return code `lock` identifies the lock. It is non-`ZERO` (see 2.3) on success, or `ZERO` on failure. In either case, `IoErr()` is set to 0 indicating success, or an error code on failure.

*No Wildcards Here!*Note that this function does not attempt to resolve wild cards, similar to `Open()`. All characters in the path are literals.

4.1.2 Duplicating a Lock

The `DupLock()` function replicates a given *lock*, returning a copy of the *lock* given as argument. This requires that the original *lock* is a *shared lock*, and it returns a *shared lock* if successful.

```
lock = DupLock( lock )
D0              D1

BPTR DupLock(BPTR)
```

This function copies the (shared) lock passed in as `lock` and returns a copy of it in `lock`. In case of error, it returns `ZERO`, and then `IoErr()` returns an error code identifying the error. On success, `IoErr()` is reset. It is not possible to copy an *exclusive lock*.

4.1.3 Obtaining the Parent of an Object

The `ParentDir()` function obtains a *shared lock* on the directory containing the locked object passed in. For directories, this is the parent directory, for files, this is the directory containing the file.

```
newlock = ParentDir( lock )
D0                   D1

BPTR ParentDir(BPTR)
```

The `lock` argument identifies the object whose parent is to be found; the function returns a *lock* on the directory containing the object. If such parent does not exist, or an error occurs, the function returns `ZERO`. The former case applies to the topmost directory of a file system, or the `ZERO` lock itself.

To distinguish the two cases, the caller should check the `IoErr()` function; if this function returns 0, then no error occurred and the passed in object is topmost and no parent exists. If it returns a non-zero error code, then the file system failed to identify the parent directory.

4.1.4 Creating a Directory

The `CreateDir()` object creates a new empty directory whose name is given by the last component of the path passed in. It does not create any intermediate directories between the first component of the path and its last component, such directories need potentially be created manually by multiple calls to this function.

```
lock = CreateDir( name )
 D0                D1

BPTR CreateDir(STRPTR)
```

The `name` argument is the path to the new directory to be created; that is, the directory given by the last component of the path (see section 3.3) will be created. If successful, the function returns an *exclusive lock* in `lock`, otherwise it returns `ZERO`.

In either case, `IoErr()` is set to either an error code, or to 0 in case the function succeeds.

Note that not all file systems support directories, i.e. flat file systems (see section ??) do not.

4.1.5 Releasing a Lock

Once you are done with a *lock* and no part of your program is using it anymore, you should release it to allow other processes or functions to access or modify the locked object. Note that setting the `CurrentDir()` to a particular lock implies usage of the lock, i.e. the lock installed as `CurrentDir()` shall not be unlocked.

```
UnLock( lock )
         D1

void UnLock(BPTR)
```

This function releases the *lock* passed in as `lock` argument. Passing `ZERO` as a lock is fine and performs no activity.

4.1.6 Changing the Type of a Lock

Once a *lock* has been granted, it is possible to change the nature of the lock, either from `EXCLUSIVE_LOCK` to `SHARED_LOCK`, or — if this is the only *lock* on the object — vice versa.

```
success = ChangeMode(type, object, newmode)
D0                   D1     D2      D3

BOOL ChangeMode(ULONG, BPTR, ULONG)
```

This function changes the access mode of `object` whose type is identified by `type` to the access mode `newmode`. The relation between `type` and the nature of the object shall be as in table 11, where the types are defined in `dos/dos.h`:

Table 11: Object Types for ChangeMode()

type	object Type
`CHANGE_LOCK`	`object` shall be a *lock*
`CHANGE_FH`	`object` shall be a *file handle*

The argument `newmode` shall be one of the modes indicated in Table 10, i.e. `SHARED_LOCK` to make either the file or the lock accessible for shared access, and `EXCLUSIVE_LOCK` for exclusive access.

On success, the function returns a non-zero result code, and `IoErr()` is set to 0. Otherwise, the function returns 0 and sets `IoErr()` to an appropriate error code.

Unfortunately, this function may not work reliable for *file handles* under all versions of AmigaDOS. In particular, the *RAM-Handler* does not interpret `newmode` correctly for `CHANGE_FH`.

4.1.7 Comparing two Locks

The `SameLock()` function compares two locks and returns information whether they are identical, or at least correspond to objects on the same volume.

```
value = SameLock(lock1, lock2)
 D0                D1     D2

LONG SameLock(BPTR, BPTR)
```

This function compares `lock1` with `lock2`. The return code, all of them defined in `dos/dos.h`, can be one of the following:

Table 12: Lock Comparison Return Code

Return Code	Description
`SAME_LOCK`	Both locks are on the same object
`SAME_VOLUME`	Locks are on different objects, but on the same volume
`LOCK_DIFFERENT`	Locks are on different volumes

This function does not set `IoErr()` consistently, and callers cannot depend on its value. Furthermore, the function does not compare a `ZERO` lock with lock on the boot volume, e.g `SYS:` as identical. It is recommended not to pass in the `ZERO` lock for either `lock1` or `lock2`.

4.2 Locks and Files

Each *file handle* is associated to a lock to the file that has been opened. The type of the *lock* depends on the access mode the file has been opened with, table 13 for how lock types and access modes relate.

Table 13: Lock and File Access Modes

Access Mode	Lock Type
`MODE_OLDFILE`	`SHARED_LOCK`
`MODE_READWRITE`	`SHARED_LOCK`
`MODE_NEWFILE`	`EXCLUSIVE_LOCK`

The association of `MODE_READWRITE` to `SHARED_LOCK` is unfortunate, and due to a defect in the *RAM-Handler* implementation in AmigaDOS 2.0 which was then later copied into the *Fast File System* implementation. Exclusive access to a file without deleting its contents can, however, be established through the `OpenFromLock()` function passing in an *exclusive lock* to the function as argument.

4.2.1 Duplicate the Implicit Lock of a File

The `DupLockFromFH()` function performs a copy of a lock implicit to a *file handle* of an openend file. For this to succeed, the file must be opened in the mode `MODE_OLDFILE` or `MODE_READWRITE`. Files openend with `MODE_NEWFILE` are based on an implicit *exclusive lock* that cannot be copied.

```
lock = DupLockFromFH(fh)
D0                   D1

BPTR DupLockFromFH(BPTR)
```

This function returns a copy of the lock the *file handle* `fh` is based on and returns it in `lock`. In case of failure, `ZERO` is returned. In either case, `IoErr()` is set to either 0 in case of succes, or an error code on failure.

4.2.2 Obtaining the Directory a File is Located in

The `ParentOfFH()` function obtains a *shared lock* on the parent directory of the file associated to the *file handle* passed in. That is, it is roughly equivalent to first obtaining a lock on the file through `DupLockFromFile()` , and then calling `ParentDir()` on it, except that this function also applies to files opened in the `MODE_NEWFILE` mode.

```
lock = ParentOfFH(fh)
 D0               D1

BPTR ParentOfFH(BPTR)
```

This function returns in `lock` a shared lock on the directory containing the file opened through the `fh` *file handle*. It returns `ZERO` on failure. In either case, `IoErr()` is set, namely to 0 in case of success or to an error code on failure.

4.2.3 Opening a File from a Lock

The `OpenFromLock()` function uses a *lock* and opens the locked file, returning a *file handle*. If the lock is associated to a directory, the function fails. The *lock* passed in is then absorbed into the *file handle* and shall not be unlocked. It will be released by the file system upon closing the file.

```
fh = OpenFromLock(lock)
D0                  D1

BPTR OpenFromLock(BPTR)
```

This function attempts to open the object locked by `lock` as file, and creates the *file handle* `fh` from it. It fails in case the `lock` argument belongs to a directory and not a file.

In case of success, the *lock* becomes an implicit part of the *file handle* and shall not be unlocked by the caller anymore. In case of failure, the function returns `ZERO` and the *lock* remains available to the caller, and also needs to be unlocked at a later time. In either case, `IoErr()` is set, to an error code in case of failure, or 0 on success.

This function allows to open files in exclusive mode without deleting its contents. For that, obtain an *exclusive lock* on the file to be opened, and then call `OpenFromLock()` as second step.

4.2.4 The struct FileLock

Locks have been so far been opaque identifiers; in fact, they are *BPTR*s to a `struct FileLock` that is defined in `dos/dosextens.h`.

```
#include <dos/dosextens.h>
lock = Lock("S:Startup-Sequence",SHARED_LOCK);
struct FileLock *flock = BADDR(lock);
```

While this structure is defined there, it is not allocated by the *dos.library* but by the *file system* itself. The file system may therefore allocate a structure that is somewhat larger and can have additional members that are not shown here.

```
struct FileLock {
    BPTR                fl_Link;        /* bcpl pointer to next lock */
    LONG                fl_Key;         /* disk block number */
    LONG                fl_Access;      /* exclusive or shared */
    struct MsgPort *    fl_Task;        /* handler task's port */
    BPTR                fl_Volume;      /* bptr to DLT_VOLUME DosList entry */
};
```

Most of the members of this structure are of no practical value, and they should not be interpreted in any way. What is listed here is the information callers can depend upon.

The `fl_Link` member has no practical value for users; the *file system* can use it to keep multiple links on object on the same volume in a list.

The `fl_Key` member can be used by the file system to identify the object that has been locked. It may not necessarily be an integer, but can be any data type, potentially a pointer to some internal management object. It shall not be interpreted in any particular way.

The `fl_Access` member keeps the type of the lock. It is either `SHARED_LOCK` or `EXCLUSIVE_LOCK`.

The `fl_Task` member points to the message port of the file system for processing requests on the lock. Any activity on the lock goes through this port.

The `fl_Volume` is a *BPTR* to the *volume node* on the *Device list*. The *volume node* identifies the volume the locked object is located on. Section 6 provides further information on this list and its entries.

Chapter 5

Working with Directories

As objects on a file system can be identified by a name, these names need to be stored somewhere on the data carrier. This object is called a *directory*. While a flat file system only contains a single, topmost directory which then contains all files, a directory of a hierarchical file system can contain other directories, thus creating a *tree* of nested objects, see also section ??.

AmigaDOS provides functions to list the directory contents, to move objects in the file system hierarchy or change their name, and to access adjust their metadata, such as comments, protection bits, or creation dates.

AmigaDOS also supports *links*, that is, entries in the file system that point to some other object in the same, or some other file system. Therefore, links circumvent the hierarchy otherwise imposed by the tree structure of the file system.

5.1 Examining Objects on File Systems

Given a lock on a file or a directory, further information on such an object can be requested by the `Examine()` function of the *dos.library*. To read multiple directory entries at once and minimizing the calling overhead, `ExAll()` provides an advantage that is, however, harder to use, but also provides options to filter entries.

One general warning upfront: As AmigaDOS is a multitasking operating system, the directory may change under your feed while scanning; in particular, entries you received through the above functions may not be up to date, may have been deleted already when the above functions return, or new entries may have been added the current scan will not reach. While a *Lock* on a directory prevents that this directory goes away, it does *not* prevent other processes to add or remove objects to this directory, so beware.

While `ExAll()` seems to provide an advantage by reading multiple directory entries in one go, the AmigaOS ROM file system does usually not profit from this feature, at least not unless a directory cache is used. The latter has, however, other drawbacks and should be avoided for different reasons, see section ??. Actually, `ExAll()` is (even more) complex to implement, and it is probably not surprising that multiple file systems have issues. The `dos.library` provides an `ExAll()` implementation for those file systems that do not implement it themselves, but even this (ROM-based) implementation had issues in the past. Therefore, `ExAll()` has probably less to offer than it seems.

`Examine()` and `ExNext()` fill a `FileInfoBlock` structure that collects information on an examined object in a directory. It is defined in `dos/dos.h` and reads as follows:

```
struct FileInfoBlock {
   LONG   fib_DiskKey;
```

```
   LONG   fib_DirEntryType;  /* Type of Directory. If < 0, then a plain file.
                              * If > 0 a directory */
   char   fib_FileName[108]; /* Null terminated. Max 30 chars used for now */
   LONG   fib_Protection;    /* bit mask of protection, rwxd are 3-0.      */
   LONG   fib_EntryType;
   LONG   fib_Size;          /* Number of bytes in file */
   LONG   fib_NumBlocks;     /* Number of blocks in file */
   struct DateStamp fib_Date;/* Date file last changed */
   char   fib_Comment[80];  /* Null terminated comment associated with file */

   /* Note: the following fields are not supported by all filesystems.  */
   /* They should be initialized to 0 sending an ACTION_EXAMINE packet. */
   /* When Examine() is called, these are set to 0 for you.             */
   /* AllocDosObject() also initializes them to 0.                      */
   UWORD  fib_OwnerUID;         /* owner's UID */
   UWORD  fib_OwnerGID;         /* owner's GID */
   char   fib_Reserved[32];
}; /* FileInfoBlock */
```

The meaning of the members of this structure are as follows:

`fib_DiskKey` is a file system internal identifier of the object. It shall not be used, and programs shall not make any assumptions on its meaning.

`fib_DirEntryType` identifies the type of an object. Object types are defined in `dos/dosextens.h`, replicated in table 14:

Table 14: Directory Entry Types

Value of fib_DirEntryType	Description
ST_SOFTLINK	Object is a soft link to another object
ST_LINKDIR	Object is a hard link to a directory
ST_LINKFILE	Object is a hard link to a file

All other types > 0 indicate directories, and all other types < 0 indicate files. Section ?? provides more details on soft links and hard links.

`fib_FileName` is the name of the object as `NUL` terminated string.

`fib_Protection` are the protection bits of the object. It defines which operations can be performed on it. The following protection bits are currently defined in `dos/dos.h`:

Table 15: Protection Bits

Protection Bits	Description
FIBB_DELETE	If this bit is 0, the object can be deleted.
FIBB_EXECUTE	If this bit is 0, the file is an executable binary.
FIBB_WRITE	If this bit is 0, the file can be written to.
FIBB_READ	If this bit is 0, the file content can be read.
FIBB_ARCHIVE	This bit is set to 0 on every write access.
FIBB_PURE	If 1, the executable is reentrant and can be made resident.
FIBB_SCRIPT	If 1, the file is a script.
FIBB_HOLD	If 1, the executable is made resident on first execution.

The flags `FIBB_DELETE` to `FIBB_READ` are shown inverted in the output of most tools, i.e. they are shown active if the corresponding flag is 0, i.e. a particular protection function is *not* active.

The FIBB_EXECUTE flag is only interpreted by the *Shell* (see section ??); if the bit is 1, the *Shell* refuses to load the file as command.

The FIBB_ARCHIVE flag is typically used by archival software. Such software will set this flag upon archiving the flag, whereas the file system will reset the flag when writing to a file, or when creating new files. The archiving software is thus able to learn which files had been altered since the last backup.

The FIBB_PURE flag inidicates an additional property of executable binaries; if the flag is set, the binaries do not alter their segments and their code can be loaded in *RAM* and stay there to be executed from multple processes in parallel. This avoids loading the binary multiple times. The *Shell* command `resident` can load such binaries into *RAM* for future usage.

The FIBB_SCRIPT flag indicates whether a file is a *Shell* or an *ARexx* script. If this flag is set, and the script is given as command to the *Shell*, it will forward this file to a suitable script interpreter, such as *ARexx* or `Execute`.

The FIBB_HOLD flag indicates whether a command shall be made resident upon loading it the first time. If the flag is 1, and the shell loads the file as executable binary, and the FIBB_PURE bit is also set, the file is kept in *RAM* and stays there for future execution.

The fib_EntryType member shall not be used; it can be identical to the fib_DirEntryType, but its use is not documented.

The fib_Size member indicates the size of the file in bytes. It should have probably be defined as an unsigned type. Its value is undefined for directories.

The fib_NumBlocks member indicates now many blocks a file occupies on the storage medium, if such a concept applies. Disks and harddisk organize their storage into blocks of equal size, and the file system manages these blocks to store data on the medium. The number of blocks can be meaningless for directories.

The fib_Date member indicates when the file was changed last; depending on the file system, the date may also indicate when the last modification was made for a directory, such as creating or deleting a file within. Which operations exactly trigger a change of a directory is file system dependent. The DateStamp structure is specified in section ??.

The fib_Comment member contains a NUL terminated string to a comment on the file. Not all file systems support comments. The comment has no particular meaning, it is only shown by some *Shell* commands or utilities and can be set by the user.

The fib_OwnerUID and fib_OwnerGID are filled in by some multi-user aware file systems. The AmigaDOS ROM file systems do not support these fields, and no provision is made to moderate access to a particular file according to an owner or its group. The two concepts are alien to AmigaDOS itself.

The fib_Reserved field is currently unused and shall not be accessed.

5.1.1 Retrieving Information on an Directory Entry

The Examine() function retrieves information on the object identified by a *lock* and fills a FileInfoBlock from it.

```
success = Examine( lock, FileInfoBlock )
D0                 D1        D2

BOOL Examine(BPTR,struct FileInfoBlock *)
```

This function fills out the FileInfoBlock providing information on the object identified by lock. The structure is discussed in section 5.1 in more detail. The function returns non-zero in case of success, and 0 for failure. In either case, IoErr() is filled, by 0 on success, on an error code on failure.

Keep it Aligned! As with most BCPL structures, the `FileInfoBlock` shall be aligned to a long-word boundary. For that reason, it should be allocated from the heap. Section 2.3 provides some additional hints on how to allocate such structures.

5.1.2 Retrieving Information from a File Handle

While `Examine()` retrieves information a locked object, `ExamineFH()` retrieves the same information from a *file handle*, or rather from the *lock* implicit to the handle.

```
success = ExamineFH(fh, fib)
D0                  D1  D2

BOOL ExamineFH(BPTR, struct FileInfoBlock *)
```

This function examines the object accessed through the *file handle* `fh`, and returns the information in the *FileInfoBlock*. Note that the file content and thus its change can be changed any time, and thus the information returned by this function may not be fully up-to-date, see also the general information in section 5.1.

This function returns non-zero in case of success, or 0 on error. In either case, `IoErr()` is set, namely to 0 on success and to an error code otherwise.

As for `Examine()`, the *FileInfoBlock* shall be aligned to a 4-byte boundary.

5.1.3 Scanning through a Directory Step by Step

The `ExNext()` function iterates through entries of a directory, retrieving information on one object after another contained in this directory. For scanning through a directory, first `Lock()` the directory itself. Then use `Examine()` on the *lock*. This provides information on the directory itself.

To learn about the objects in the directory, iteratively call `ExNext()` on the same `lock` and on the same `FileInfoBlock` until the function returns `DOSFALSE`. Each iteration provides then information on the subsequent element in the directory of the `lock`.

```
success = ExNext( lock, FileInfoBlock )
D0                D1        D2

BOOL ExNext(BPTR, struct FileInfoBlock *)
```

This call returns information on the subsequent entry of a directory identified by `lock` and deposits this information in the `FileInfoBlock` described in 5.1. The `lock` shall be a *lock* on a directory, in particular.

On success, `ExNext()` returns non-zero. If there is no further element in the scanned directory, or on an error, it returns `DOSFALSE`. In either event, `IoErr()` is set, namely to 0 in case of success, or to an error code otherwise.

At the end of the directory, the function returns `DOSFALSE`, and the error code as obtained from `IoErr()` is set to `ERROR_NO_MORE_ENTRIES`.

Same Lock, Same FIB To iterate through a directory, a lock to the same directory as passed into `Examine()` shall be used. Actually, the same *lock* should be used, and the same *FileInfoBlock* should be used. As important state information is associated to the *lock* and *FileInfoBlock*, `UnLock()`ing the original *lock* and obtaining a new *lock* on the same directory looses this information; using a different *FileInfoBlock* also looses this state information, requiring the *file system* to rebuild this state information, which is not only complex, but also slows down scanning the directory. In particular, you shall *not* use the same *FileInfoBlock* you used for scanning one directory for scanning a second, different directory as this can confuse the *file system*. Also, as for `Examine()`, the `FileInfoBlock` shall be aligned to a long-word boundary.

5.1.4 Examine Multiple Entries at once

While scanning a directory with `ExNext()` requires one interaction with the *file system* for each entry and is therefore potentially slow, `ExAll()` retrieves as many entries as possible in one go. Whether a particular file system can take advantage of such a block transfer is a matter of its orginal organization, however.

```
continue = ExAll(lock, buffer, size, type, control)
D0               D1     D2     D3    D4     D5

BOOL ExAll(BPTR,STRPTR,LONG,LONG,struct ExAllControl *)
```

This function examines as many directory entries belonging to the directory identified by `lock` as fit into the buffer `buffer` of `size` bytes. This buffer is filled by a linked list of `ExAllData` structures, see below for their layout. `type` determines which elements of `ExAllData` is filled.

The `lock` shall belong to a directory for this function to succeed. It shall not be `ZERO`.

To start a directory scan with `ExAll()`, first allocate a `ExAllControl` structure through `AllocDosObject()`, see ??. This structure looks as follows:

```
struct ExAllControl {
        ULONG   eac_Entries;     /* number of entries returned in buffer      */
        ULONG   eac_LastKey;     /* Don't touch inbetween linked ExAll calls! */
        UBYTE  *eac_MatchString; /* wildcard string for pattern match or NULL */
        struct Hook *eac_MatchFunc; /* optional private wildcard function     */
};
```

`eac_Entries` is provided by the *file system* upon returning from `ExAll` and then contains the number of entries that fit into the `buffer`. Note that this number may well be 0, which does not need to indicate termination of the scan. Callers shall instead check the return code of `ExAll()` to learn on whether scanning may continue or not.

`eac_LastKey` is a *file system* internal identifier of the current state of the directory scanner. This member shall not be interpreted nor modified in any way.

`eac_MatchString` filters the directory entry names, and returns only those that match the wild card pointed to by this member. This entry shall be either `NULL`, or a pre-parsed pattern as generated by `ParsePatternNoCase()`.

`eac_MatchFunc` is a even more flexible option to filter directory entries. It shall be either `NULL` or point to a `struct Hook` as defined in `utility/hooks.h`. If set, then for each directory entry the hook function `h_Entry` is called as follows:

```
match = (hook->h_Entry)(struct Hook *hook, LONG *datap,
d0                                  a0          a2

                        struct ExAllData *buf )
                                          a1
```

that is, register `a0` points to the called hook, register `a1` to the data buffer to be filled, which is part of the `buffer` supplied by the caller of `ExAll()` and which is already filled in. Register `a2` points to a `LONG`, which is a copy of the `type` argument supplied to `ExAll()`. If the hook function returns non-zero, a match is assumed and the directory entry remains in the output buffer. Otherwise, the data is discarded.

`eac_MatchFunc` and `eac_MatchString` shall not be filled in simultaneously, only one of the two shall be non-`NULL`. If both members are `NULL`, all entries match.

The `buffer` supplied to `ExAll()` is filled by a singly linked list of `ExAllData` structures that look as follows:

```
struct ExAllData {
        struct ExAllData *ed_Next;
        UBYTE  *ed_Name;
        LONG    ed_Type;
        ULONG   ed_Size;
        ULONG   ed_Prot;
        ULONG   ed_Days;
        ULONG   ed_Mins;
        ULONG   ed_Ticks;
        UBYTE  *ed_Comment;     /* strings will be after last used field */
        UWORD   ed_OwnerUID;    /* new for V39 */
        UWORD   ed_OwnerGID;
};
```

The members of this structure are as follows:

`ed_Next` points to the next `ExAllData` structure within `buffer`, or `NULL` for the last structure filled in.

`ed_Name` points to the file name of a directory entry, and supplies the same name as `fib_FileName` as in the `FileInfoBlock`.

`ed_Type` identifies the type of the entry. It identifies directory entries according to table 14 and corresponds to `fib_DirEntryType`.

`ed_Size` is the size of the directory element for files. It is undefined for directories. It corresponds to `fib_Size`.

`ed_Prot` collects the protection bits of the directory entry according to table 15 and by that corresponds to `fib_Protection`.

`ed_Days`, `ed_Mins` and `ed_Ticks` identifies the date of the last change to the directory element. It corresponds to `fib_Date`. Section 5.2.4 defines these elements more rigorously.

`ed_Comment` points to a potential comment on the directory entry and corresponds to `fib_Comment`.

`ed_ed_OwnerUID` and `ed_OwnerGID` contain potential user and group IDs if the file system is able to provide such information. All the AmigaDOS native file systems do not.

Which members of the `ExAllData` structure are filled in is selected by the `type` argument. It shall be selected according to table 16, whose elements are defined in `dos/exall.h`:

Table 16: Type Values

Type	Filled Members
`ED_NAME`	Fill only `ed_Next` and `ed_Name`
`ED_TYPE`	Fill all members up to `ed_Type`
`ED_SIZE`	Fill all members up to `ed_Size`
`ED_PROTECTION`	Fill all members up to `ed_Prot`
`ED_DATE`	Fill all members up to `ed_Ticks`, i.e. up to the date
`ED_COMMENT`	Fill all members up to `ed_Comment`
`ED_OWNER`	Fill all members up to `ed_OwnerGID`

The return code `continue` is non-zero in case the directory contents was too large to fit into the supplied `buffer` completely. In such a case, either `ExAll()` shall be called again to read additional entries, or `ExAllEnd()` shall be called to terminate the call and release all internal state information.

If `ExAll()` is called again, the `lock` shall be identical to the `lock` passed into the first call, and not only a copy on the same directory as for the first call.

The return code `continue` is `DOSFALSE` in case the scan result fit entirely into `buffer` or in case an error occured.

Regardless of the return code, `IoErr()` is set to 0 in case `continue` is non-zero, or to an error code otherwise. If the error code is `ERROR_NO_MORE_ENTRIES`, then `ExAll()` terminated because all

entries have been read and scanning the directory completed. In this case, `ExAllEnd()` should not be called.

Not all file systems — actually, none delivered with AmigaOs — support `ED_OWNER`. If `continue` is `DOSFALSE` and `IoErr()` is `ERROR_BAD_NUMBER`, try to reduce `type` and call `ExAll()` again.

Some file systems do not implement `ExAll()` themselves; in such a case, the *dos.library* provides a fall-back implementation keeping `ExAll()` workable regardless of the completeness of the target *file system*.

5.1.5 Aborting a Directory Scan

To abort an `ExAll()` scan through a directory, `ExAllEnd()` shall be called to explicitly release all state information associated to the scan. This is unlike an item-by-item scan through `ExNext()` which does not require explicit termination.

```
ExAllEnd(lock, buffer, size, type, control)
          D1     D2     D3    D4     D5

void ExAllEnd(BPTR,STRPTR,LONG,LONG,struct ExAllControl *)
```

This function aborts an `ExAll()` driven directory scan before it terminates due to an error or due to the end of the directory, i.e. whenever `ExAll()` returns with a non-zero result code which would indicate that the function should be called again.

`ExAll()` may also be the fastest way to terminate a directory scan once it is running, for example on network file systems where the scan may proceed offline on a separate server. The arguments to `ExAllEnd()` shall be exactly those supplied to `ExAll()` which it is supposed to terminate. Note in particular that the `lock` shall be identical to the *lock* passed into `ExAll()`, and not just a *lock* to the same object.

5.2 Modifying Directory Entries

While the functions in section 5.1 read directory entries, the functions listed here modify the directory and its entries.

5.2.1 Deleting Objects on the File System

The `DeleteFile()` function removes — despite its name — not only files, but also directories and links from a directory. For this to succeed, the object need to allow deletion through its protection bits (see section 5.1), and no *locks* are held on the object (see section 4). To be able to delete a directory, this directory needs to be empty in addition.

```
success = DeleteFile( name )
 D0                   D1

BOOL DeleteFile(STRPTR)
```

This function deletes the object given by the last component of the path passed in as `name`. It returns non-zero in case of success, or 0 in case of error. In either case, `IoErr()` is set, namely 0 on success or an error code in case of failure.

5.2.2 Rename or Relocate an Object

The `Rename()` function changes the name of an object, or even relocates it from one directory to another.

```
success = Rename( oldName, newName )
  D0                D1       D2

BOOL Rename(STRPTR, STRPTR)
```

This function renames and optionally relocates an object between directories. The `oldName` is the current path to the object, and its last component is the current name of the object to relocate and rename; `newName` is the target path and its last component the target name of the object. The target directory may be different from the directory the object is currently located in, and the target name may be different from the current name. However, current path and target path shall be on the same volume, and the target directory shall not already contain an object of the target name; otherwise, current and target path may be either relative or absolute paths.

A third condition is that if the object to relocate is a directory, then the target path shall not be a position within the object to relocate, i.e. you cannot move a directory into itself.

This function returns a boolean success indicator. It is non-zero on success, or 0 on error. In either case, `IoErr()` is set, to 0 on success, or to an error code otherwise.

5.2.3 Set the File Comment

The `SetComment()` function sets the comment of an directory entry, provided the *file system* supports comments.

```
success = SetComment( name, comment )
D0                    D1    D2

BOOL SetComment(STRPTR, STRPTR)
```

This function sets the comment of the *file system* object whose path is given by `name` to `comment`. It depends on the file system whether or how long comments can grow. The maximum comment length AmigaDOS supports is 79 characters, due to the available space in the `FileInfoBlock` structure.

This function returns non-zero on success and 0 on error. In either case, the function sets `IoErr()` to 0 on success or to an error code otherwise.

5.2.4 Set the Modification Date

The `SetFileDate()` function sets the modification date of an object of a *file system*. Despite its name, the function can also set the modification date of directories and links if the file system supports them.

```
success = SetFileDate(name, date)
D0                     D1    D2

BOOL SetFileDate(STRPTR, struct DateStamp *)
```

This function adjusts the modification date of the *file system* object identified by path as given by `name` to `date`. The `DateStamp` structure is specified in section ??.

This function returns 0 on error or non-zero on success. In either case, `IoErr()` is set, either to 0 on success or to an error code otherwise.

Note that not all file systems may be able to set the date precisely to ticks, e.g. `FAT` has only a precision of 2 seconds. Some file systems may refuse to set the modification date if an object is exclusively locked, this is unfortunately not handled consistently.

5.2.5 Set User and Group ID

The `SetOwner()` function sets the user and group ID of an object within a *file system*. Both are concatenated to a 32-bit ID value. While this function seems to imply that the file system or AmigaDOS seems to offer some multi-user capability, this is not the case. User and group ID are purely metadata that is returned by the functions discussed in section 5.1, they usually ignore them. AmigaDOS has no concept of the current user of a *file system* and thus cannot decide whether a user is priviledged to access an object on a file system. In fact, all ROM based file systems delivered with AmigaDOS do not support setting the user or group ID.

```
success = SetOwner( name, owner_info )
D0                  D1       D2

BOOL SetOwner (STRPTR, LONG)
```

This function sets the user and group ID of the *file system* object identified by the path in `name` to the value `owner_info`. How exactly the `owner_info` is encoded is *file system* specific. Typically, the owner is encoded in the topmost 16 bits, and the group in the least significant 16 bits.

This function returns a boolean success indicator which is non-zero on success and 0 on error. This function always sets `IoErr()`, either to 0 on success or to an error code otherwise.

5.3 Working with Paths

The *dos.library* contains a couple of support functions that help working with paths, see also section 3.3. What is different from the remaining functions is that the paths are not interpreted by the file system, but rather by the *dos.library* itself. This has several consequences: First, there is no 255 character limit as the path is never communicated into the *file system* as it was stated in section 3.3.3. Second, as the paths are constructed or interpreted by the library and not the *file system*, the syntax of the path is also that imposed by the library.

That is, for these functions to work, the separator between component must be the forwards slash ('/') and the parent directory must be indicated by an isolated single forward slash without a component upfront. This implies, in particular, that the involved file systems follow the conventions of AmigaDOS.

5.3.1 Find the Path From a Lock

The `NameFromLock()` function constructs a path to the locked object, i.e. if the constructed path is used to create a lock, it will refer to the same object.

```
success = NameFromLock(lock, buffer, len)
D0                     D1     D2    D3

BOOL NameFromLock(BPTR, STRPTR, LONG)
```

This function constructs in `buffer` an absolute path that identifies the object locked by `lock`. At most `len` bytes will be filled into `buffer`, including `NUL` termination of the string. The created string is always `NUL`-terminated, even if the buffer is too short. However, in such a case the function returns 0, and `IoErr()` is set to `ERROR_LINE_TOO_LONG`.

If the path cannot be constructed due to an error, `success` is also set to 0 and `IoErr()` is set to an error code. However, on success, `IoErr()` is not set consistently and cannot be depended upon. Possible cases of failure are that the volume the locked object is located on is currently not inserted in which case it will be requested. The `ZERO` lock is correctly interpreted, and resolves into the string `SYS:`. The `lock` remains valid after the call.

5.3.2 Append a Component to a Path

The `AddPart()` adds an absolute or relative path to an existing path; the resulting path is constructed as if the input path is a directory, and the attached (second) path identifies an object relative to this given directory. The function handles special cases such as the colon (':') and one or multiple leading slashes ('/') correctly and are interpreted according to the rules explained in section 3.3: The colon identifies the root of the volume, and a leading slash the parent directory, upon which the trailing component of the input path is removed.

```
success = AddPart( dirname, filename, size )
D0                 D1        D2      D3

BOOL AddPart( STRPTR, STRPTR, ULONG )
```

This function attaches to the existing path in `dirname` another path in `filename`. The constructed path will overwrite the buffer in `dirname`, which is able to hold `size` bytes, including a terminating `NUL` byte.

If the required buffer for the constructed path, including termination, is larger than `size` bytes, then the function returns 0 and `IoErr()` is set to `ERROR_LINE_TOO_LONG`, and the input buffers are not altered. Otherwise, the function returns non-zero, and `IoErr()` is not altered.

This function does not interact with a *file system* and does not check whether the paths passed in correspond to accessible objects. The output path is constructed purely based on the AmigaDOS syntax of paths.

5.3.3 Find the last Component of a Path

The `FilePart()` function finds the last component of a path; the function name is a bit misleading since the last component does not necessarily correspond to a file, but could also correspond to a directory once identified by a *file system*. If there is only a single component in the path passed in, this component is returned. If the path passed in terminates with at least two slashes ('/') indicating that the last component is at least one level above, a pointer to the terminating slash is returned.

```
fileptr = FilePart( path )
D0                  D1

STRPTR FilePart( STRPTR )
```

This function returns in `fileptr` a pointer to the last component of the path passed in as `path`, or a pointer to '/' in case the input path terminates with at least two slashes.

This function cannot fail, and does not touch `IoErr()`.

5.3.4 Find End of Next-to-Last Component in a Path

The `PathPart()` identifies the end of the next-to-last component in a path. That is, if a `NUL` is injected at the pointer returned by this function, the resulting string starting at the passed in buffer corresponds to a path that corresponds to the directory containing the last component of the path. If the passed in path consists only of a single component, the returned pointer is identical to the pointer passed in.

```
fileptr = PathPart( path )
D0                  D1

STRPTR PathPart( STRPTR )
```

This function returns in `fileptr` a pointer to the end of the next-to-last component of the `path` passed in. This function cannot fail and does not alter `IoErr()`.

The only difference between this function and `FileParth()` is that the latter advanced over a potential trailing slash. That is, if the last character of the input path of `PathPart()` would be a slash, then `PathPart()` would return a pointer to this slash, but `FilePart()` would advance beyond this slash. That is, the "file part" of a path that explicitly indicates a directory is empty, though the "path part" is the same path without the trailing slash.

5.4 Links

Links are tools to escape the tree-like hierarchy of directories, sub-directories and files. A *link* mirrors one object of a file system to another location such that if the object is changed using the path of one location, the changes are reflected in another location. Put differently, creating a link is like copying an object except that copy and original are always in sync. The storage for the payload data of a file is only required once, the link just points to the same data as the original directory entry. The same goes for links between directories: Whenever a new entry is made in one directory, the change also appears in the other.

AmigaDOS supports two (or, actually, three) types of links: *Hard-links* and *Soft-links*. The *RAM-Handler* supports a third type that will be discussed below. *Hard-links* establish the relation between two *file-system* objects on the same volume at the level of the file system. That is, whenever a link is accessed, the file system resolves the link, transparent to its user. While for the Amiga *Fast File System* and the *RAM-Handler* a *hard-link* is a distinct directory entry type, some file systems do not distinguish between the original object and a *hard-link* to it. For such file systems, the same payload data is just referenced by two directory entries. If the larget of a link is deleted on the *Fast File System* or the *RAM-Handler*, and (at least one) link to the object still exists, then (one of) the link(s) takes over and becomes the object itself. For other file systems, only a file system internal reference counter is decreased, and the payload data is removed only if this counter becomes zero.

Soft-Links work differently and can also be established between two different *file systems*, or between two different volumes. Here, the *soft-link* is a type of its own that contains the path of the referenced object. If such a *soft-link* is accessed, an error code is reported by the *file-system* and it is then up to a higher layer such as the *dos.library* or an application program to read the link destination, and use it to create a path from the original path and the link destination. The access is then (hopefully) retried under the updated path. As this object may also be a *soft-link*, this process can continue; in worst case, indefinitely if one link refers to another in a circular way. To avoid this situation, the *dos.library* follows at most 15 links.

The *dos.library* supports *Soft-Links* through the functions listed in Table 17:

Table 17: Softlink aware functions

Function	Purpose
`Open()`	Open a file
`Lock()`	Obtain access rights to an object
`CreateDir()`	Create a directory
`SetProtection()`	Modify protection bits
`SetFileDate()`	Set the modification date of a file
`DeleteFile()`	Delete an object on a file system
`SetComment()`	Modify object comment
`MakeLink()`	Create a link to an object
`SetOwner()`	Set User and Group ID

All of the above functions take a path of its first argument. If the path consists of multiple components, i.e. identifies an object in a nested directory, and one of the intermediate components are, in fact, *soft-links*, the *dos.library* will automatically resolve such an intermediate link and construct internally the true path to the link destination. Whether a soft-link at the last component is resolved is typically *file system* and function dependent. For example, `Open()` will always resolve *soft-links*, but `Lock()` or `SetProtection()` may not and may instead affect the link, not the target object. `DeleteFile()` will never resolve a link at the final component of the path, and will therefore delete the link, not the object linked to.

If the target of a *Soft-Link* is deleted (and not the link itself), a link pointing to it becomes invalid, even though remains in the *file system*. Any attempt to resolve the link then, obviously, fails. AmigaDOS does not attempt to identify such invalid links. The same cannot happen for *hard-links*.

Finally, the *RAM-Handler* supports a special type of *hard-links* that goes across volumes. These *external links* copy the linked object on a read-access into the RAM disk, i.e. the *RAM-Handler* implements a *copy on access*. This feature is used for the `ENV:` assign containing all active system settings. This assign points to a directory in the RAM disk which itself is externally linked to `ENVARC:`. Thus, whenever a program attempts to access its settings — such as the preferences programs — the *RAM-Handler* automatically copies the data from `ENVARC:` to `ENV:`, avoiding a manual copy and also saving RAM space for settings that are currently not accessed and thus unused.

The `FileInfoBlock` introduced in section 5.1 identifies links through the `fib_DirEntryType` member. As seen from table 14, *hard-links* to files are indicated by `ST_LINKFILE` and *hard-links* to directories by `ST_LINKDIR`. Note, however, that not all file systems are able to distinguish *hard-links* from regular directory entries, so this feature cannot be dependened upon. In particular, *external links* of the *RAM-Handler* cannot be identified by any particular value of the `fib_DirEntryType`.

Table 14 also provides the `fib_DirEntryType` for *soft-links*, namely `ST_SOFTLINK`. As the target of a *soft-link* may not under control of the *file system*, it cannot know whether the link target is a file or a directory (or maybe another link), and therefore a single type is sufficient to identify them.

5.4.1 Creating Links

The `MakeLink()` function creates a *hard-link* or a *soft-link* to an existing object on a *file system*.

```
success = MakeLink( name, dest, soft )
D0                  D1    D2    D3

BOOL MakeLink( STRPTR, LONG, LONG )
```

This function creates a new link at the path `name` of the type given by `soft`. The destination the link points to is given by `dest`.

If `soft` is `FALSE`, `dest` is a *lock* represented by `BPTR`. For most *file systems*, `dest` shall be on the same volume as the one identified by the path in `name`. The currently only exception is the *RAM-Handler* for which the destination *lock* may be on a different volume. In such a case, an *external link* is created. While the target object will be created, it may look initially like an empty file or an empty directory, depending on the type of the link destination. Its contents is copied, potentially recursively creating directories, by copying the contents of the link destination into the link, or to a file or directory within the link. Thus, the link becomes a mirror of the link destination whenever an object within the link or the link itself is accessed.

If `soft` is non-zero, `dest` is a `const UBYTE *` that shall be casted to a `LONG`. Then, this function creates a *soft-link* that is relative to the path of the link, i.e. `name`. For details on *soft-link* resolution, see section ??.

This function returns in `success` non-zero if creation of the lock succeeded, or 0 in case of failure. In either case, `IoErr()` is set to an error code on failure, or 0 on success.

5.4.2 Resolving Soft-Links

The `ReadLink()` function locates the destination of a *soft-link* and constructs from the path and directory of the link a new path that identifies the target of the link. A typical use case for this function is if a *dos.library* function returns with the error `ERROR_IS_SOFT_LINK`, indicating that the *file system* needs help from a higher layer to grant access to the object. You then typically retry the access to the object with the path constructed by this function. Note well that this path may be that of yet another *soft-link*, requiring recursive resolution of the link. To avoid endless recursion, this loop should be aborted after a maximum number of attempts, then generating an error such as `ERROR_TOO_MANY_LEVELS`. A suggested maximum level of nested *soft-links*, also used by the *dos.library*, is 15 links.

Note, however, that such steps would not be necessary for the functions listed in table 17 as they already perform such steps internally.

```
success = ReadLink( port, lock, path, buffer, size)
D0                  D1    D2    D3     D4     D5

BOOL ReadLink( struct MsgPort *, BPTR, STRPTR, STRPTR, ULONG)
```

This function creates in `buffer` of `size` bytes a path to the target of a *soft-link* contained in the input `path` relative to the directory represented by `lock`. Typically, `path` is the path given to some object you attempted to access, and `lock` is the *lock* as given by the current directory to which the path is relative. The output path constructed in `buffer` is then an updated path relative to the same directory, i.e. relative to `lock`.

The `port` is the message port of the file system that is queried to resolve the *soft-link*; this port should be obtained from `GetDeviceProc()`, see section ??. For relative paths, this port is identical to the one in the `fl_Task` member of the `FileLock` structure representing `lock`, see section 4.2.4.

If `size` is too small to hold the adjusted path, the function returns 0 and sets `IoErr()` to `ERROR_LINE_TOO_LONG`.

The function returns non-zero in case of success, or 0 in case of error. In either case, `IoErr()` is set to ether 0 on succes, or an error code otherwise.

Chapter 6

Administration of Volumes, Devices and Assigns

The *dos.library* is just a layer of AmigaDOS that provides a common API for input/output operations; these operations are not implemented by the library itself, but forwarded to *file systems* or *handlers*. This forwarding is based on the exec *message* and *message port* system, and to this end, the `FileLock` structure and the `FileHandle` structure contain a pointer to a `MsgPort`.

However, the *dos.library* also needs to obtain this port from somewhere; for relative paths (see section 3.3), the current directory (see section ??) provides it. For absolute paths, i.e. paths that contain a colon (':'), the string upfront the colon identifies handler, directly or indirectly. If this string is empty, i.e. the path starts with a colon, it is again the handler of the current directory that is contacted, but otherwise, the dos searches the *device list* to find a suitable *message port*. This algorithm is also available as a function, namely `GetDeviceProc()`, which is documented in section ??.

Internally, the *dos.library* keeps the relation between such names and the corresponding ports in the `DosList` structure. Such a structure is also created when *mounting* a handler, i.e. advertizing the handler to the system, or when creating an *Assign*, see section 3.3.1.3, or when inserting a disk into a drive, thus making a particular *volume* available to the system (see also 3.3.1.2). Only the names from table 3 in 3.3.1.1 are special cases and hard-coded into the *dos.library* without requiring an entry in the *device list* in the form of a `DosList` structure.

This structure, defined in `dos/dosextens.h` reads as follows:

```
struct DosList {
    BPTR                dol_Next;       /* bptr to next device on list */
    LONG                dol_Type;       /* see DLT below */
    struct MsgPort     *dol_Task;       /* ptr to handler task */
    BPTR                dol_Lock;
    union {
        struct {
        BSTR    dol_Handler;    /* file name to load if seglist is null */
        LONG    dol_StackSize;  /* stacksize to use when starting process */
        LONG    dol_Priority;   /* task priority when starting process */
        ULONG   dol_Startup;    /* startup msg: FileSysStartupMsg for disks */
        BPTR    dol_SegList;    /* already loaded code for new task */
        BPTR    dol_GlobVec;    /* BCPL global vector to use when starting
                                 * a process. -1 indicates a C/Assembler
                                 * program. */
```

```
        } dol_handler;

        struct {
        struct DateStamp        dol_VolumeDate;  /* creation date */
        BPTR                    dol_LockList;    /* outstanding locks */
        LONG                    dol_DiskType;    /* 'DOS', etc */
        } dol_volume;

        struct {
        UBYTE   *dol_AssignName;     /* name for non-or-late-binding assign */
        struct AssignList *dol_List; /* for multi-directory assigns (regular) */
        } dol_assign;

    } dol_misc;

    BSTR                dol_Name;        /* bptr to bcpl name */
};
```

and its members have the following purpose:

`dol_Next` is a *BPTR* to the corresponding next entry in a singly linked list of `DosList` structures. However, this list should not be walked manually, but instead `FindDosEntry()` should be used for iterating through this list.

`dol_Type` identifies the type of the entry, and by that also the layout of the structure, i.e. which members of the unions are used. The following types are defined in `dos/dosextens.h`:

Table 18: *DosList* Entry Types

dol_Type	Description
DLT_DEVICE	A *file system* or *handler*, see 3.3.1.1
DLT_DIRECTORY	A regular assign, see 3.3.1.3
DLT_VOLUME	A volume, see 3.3.1.2
DLT_LATE	A late binding assign, see 3.3.1.3
DLT_NONBINDING	A non-binding assign, see 3.3.1.3

`dol_Task` is the *MsgPort* of the handler to contact for the particular *handler*, *assign* or *volume*. It may be `NULL` if the *handler* is not started, or a new handler process is supposed to be started for each file opened. This is, for example, the case for the console which requires a process for each window it handles. *File systems* usually provide their port here such that the same process is used for all objects on the volume. *Volumes* keep here the *MsgPort* of the *file system* that operates the volume, but it to `NULL` in case the volume goes away, e.g. is ejected. For *regular assigns*, this is also the pointer to the *MsgPort* of the *file system* the assign binds to; in case the assign is a *multi-assign*, this is the *MsgPort* of the first directory bound to. All additional ports are part of the `AssignList`. For *late assigns* this member is initially `NULL`, but will be filled in as soon as the assign in bound to a particular directory, and then becomes the pointer to the *MsgPort* of the handler the assign is bound to. Finally, for *non-binding assigns* this member always stays `NULL`.

`dol_Lock` is only used for *assigns*, and only if it is bound to a particular directory. That is, the member remains `ZERO` for *non-binding assigns* and is initially `ZERO` for *late assigns*. For all other types, this member stays `ZERO`.

`dol_Name` is a *BPTR* to a *BSTR* is the name under which the *handler*, *volume* or *assign* is accessed. That is, this string corresponds to the path component upfront the colon.

The members within `dol_handler` are used by *handlers* and *file systems*, i.e. if `dol_Type` is `DLT_DEVICE`.

`dol_Handler` is a *BPTR* to a *BSTR* containing the file name from which the *handler* or *file system* is loaded from. It corresponds to the `Handler`, `FileSystem` and `EHandler` fields of the mount list. They all deposit the file name here.

`dol_StackSize` specifies the size of the stack for creating the *handler* or *file system* process. Interestingly, the unit of the stack size depends on the `dol_GlobVec` entry. If `dol_GlobVec` is negative indicating a C or assembler handler, `dol_StackSize` is in bytes. Otherwise, that is, for BCPL handlers, it is in 32-bit long words. This member corresponds to the `Stacksize` entry of the mount list.

`dol_Priority` is priority of the handler process. Even though it is a `LONG`, it shall be a number between −128 and 127 because priorities of the exec task scheduler are `BYTE`s. For all practical purposes, the priority should be a value between 0 and 19. It corresponds to the `Priority` entry of the mount list.

`dol_Startup` is a handler-specific startup value that is used to commumicate a configuration to the handler during startup. While this value may be whatever the handler requires, the `mount` command either deposits here a small integer, or a pointer to the `FileSysStartupMsg` structure defined in `dos/filehandler.h`. Section ?? provides more details on mounting handlers and how the startup mechanism works. Unfortunately, it is hard to interpret `dol_Startup` correctly, see ??. One way to set this member is to set `Startup` in the mount list, see ?? for details.

`dol_SegList` is a *BPTR* to the chained segment list of the handler if it is loaded. For disk-based handlers, this member is initially `ZERO`. When a program attempts to access a file on the handler, the *dos.library* first checks whether this field is `ZERO`, and if so, attempts to find a segment, i.e. a binary, for the handler. If the `FORCELOAD` entry of the mount list is non-zero, the `mount` command already performs this activity. The process of loading a handler depends on the nature of the handler and explained in more detail in section ??.

`dol_GlobVec` identifies the nature of the handler as AmigaDOS supports (still) BPCL and C/assembler handlers and defines how access to the *dos list* is secured for handler loading and startup. BCPL handlers use a somewhat more complex loading and linking mechanism as the language-specific *global vector* needs to be populated. This is not required for C or assembler handlers where a simpler mechanism is sufficient, more on this in section ??. Another aspect of the startup process is how the *device list* is protected from conflicting accesses from multiple processes. Two types of access protection are possible: Exclusive access to the list, or shared access to the list. Exclusive access protects the *device list* from any changes while the handler is loaded and until handler startup completed. This prevents any other modification to the list, but also read access from any other process to the list. Shared access allows read accesses to the list while preventing exclusive access to it.

The value in `dol_GlobVec` corresponds to the `GlobVec` entry in the mount list. It shall be one of the values in table 19.

Table 19: GlobVec Values

dol_Type	Description
`-1`	C/assembler handler, exclusive lock on the *dos list*
`-2`	C/assembler handler, shared lock on the *dos list*
`0`	BCPL handler using system GV, exclusive lock on the *dos list*
`-3`	BCPL handler using system GV, shared lock on the *dos list*
`>0`	BPCL handler with custom GV, exclusive lock on the *dos list*

The values 0, −3 and > 0 all setup a BCPL handler, but differ in the access type to the *device list* and how the BCPL *global vector* is populated. This vector contains all global objects and all globally reachable functions of a BCPL program, including functions of the *dos.library*. The values 0 and −3 fill this vector with the system functions first, and then use the BPCL binding mechanism to extend or override entries in this vector with the values found in the loaded code. Any values > 0 defines a *BPTR* to a custom vector which is used instead for initializing the handler. This startup

mechanism has never been used in AmigaDOS and is not quite practical as this vector needs to be communicated into the *dos.library* somehow. For new code, BCPL linkage and binding should not be used anymore.

Members of the `dol_volume` structure are used if `dol_Type` is `DLT_VOLUME`, identifying this entry as belonging to a known specific data carrier.

`dol_VolumeDate` is the creation date of the volume. It is a `DateStamp??` structure that is specified in section ??. It is used to uniquely identify the volume, and to distinguish this volume from any other volume of the same name.

`dol_LockList` is a pointer to a singly-linked list of *locks* on the volume. This list is created by the *file system* when the volume is ejected, and contains all locks on this volume. It is stored here to allow a similar file system to pick up the locks once the volume is re-inserted, even if it is re-inserted into another device. Note that the linkage is performed with *BPTR*s and the `fl_Link` member of the `FileLock` structure.

`dol_DiskType` is an identifier of the *file system type* that operated the volume and placed here such that an alternative process of the same file system is able to pick up or refuse the locks stored here for non-available volumes.

Members of the `dol_assign` structure are used for all other types, i.e. all types of *assigns*.

`dol_AssignName` is pointer to the target name of the assign for *non-binding* and *late assigns*. The *dos.library* uses this string to locate the target of the assign. For *late assigns*, this member is used only on the first attempt to access the assign at which `dol_Lock` is populated.

`dol_List` contains additional locks for *multi-assigns* and is only used if `dol_Type` is `DLT_DIRECTORY`. In such a case, `dol_Lock` is the lock to the first directory of the *multi-assign*, while `dol_List` contains all following *locks* in a singly-linked list of `AssignList` structures:

```
struct AssignList {
        struct AssignList *al_Next;
        BPTR               al_Lock;
};
```

`al_Next` points to the next *lock* that is part of the *multi-assign* and `al_Lock` is the lock itself. This structure is also defined in `dos/dosextens.h`.

6.1 Finding Handler or File System Ports

The following functions find the *MsgPort* of the *handler* or *file system* that is responsible for a given object. The functions search the *device list*, check whether the handler is already loaded or load it if necessary, then check whether the handler is already running, and if not, launch another instance of it. If *multi-assigns* are involved, it can become necessary to contact multiple *file systems* to resolve the task and thus to iterate through multiple potential *file systems* to find the right one.

6.1.1 Iterate through Devices Matching a Path

The `GetDeviceProc()` find a handler, or the next handler responsible for a given path. Once the handler has been identified, or iteration through matching handlers is to be aborted, `FreeDeviceProc()` shall be called to release temporary resources.

```
devproc = GetDeviceProc(name, devproc)
  D0                    D1     D2

struct DevProc *GetDeviceProc(STRPTR, struct DevProc *)
```

This function takes a path in `name` and either `NULL` on the first iteration or a `DevProc` structure from a previous iteration and returns either a `DevProc` structure in case a matching handler could be identified, or `NULL` if no matching handler could be found or all possible matches have been iterated over already already.

The `DevProc` structure, defined in `dos/dosextens.h` looks as follows:

```
struct DevProc {
        struct MsgPort *dvp_Port;
        BPTR            dvp_Lock;
        ULONG           dvp_Flags;
        struct DosList *dvp_DevNode;    /* DON'T TOUCH OR USE! */
};
```

`dvp_Port` is a pointer to a candidate *MsgPort* that should be tried to resolve `name`.

If the matching handler is a *file system*, then `dvp_Lock` is a *lock* of a directory. The path in `name`is a path relative to this directory. This *lock* shall not be released, but it may be copied with `DupLock`.

`dvp_Flags` identifies the nature of the found port. If the bit `DVPB_ASSIGN` is set, i.e `dvp_Flags & DVPF_ASSIGN` is non-zero, then the found match is part of a *multi-assign* and `GetDeviceProc()` may be called again with the `devproc` argument just returned as second argument. This will return another candidate for a path. `DVPB_UNLOCK` is another bit of the flags but shall not be interpreted and is only used internally by the function.

The member `dvp_DevNode` shall not be touched or used and is required internally by the function.

If the function returns `NULL`, then `IoErr()` provides additional information on the failure. If the error code is `ERROR_NO_MORE_ENTRIES`, then the last directory of a *multi-assign* has been reached. If the error code is `ERROR_DEVICE_NOT_MOUNTED`, then no matching device could be found. Other errors may be returned, e.g. if the function could not allocate sufficient memory for its operation.

Unfortunately, the function does not set `IoErr()` consistently if `GetDeviceProc()` is called again on an existing `DevProc` structure as second argument with `DVPB_ASSIGN` cleared. `IoErr()` remains then unaltered and it is therefore advisable to clear it upfront.

The function also returns `NULL` if `name` corresponds to the `NIL:` pseudo-device and then sets `IoErr()` to `ERROR_DEVICE_NOT_MOUNTED`. This is not fully correct, and callers need to be aware of this defect.

Also, `GetDeviceProc` does not handle the path "`*`" at all, even though it indicates the current console and the *Console-Handler* is responsible for it. This case also needs to be detected by the caller, and in such a case, `GetConsoleTask()` delivers the correct port.

6.1.2 Releasing DevProc Information

The `FreeDeviceProc()` function releases a `DevProc` structure previously returned by `GetDeviceProc()` and releases all temporary resources allocated by this function. It shall be called as soon as the `DevProc` structure is no longer needed.

```
FreeDeviceProc(devproc)
               D1

void FreeDeviceProc(struct DevProc *)
```

This function releases the `DevProc` structure and all its resources from an iteration through one or multiple `GetDeviceProc()` calls. If `GetDeviceProc()` returned `NULL` itself it had already released such resources itself and no further activity is necessary.

The `dvp_Port` or `dvp_Lock` within the `DevProc` structure shall not be used after releasing it with `FreeDeviceProc()`. If a *lock* is needed afterwards, a copy of `dvp_Lock` shall be made with

DupLock(). If the port of the *handler* or *file system* is needed afterwards, a resource of this handler shall be obtained, e.g. by opening a file or obtaining a lock on it. Both the FileHandle and the FileLock structures contain a pointer to the port of the corresponding handler.

It is safe to call FreeDeviceProc() with a NULL argument; this performs no activity.

This function does not set IoErr() consistently and no particular value may be assumed. It may or may not alter its value.

6.1.3 Legacy Handler Port Access

The DeviceProc() function is a legacy variant of GetDeviceProc() that should not be used anymore. It is not able to reliably provide locks to *assigns* and will not work through all directories of a *multi-assign*.

```
process = DeviceProc( name )
 D0                   D1

struct MsgPort *DeviceProc (STRPTR)
```

This function returns a pointer to a port of a *handler* or *file system*able to handle the path name. It returns NULL on error in which case it sets IoErr().

If the passed in name is part of an *assign*, the handler port of the directory the assign binds to is returned, and IoErr() is set to the *lock* of the assign. Unfortunately, one cannot safely make use of this *lock* as the *device list* may be altered any time, including the time between the return from this function and its first use by the caller. Thus, GetDeviceProc() shall be used instead which locks resources such as the *device list*; they are released through FreeDeviceProc().

This function does not operate properly on *multi-assigns* where it only provides the port and *lock* to the first directory participating in the assign. It also returns NULL for *non-binding assigns* as there is no way to release a temporary lock obtained on the target of the *assign*. Same as GetDeviceProc(), it does not properly handle NIL: and "*".

6.1.4 Obtaining the Current Console Handler

The GetConsoleTask() function returns the *MsgPort* of the handler responsible for the console of the calling process, that is, the process that takes care of the file name "*" or paths relative to CONSOLE:.

```
port = GetConsoleTask()
 D0

struct MsgPort *GetConsoleTask(void)
```

This function returns a port to the handler of the console of the calling process, or NULL in case there is no console associated to the caller. The latter holds for example for programs started from the workbench. It does not alter IoErr().

6.1.5 Obtaining the Default File System

The GetFileSysTask() function returns the *MsgPort* of the default *file system* of the caller. The default *file system* is used as fall-back if a *file system* is required for a path relative to the ZERO lock, and the path itself does not contain an indication of the responsible handler, i.e. is a relative path itself.

The default *file system* is typically the boot file system, or the file system of the SYS: *assign*, though it can be changed with SetFileSysTask() at any point.

```
port = GetFileSysTask()
 D0

struct MsgPort *GetFileSysTask(void)
```

This function returns the port of the default file system of this task. It does not alter `IoErr()`. Note that `SYS:` itself is an *assign* and paths starting with `SYS:` do therefore not require resolution through this function, though the default *file system* and the file system handling `SYS:` are typically identical. However, as the former is returned by `GetFileSysTask()` and the latter is part of the *device list assign*, they can be different.

The `DateStamp` structure reads as follows:

```
struct DateStamp {
   LONG  ds_Days;             /* Number of days since Jan. 1, 1978 */
   LONG  ds_Minute;           /* Number of minutes past midnight */
   LONG  ds_Tick;             /* Number of ticks past minute */
};
```

`ds_Days` counts the number of days since January 1st 1978.

`ds_Minute` counts the number of minutes past midnight, i.e. the start of the day.

`ds_Tick` counts the ticks since the start of the minute. A tick is 1/50th of a second, regardless whether the machine is a PAL or NTSC system. This constant is also defined as `TICKS_PER_SECOND` in `dos/dos.h`.

Bibliography

[1] Commodore-Amiga Inc: *AmigaDOS Manual, 3rd Edition* Random House Information Group (1991)

[2] Motorola MC68030UM/AD Rev. 2: *MC68030 Enhanced 32-Bit Microprocessor User's Manual, 3rd ed.* Prentice Hall, Englewood Cliffs, N.J. 07632 (1990)

[3] Motorola MC68040UM/AD Rev. 1: *MC68040 Microprocessor User's Manual, revised ed.* Motorola (1992,1993)

[4] Motorola MC68060UM/AD Rev. 1: *MC68060 Microprocessor User's Manual.* Motorola (1994)

[5] Motorola MC68000PM/AD Rev. 1: *Programmer's Reference Manual.* Motorola (1992)

[6] Yu-Cheng Liu: *The M68000 Microprocessor Family.* Prentice-Hall Intl., Inc. (1991)

[7] Dan Baker (Ed.): *Amiga ROM Kernal Reference Manual: Libraries. 3rd. ed.* Addison-Wesley Publishing Company (1992)

[8] Dan Baker (Ed.): *Amiga ROM Kernal Reference Manual: Devices. 3rd. ed.* Addison-Wesley Publishing Company (1992)

[9] Dan Baker (Ed.): *Amiga ROM Kernal Reference Manual: Includes and Autodocs. 3rd. ed.* Addison-Wesley Publishing Company (1991)

[10] Ralph Babel: *The Amiga Guru Book.* Ralph Babel, Taunusstein (1993)

Index