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 8.

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. They also require the volume to be present at the time the assign is created.

Non-binding assigns avoid these problems 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 8. 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 (":"). Otherwise, the call checks whether the *handler* responsible for the current directory of the calling process is a file system.

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 Readfunction 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. This is particularly important if the volume is ejected from its drive and another file system needs to take over the *locks* if the volume is later inserted into another drive.

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.

> *May go away while you look!* 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 **??**) and the Workbench; if the bit is 1, the *Shell* and the Workbench refuse 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.

Note that `Rename()` is currently not on the list supporting softlinks as part of the path to the object to be renamed, or as part of the target path.

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 5.4.2.

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 8.2.8) 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. As a courtesy to C and assembler functions, AmigaDOS ensures that this string is `NUL` terminated, i.e. `dol_Name + 1` is a regular C string whose length is available in `dol_Name[0]`.

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 BYTEs. 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 an 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.

Give back what you got To release all temporary resources, the DevProc structure returned by GetDeviceProc() shall be either be released through FreeDeviceProc() then aborting the scan, or used as first argument for GetDeviceProc() to continue the iteration. The last call to this function will return NULL and then also release all resources.

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 nameis 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.

Does not like all paths The GetDeviceProc() function unfortunately does not handle all device specifiers correctly, and some special cases need to be filtered out by the caller. Namely "*" indicating the current console, and NIL: for the NIL pseudo-device are not handled here.

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().

> *Obsolete and not fully functional* DeviceProc() 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.

6.2 Iterating and Accessing the Device List

While `GetDeviceProc()` uses the *device list* to locate a particular *MsgPort* and *Lock*, all other members of the `DosList` structure remain unavailable. For them, the *device list* containing these structures need to be scanned manually. The *dos.library* provides functions to grant access, search and release access to this list.

6.2.1 Gaining Access to the Device List

The `LockDosList()` function requests shared or exclusive access to a subset of entries of the *device list* containing all *handlers*, *volumes* and *assigns* and blocks until access is granted. It requires as input multiple sets that specify which parts of the list to access:

```
dlist = LockDosList(flags)
 D0                  D1

struct DosList *LockDosList(ULONG)
```

This function grants access to a subset of entries of the *device list* indicated by `flags`, and returns an opaque handle through which elements of the list can be accessed. For this, see `FindDosEntry()`.

The `flags` value shall be combination of the following values, all defined in `dos/dosextens.h`:

Table 20: LockDosList Flags

Flags	Description
LDF_DEVICES	Access *handlers* and *file system* entries, see 3.3.1.1
LDF_VOLUMES	Access *volumnes*, see 3.3.1.2
LDF_ASSIGNS	Access *assigns*, see 3.3.1.3
LDF_ENTRY	Lock access to a `DosList` entries
LDF_DELETE	Lock *device list* for deletion
LDF_READ	Shared access to the *device list*
LDF_WRITE	Exclusive access to the *device list*

At least LDF_READ or LDF_WRITE shall be included in the flags, they shall not be set both. The three first flags may also be combined to access multiple types.

LDF_ENTRY and LDF_DELETE are additional flags that moderate access to entries of the *device list*. If LDF_ENTRY is set, then exclusive access to the selected entries is requested and entries shall not be altered or removed. The LDF_ENTRY flag shall not be combined with LDF_READ. If LDF_DELETE is set, then access is granted for removing entries from the list.

The result code dlist is *not* a pointer to a DosList structure, but only a handle that may be passed into FindDosEntry() or NextDosEntry(). If dlist is NULL, then locking failed because the combination of flags passed in was invalid.

This function does not alter IoErr().

6.2.2 Requesting Access to the Device List

The AttemptLockDosList() requests access to the *device list* or a subset of its entries, and, in case it cannot gain access, returns NULL. Unlike LockDosList(), it does not block.

```
dlist = AttemptLockDosList(flags)
D0                         D1

struct DosList *AttemptLockDosList(ULONG)
```

The flags argument specifies which elements of the *device list* are requested for access, and which type of access is required. The flags are a combination of the flags listed in table 20, and the semantics of the flags are exactly as specified for LockDosList(), see there for details.

The result code is either a (non-NULL) handle that may be passed into FindDosEntry() or NextDosEntry() in case access could be granted, or NULL. In the latter case, the list is either currently locked and access cannot be granted without blocking, or flags are invalid. These two cases of failure cannot be distinguished unfortunately.

This function does not alter IoErr().

6.2.3 Release Access to the Device List

The UnLockDosList() function releases access to the *device list* once obtained through LockDosList().

```
UnLockDosList(flags)
              D1

void UnLockDosList(ULONG)
```

This function releases access to the *device list* again. The flags argument shall be identical to the flagsargument provided to LockDosList().

6.2.4 Iterate through the Device List

The NextDosEntry() iterates to the next entry in the *device list* given the current entry or the handle returned by LockDosList().

```
newdlist = NextDosEntry(dlist,flags)
 D0                     D1    D2

struct DosList *NextDosEntry(struct DosList *,ULONG)
```

This function returns the next DosList structure of the *device list* which shall have been locked with LockDosList(). The dlist argument shall be either the return code of a previous NextDosEntry() or FindDosEntry() call, or the handle returned by LockDosEntry().

The flags argument shall be a subset of the flags argument into LockDosList() and specifies the type of DosList structures that shall be found. Only the first 3 elements of Table 20 are relevant here, all other flags are ignored but may be included.

The newdlist result is either a pointer to a DosList structure of the requested type, or NULL if the end of list has been reached. This function does not alter IoErr().

6.2.5 Find a Device List Entry by Name

The FindDosEntry() function finds a DosList structure of a particular type and particular name, from a particular entry on, or the handle returned by LockDosList().

```
newdlist = FindDosEntry(dlist,name,flags)
D0                      D1    D2   D3

struct DosList *FindDosEntry(struct DosList *,STRPTR,ULONG)
```

This function scans through the *device list* starting at the entry dlist, or the handle returned by LockDosList(), and returns the next DosList structure that is of the type indicated by flags and has the name name.

The flags shall be a subset of the flags argument passed into LockDosList(). Only the first 3 elements of Table 20 are relevant here, all other flags are ignored but may be included.

The name argument is the (case-insensitive) name of the assign, handler, file system or volume the function should look for. The name *shall not* include the colon (':') that separates the name from the remaining components of a path, see section 3.3. It may be NULL in which case every entry of the requested type matches.

The returned newdlist is a pointer to a DosList structure that matches the name (if provided) and flags passed in, or NULL in case no match could be found and the entire list has been scanned. Note that the returned DosList may be identical to the dlist passed in if it already fits the requirements. Thus potentially, NextDosList() may be called upfront to scan from the subsequent entry.

Passing NULL as dlist is safe and returns NULL, i.e. the end of the list. Note that the (pseudo-) devices from tables 3 and 5 are not part of the *device list*, i.e. NIL, CONSOLE, * and PROGDIR cannot be found and are special cases of GetDeviceProc().

This function does not alter IoErr().

6.3 Adding or Removing Entries to the Device List

The *dos.library* provides two service functions to add or remove DosList structures from the *device list*. They secure the *dos.library* internal state from inconsistencies as other processes may attempt to access the *device list* simultaneously, and they also ensure proper linkage of the structures.

Locking the device list in file systems There is one particular race condition *file system* authors should be aware of. When opening a file, or obtaining a *lock*, the *dos.library* calls through `GetDeviceProc()` to identify a *handler* responsible for the requested path. As `GetDeviceProc()` requires access to the *device list*, it will secure access to it through `LockDosList()`, then possibly start up the *handler*, and then unlock the list. Thus, at the time the handler is initiated, it may find the *device list* unaccessible. Attempting to lock it would result in a *deadlock* situation as the *dos.library* waits for the *handler* to reply its startup packet, and the *handler* waits for the *dos.library* to grant access to the *device list*. The following sections provide workarounds how to avoid this situation, see also section **??** for details on the *handler* and *file system* startup mechanism.

6.3.1 Adding an Entry to the Device List

The `AddDosEntry()` adds an initialized `DosList` structure to the *device list*.

```
success = AddDosEntry(dlist)
D0                    D1

LONG AddDosEntry(struct DosList *)
```

This function takes an initialized `DosList` entry pointed to by `dlist` and attempts to add it to the *device list*. For this, it requests write access to the list, i.e. locking of the *device list* through the caller is not necessary. The `DosList` may be either created manually, by `MakeDosEntry()` of the *dos.library* or by `MakeDosNode()` of the *expansion.library*. While there the structure is called a `DeviceNode`, it is still a particular incarnation of a `DosList` and may be savely used here.

Assigns shall not be added to the *device list* through this function, but rather through the functions in section 6.5. This avoids memory management problems when releasing or changing assigns.

Particular care needs to be taken if this function is called from within a *handler* or *file system*, e.g. to add a *volume* representing an inserted medium. As the list may be locked by the *dos.library* to secure the list from modifications within a `GetDeviceProc()` function, a deadlock may result where *file system* and *dos.library* mutually block access. To prevent this from happening handlers should check upfront whether the *device list* is available for modifications by `AttemptLockDosList()`, e.g.

```
if (AttemptLockDosList(LDF_VOLUMES|LDF_WRITE)) {
  rc = AddDosEntry(volumenode);
  UnLockDosList(LDF_VOLUMES|LDF_WRITE);
}
```

when adding a `DosList` entry of type `DLT_VOLUME`. If attempting to get write access failed, the handler should check for incoming requests, handle them, and attempt adding the entry later.

The function fails if an entry is to be added and an entry of the same name, regardless its type, is already present on the list. The only exception is that the list may contain two *volumes* of the same name, provided provided their creation date `dol_VolumeDate` differs, see section 6.

If successful, the function returns non-zero, but then does not alter `IoErr()`. The `DosList` is then enqueued in the *dos.library* database and it and its members shall then no longer be altered or released by the caller. On failure, the function returns 0 and `IoErr()` is set to `ERROR_OBJECT_EXISTS`.

6.3.2 Removing an Entry from the Device List

The `RemDosEntry()` removes a `DosList` entry from the *device list*, making it unacessible for AmigaDOS.

```
success = RemDosEntry(dlist)
D0                    D1

BOOL RemDosEntry(struct DosList *)
```

This function attempts to find the `DosList` structure pointed to by `dlist` in the *device list* and, if present, removes it. Unlike what some other documentation says, this funciton locks the *device list* properly before attempting to remove an entry, locking it upfront is not necessary.

The function does *not* attempt to release the memory allocated for the `DosList` passed in, or any of its members, it just removes the `DosList` from the *device list*. While *file systems* may know how they allocated the `DosList` structures represening their *volumes* and hence should be aware how to release the memory taken by them, there is no good solution on how to recycle memory for `DosList` structures representing *handlers*, *file systems* or *assigns*. Some manual footwork is currently required, see also `FreeDosNode()`. In particular, as entries representing *handlers* and *file systems* may have been created in multiple ways, their memory cannot be safely recycled.

Particular care needs to be taken if this function is called from within a *handler* or *file system*, e.g. to remove a *volume* representing a removed medium. As the list may be locked by the *dos.library* to secure the list from modifications within a `GetDeviceProc()` function, a deadlock may result where *file system* and *dos.library* mutually block access. To prevent this from happening handlers should check upfront whether the *device list* is available for modifications by `AttemptLockDosList()`, e.g.

```
if (AttemptLockDosList(LDF_DELETE|LDF_ENTRY|LDF_WRITE)) {
  rc = RemDosEntry(volumenode);
  UnLockDosList(LDF_DELETE|LDF_ENTRY|LDF_WRITE);
}
```

when removing a `DosList` entry. If attempting to get write access failed, the handler should check for incoming requests, handle them, and attempt adding the entry later.

This function returns a success indicator; it returns non-zero if the function succeeds, and 0 in case it fails. The only reason for failure is that `dlist` is not a member of the *device list*. This function does not touch `IoErr()`.

6.4 Creating and Deleting Device List Entries

AmigaOs offers multiple functions to create `DosList` structures. The `MakeDosEntry()` function is a low-level function that allocates a `DosList` but only performs minimal intialization of the structure. For *assigns*, the functions in section 6.5 shall be used as they include complete initialization of the `DosList`, and for *handlers* and *file systems*, the *expansion.library* function `MakeDosNode()` is a proper alternative. Releasing `DosLists` along with all its resources is unfortunately much harder. For *assigns*, the algorithm in section 6.4.2 provides a workable function based on `FreeDosEntry()`.

`DosLists` representing *Volumes* are build and released by *file systems*; it depends on them which resources need to be released along with the `DosList` structure itself. While it is recommended that *file systems* should go through `MakeDosEntry()` and `FreeDosEntry()`, it is not a requirement.

Releasing a `DosList` representing a *handler* or *file system* is currently not possible in a completely robust way. It is suggested just to unlink such nodes if absolutely necessary, but tolerate the memory leak.

6.4.1 Creating a Device List Entry

The `MakeDosEntry()` creates an empty `DosList` structure of the given type, and makes all elementary initializations. It does not accquire any additional resources, and neither inserts it into the *device list*.

If an *assign* is to be created, the functions in section 6.5 are better alternatives and should be preferred as they perform a more sophisticated initialization.

```
newdlist = MakeDosEntry(name, type)
D0                      D1    D2

struct DosList *MakeDosEntry(STRPTR, LONG)
```

This function allocates a `DosList` structure and initializes its `dol_Type` to `type`. The `type` argument shall be one of the values from table 18. The function also makes a copy of `name` and initializes the `dol_Name` to a *BSTR* copy of `name`, which is a `NUL` terminated C string.

Note that this function performs only minimal initialization of the `DosList` structure. All other members except `dol_Type` and `dol_Name` are initialized to 0.

This function either returns the allocated structure, or `NULL` for failure. In the latter case, `IoErr()` is set to `ERROR_NO_FREE_STORE`. On success, `IoErr()` remains unaltered.

6.4.2 Releasing a Device List Entry

The `FreeDosEntry()` function releases a `DosList` structure allocated by `MakeDosEntry()`. The `DosList` shall be already removed from the *device list* by *RemDosEntry()*. While this call releases the memory holding the name of the entry, and also the `DosList` structure itself, it does not release any other resources. They shall be released by the caller of this function. Furthermore, this function shall not be called if the `DosList` structure was allocated by any other means than `MakeDosEntry()`.

```
FreeDosEntry(dlist)
             D1

void FreeDosEntry(struct DosList *)
```

This function releases the `DosList` structure pointed to by `dlist` and its name, but only these two resources, and no other resources.

If `dol_Type` is `DLT_DEVICE`, corresponding to *handlers* or *file systems*, this function should better not be called at all as the means of how the `DosList` was allocated is unclear. In such a case, a memory leak is the least dangerous side effect.

If `dol_Type` is `DLT_DIRECTORY` or `DLT_LATE`, then `dol_Lock` should be unlocked. If `dol_List` is non-`NULL`, then each entry of the `AssignList` structure shall be released, along with the lock kept within. For `DLT_LATE` and `DLT_NONBINDING`, the `dol_AssignName` function shall also be released. The following code segment releases all resources for *assigns*:

```
struct AssignList *al,*next;
UnLock(dol->dol_Lock);
al = dol->dol_misc.dol_assign.dol_List;
while(al) {
    next = al->al_Next;
    UnLock(al->al_Lock);
    FreeVec(al);
    al = next;
}
FreeVec(dol->dol_misc.dol_assign.dol_AssignName);
FreeDosEntry(dol);
```

The above code reflects the way how resources were originally allocated by the *dos.library*.

If the type is `DLT_VOLUME`, it is up to the *file system* to release any resources it allocated along with the `DosList`. It is file system dependent which resources can or should be released. `DosList` entries of this type should only be touched by the *file system* that created them.

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

6.5 Creating and Updating Assigns

While `MakeDosEntry()` creates a `DosList` entry for the *device list*, it only performs minimal initialization of the structure. For *assigns*, specifically, the *dos.library* provides specialized functions that allocate, initialize and enqueue `DosList` structures representing assigns in a single call and are thus easier to use.

6.5.1 Create and Add a Regular Assign

The `AssignAdd()` function creates a new assign to a directory from a *lock*, and then enqueues it into the *device list*.

```
success = AssignLock(name,lock)
D0                   D1   D2

BOOL AssignLock(STRPTR,BPTR)
```

This function creates a (regular) *assign* onto the directory identified by `lock`. The *assign* created under the name as given by `name`. The name shall not include a trailing colon (“`:`”) that separates the *assign* name from the rest of the path. The lock shall be a *shared lock*.

If the function is successful, it returns a non-zero result code. The `lock` is then absorbed into the *assign* and shall no longer be used by the calling program. On success, `IoErr()` is not altered.

On error, the function returns 0 and the `lock` remains available to the caller. `IoErr()` is set to an error code identifying the cause of the failure. `ERROR_NO_FREE_STORE` is returned if the function run out of memory. If a `DosList` of the same name (regardless of which type) already exists, the error code is `ERROR_OBJECT_EXISTS`.

6.5.2 Create a Non-Binding Assign

The `AssignPath()` function creates a *non-binding assign* and adds it to the *device list*. This type of assign binds to a path independent of the volume the path is located on; that is, the *assign* resolves to whatever *volume*, *handler* or even other *assign* matches the path.

```
success = AssignLate(name,path)
D0                   D1   D2

BOOL AssignLate(STRPTR,STRPTR)
```

This function creates a *non-binding* assign whose name is given by the first argument, and which resolves to the path given as second argument, and then adds the *assign* to the *device list*. The `name` shall not contain a trailing colon (“`:`”). While not a formal requirement of the function or *non-binding assigns*, the `path` should better be an absolute path as otherwise resolution of the created *assign* can be very confusing — it is then resolved relative to the current directory of the calling process.

If the function is successful, it returns a non-zero result code. On success, `IoErr()` is not altered.

On error, the function returns 0 and `IoErr()` is set to an error code identifying the cause of the failure. `ERROR_NO_FREE_STORE` is returned if the function run out of memory. If a `DosList` of the same name (regardless of which type) already exists, the error code is `ERROR_OBJECT_EXISTS`.

6.5.3 Create a Late Assign

The `AssignLate()` function creates a *late assign* whose target is initially given by a path; but after its first resolution, the *assign* reverts to a *regular assigns* such that the target of the *assign* will point to the same directory of the volume from that point on. This has the advantage that the target of the assign does not need to be available at creation time of the assign, yet remains unchanged after its first usage.

```
success = AssignLate(name,path)
D0                   D1   D2

BOOL AssignLate(STRPTR,STRPTR)
```

This function creates a *late binding assign* of the name `name` pointing to `path` as its destination and adds it to the *device list*. The `name` shall not contain a trailing colon (“`:`”). While not explicitly required by this function, the `path` should better be an absolute path as otherwise resolving the *assign* can be very confusing. The `path` is then relative to the current directory of the process using the *assign* the first time.

If the function is successful, it returns a non-zero result code. On success, `IoErr()` is not altered.

On error, the function returns 0 and `IoErr()` is set to an error code identifying the cause of the failure. `ERROR_NO_FREE_STORE` is returned if the function run out of memory. If a `DosList` of the same name (regardless of which type) already exists, the error code is `ERROR_OBJECT_EXISTS`.

6.5.4 Add a Directory to a Multi-Assign

The `AssignAdd()` function adds a directory, identified by a *lock*, to an already existing *regular* or *multi-assign*. On success, a *regular assign* is then converted into a *multi-assign*.

```
success = AssignAdd(name,lock)
D0                  D1   D2

BOOL AssignAdd(STRPTR,BPTR)
```

This function adds the `lock` at the end of the target directory list of the *assign* identified by `name`. The `name` does not contain a trailing colon (“`:`”).

A `DosList` of the given `name` shall already when entering this function, and this `DosList` shall be a *regular assign*. Attempting to add a directory to a *handler*, *file system*, *volume* or any other type of *assign* fails.

On success, the function returns a non-zero result code. In such a case, the `lock` is absorbed into the *assign* and shall no longer be used by the caller. The *assign* is converted into a *multi-assign* on access if it is not already one. The `lock` is added at the end of the directory list, i.e. the new directory is scanned last when resolving the *assign*.

On error, the function returns 0 and the `lock` remains available to the caller. Unfortunately, this function does not set `IoErr()` consistently, i.e. it is unclear on failure what caused the error, i.e. whether the function run out of memory, whether no fitting *device list* entry was found, or whether the entry found was not a *regular assign*.

6.5.5 Remove a Directory From a Multi-Assign

The `RemAssignList()` function removes a directory, represented by a `lock`, from a `multi-assign`. If only a single directory remains in the *multi-assign*, it is converted into a *regular assign*. If the *assign* was a regular assign, and the only directory is removed from it, the *assign* itself is removed from the *device list* and released, destroying it and releasing all resources.

```
success = RemAssignList(name,lock)
D0                      D1   D2

BOOL RemAssignList(STRPTR,BPTR)
```

This function removes the directory identified by `lock` from a *regular* or *multi-assign* identified by `name`. The name shall not contain a trailing colon (“`:`”). If only a single directory remains in the *assign*, it is converted to a *regular assign*. If no directory remains at all, the *assign* is deleted and removed from the *device list*. The `lock` remains available to the caller, regardless of the result code. Note that the `lock` passed in does not need to be identical to the *lock* contained in the *assign*, but it needs to be a *lock* on the same directory. This function uses `SameLock()` function to compare the two locks.

On success, the function returns a non-zero result code in `success`. On error, the function returns `0`. Unfortunately, it does not set `IoErr()` consistently in all cases, and thus, the cause of an error cannot be determined upon return. Possible causes of error are that `name` does not exist, or that it is not a *assign* or a *multi-assign*.

Chapter 7

Pattern Matching

Unlike other operating systems, it is neither the file system nor the shell that expands wild cards, or patterns. Instead, separate functions exist that, given a wildcard, scan a directory or an entire directory tree and deliver all files, links and directories that match a given pattern.

The pattern matcher syntax is build on special characters or *tokens* that define which names to match. The following tokens are currently defined:

`?` The question mark matches a single, arbitrary character within a component. When using the pattern matcher for scanning directories, the question mark does not match the component separator, i.e. the slash ("`/`) and the colon ("`:`") that separates the path from the device name. Note in particular that the question mark also matches the dot ("`.`") which is not a special character under AmigaDOS.

`#` The hash mark matches zero or more repeats of the token immediately following it. In particular, the combination "`#?`" matches zero or more arbitrary characters. If a group of more than one token is required to describe which combination needs to match, this group needs to be enclosed in brackets.

`()` The brackets bind tokens together forming a single token. This is particularly useful for the hash mark `#` as it allows to formulate repeats of longer character or token groups. For example, `#(ab)` indicates zero or more repeats of the character sequence `ab`, such as `ab`, `abab` or `abababab`.

`~` The ASCII tilde ("~") matches names that do not match the next token. This is particularly valuable for filtering out the workbench icon files that end on `.info`, i.e. `~(#?.info)` matches all files that do not end with `.info`.

`[]` The square brackets ("`[]`") matches a single character from a range, e.g. `[a-z]` matches a single alphabetic character and `[0-9]` matches a single digit. Multiple ranges and individual characters can be combined, for example `[ab]` matches the characters `a` and `b`, whereas `[a-cx-z]` matches the characters from `a` to `c` and from `x` to `z`. If the minus sign ("–") is supposed to be part of the range, it shall appear first, directly within the bracket, e.g. `[-a-c]` matches the dash and the characters `a` to `c`. If the dash is the last character in the range, all characters up to the end of the ASCII range, i.e. `0x7f` match, but none of the extended ISO Latin 1 characters match. If the closing square bracket ("`]`") is to matched, it shall be escaped by an apostroph ("`'`"), i.e. `[[-']]` matches the opening and the closing bracket. If the pattern matcher is used for scanning directories, the above example does not match the slash ("`/`") even though its code point lies between the opening and closing bracket because the slash cannot be part of a component name and rather separates components. If the first character of the range is an ASCII tilde ("~"), then the character class matches all characters *not* in the class, i.e. `[~a-z]` matches all characters except alphabetic characters. In all other places, the tilde stands for itself.

`'` The apostroph (`'`) is the escape character of the pattern matcher and indicates that the next character is not a token of the matcher, but rather stands for itself. Thus, `'?` matches the question mark, and only the question mark, and no other character.

`%` The percent sign ("`%`") matches the empty string.

`|` The vertical bar ("`|`") defines alternatives and matches the token to its left or the token to its right. The alternatives along with the vertical bar shall be enclosed in round brackets to bind them, i.e. `(a|b)` is either the character `a` or `b` and therefore matches the same strings `[ab]` matches. A particular example is `~((#?.info)|.backdrop)` which matches all files not used by the workbench for storing meta-information.

> *The Asterisk `*` is not a Wildcard* Unlike many other operating systems, the asterisk ("`*`") has a (two) other meanings under AmigaDOS. It rather refers to the current console as file name, or is the escape character for quotation and control sequences; those are properties AmigaDOS inherits from the BCPL syntax and TRIPOS. While there is a flag in the *dos.library* that makes the asterisk *also* available as a wildcard, such usage is discouraged because it can lead to situations where the asterisk is interpreted differently than intended — as it has already two other meanings.

Pattern matching works in in two steps: In the first step, the pattern is tokenized into an internal representation, which is then later on used to perform the actual match of a string against a wildcard. The directory scanning function `MatchFirst()` performs this conversion internally, and thus no additional preparation is required by the caller in this case. However, if the pattern matcher is used to search for strings or wildcards within a text file, the pattern tokenizers `ParsePattern()` or its case-insensitive counterpart `ParsePatternNoCase()` shall be called first.

> *Only ISO-Latin Codepoints* The pre-parsing step that prepares from the input pattern its tokenized version uses the code points `0x80` to `0x9f` for tokenized versions of wild-cards and other instructions for the pattern matcher. This is identical to the extended ISO-Latin control sequence region, and does not represent printable charactes. While file names on AmigaDOS *file systems* may in principle include such code-points, patterns of the pattern matcher *shall not* contain unprintable code points from the region `0x00` to `0x1f` or from `0x80` to `0x9f`. These regions are reserved for the pattern matcher.

7.1 Scanning Directories

The prime purpose of the pattern matcher is to scan a directory, or even a tree of directories, identifying all *file system* objects such as files, links or directories that match a given pattern. The pattern matcher can even descend recursively into sub-directories if instructed to do so. This service is used by many shell commands stored in the `C:` *assign*. The directory scanner requires the following steps:

First, the user shall provide an `AnchorPath` structure. This structure contains the state of the directory matcher, including the `FileInfoBlock` structure of the matched object. This structure is defined in section 5.1. Optionally, the `AnchorPath` structure may also contain the complete (relative) path of the matched object. This structure shall then be initialized, setting all flags required, see below for their definition.

> *Must be Long-Word Aligned* As the `AnchorPath` structure embedds a `FileInfoBlock` structure that requires long-word alignment, the `AnchorPath` structure shall be aligned to long-word boundaries as well. The simplest way to ensure this is to allocate it with either `AllocMem()` or `AllocVec()`, see also section 2.3.

Then, with the initialized AnchorPath structure, MatchFirst() shall be called, returning the first match of the pattern if there is any. The AnchorPath structure then contains all information on the found match.

If there is any match, and the match is a directory the caller wants to enter recursively, the APF_DODIR flag of the AnchorPath structure may be set. Then, MatchNext() may be called to continue the scan, potentially entering this directory. Once the end of a recursively entered directory has been reached, MatchNext() sets the APF_DIDDIR flag, then reverts back to the parent directory continuing the scan there. As APF_DIDDIR is never cleared by the pattern matcher, the caller should clear it once the end of a sub-directory had been noticed.

The above iterative procedure of MatchNext() may continue, either until the user or the running program requests termination, or until MatchNext() returns an error. Then, finally, the scan is aborted and all resources but the AnchorPath structure shall be released by calling MatchNext().

The AnchorPath structure is defined in dos/dosasl.h and looks as follows:

```
struct AnchorPath {
        struct AChain   *ap_Base;
#define ap_First ap_Base
        struct AChain   *ap_Last;
#define ap_Current ap_Last
        LONG    ap_BreakBits;
        LONG    ap_FoundBreak;
        BYTE    ap_Flags;
        BYTE    ap_Reserved;
        WORD    ap_Strlen;
        struct  FileInfoBlock ap_Info;
        UBYTE   ap_Buf[1];
};
```

The members of this structure are as follows:

ap_Base and ap_Last are pointers to an AChain structure that is also defined in dos/dosasl.h. This structure is allocated and released by the *dos.library*, transparently to the caller. The AChain structure describes a directory in the potentially recursive scan through a directory tree. ap_Base describes the topmost directory at which the scan started, whereas ap_Last describes the directory which is currently being scanned.

The AChain structure is also defined in dos/dosasl.h:

```
struct AChain {
struct AChain *an_Child;
struct AChain *an_Parent;
BPTR an_Lock;
struct FileInfoBlock an_Info;
BYTE an_Flags;
UBYTE an_String[1];
};
```

an_Child and an_Parent are only used internally and shall not be interpreted by the caller.

an_Lock is a lock to the directory described by this AChain structure. In particular, ap_Last->an_Lock is a *lock* to the directory that is currently being scanned, and ap_Base->an_Lock a lock to the topmost directory at which the scan started. These two locks have been obtained and will be unlocked by the *dos.library*; they may be used by the caller provided they are not unlocked manually.

an_Info is only used internally and is the FileInfoBlock of the directory being describes by the AChain structure, see section 5.1.

an_Flags is only used internally, and an_String can contain potentially the path to the directory; both shall not be modified or interreted by the caller.

ap_BreakBits of the AnchorPath structure shall be initialized to the signal mask upon which MatchNext() aborts a directory scan. This is typically a combination of signal masks found in dos/dos.h, e.g. SIGBREAKF_CTRL_C to abort on Ctrl-C in the console.

ap_FoundBreak contains, if MatchNext() aborts with ERROR_BREAK, the signal mask that caused the abortion.

ap_Flags contains multiple flags that can be set or inspected by the caller while scanning a directory. In particular:

APF_DOWILD while documented, is not used nor set at all by the pattern matcher.

APF_ITSWILD is set by MatchFirst() if the pattern includes a wildcard and more than a single *file system* object may match. Otherwise, no directory scan is performed. The user may also set this flag to enforce a scan. This may resolve situations in which matching an explicit path without a wildcard is not possible because the object is locked exclusively.

APF_DODIR may be set or reset by the caller of MatchNext() to enforce entering a directory recursively, or avoid entering a directory. This flag is cleared by MatchNext() when entering a directory, and it shall only be set by the caller if a match describes a directory.

APF_DIDDIR is set by MatchNext() if the end of a recursively entered directory has been reached, and thus the parent directory is re-entered. As this flag is never cleared by the pattern matcher, it should be cleared by the caller.

APF_NOMEMERR is an internal flag that should not be interpreted; it is set if an error is encountered while scanning a directory. It is not necessarily restricted to memory allocation errors.

APF_DODOT is, even though documented, not actually used.

APF_DirChanged is a flag that is set by MatchNext() if the scanned directory changes, either by entering a directory recursively, or by leaving a directory. It is also cleared if the directory is the same as in the previous call.

APF_FollowHLinks may be set by the caller to indicate that hard links to directories shall be followed, and such directories shall be recursively entered if APF_DODIR is set as well. Otherwise, hard links to directories are not entered. Softlinks are neither entered, this this cannot be changed by any flag. A potential danger of links is that they may cause endless recursion if a link within a directory points to a parent directory. Thus, callers should be aware of such situations and store directories that have already been analyzed. Otherwise, it is safer to keep this flag cleared.

ap_Strlen is the size of the buffer ap_Buf that contains the full path of the matched entry. This buffer shall be allocated by the user at the end of the AnchorPath structure. Unlike what the name suggests, this is not a string length, but the byte size of the buffer, including the terminating NUL byte of a string. If the full path of the match does not fit into this buffer, it is truncated *without* proper string termination and the error code ERROR_BUFFER_OVERFLOW is returned. If the full path is not required, this member shall be set to 0.

ap_Info contains the FileInfoBlock of the matched entry, including all metadata the file system has available for it. Note that fib_FileFile only contains the name of the object, not its full path.

ap_Buf is filled with the full path to the matched object if ap_Strlen is non-zero. This buffer shall be allocated by the caller at the end of the AnchorPath structure, i.e. for a buffer of l bytes, in total sizeof(AnchorPath)+l-1 bytes are required to store the structure and the buffer. The byte size of this additional buffer shall be placed in ap_Strlen. If this buffer is not required, ap_Strlen shall be set to 0.

7.1.1 Starting a Directory Scan

The MatchFirst() function starts a directory scan, locating all objects matching a pattern and potentially entering directories recursively.

```
error = MatchFirst(pat, AnchorPath)
D0                 D1       D2

LONG MatchFirst(STRPTR, struct AnchorPath *)
```

This function starts a directory scan, locating all objects matching the pattern pat. This pattern does *not* require pre-parsing (e.g. the functions in section 7.2), MatchFirst() performs the parsing.

AnchorPath shall be a pointer to an AnchorPath structure allocated and initialized by the caller. In particular, ap_BreakBits shall be initialized to a signal mask on which the scan terminates, ap_FoundBreak to 0, and ap_Strlen to the size of the buffer ap_Buf which is filled by the path name of the matching objects. If this path name is not required, ap_Strlen shall be set to 0. ap_Flags shall be set to the flags you need, see the parent section.

Unlike many other functions, MatchFirst() returns an error code directly, and not a success/failure indicator. That is, 0 indicates success. In particular, if ERROR_BREAK is returned in case any of the signal bits in ap->ap_BreakBits have been received during the scan.

On success, ap->ap_Info.fib_FileName contains the name of the first matched object, the directory represented as a *lock* containing the object is available in ap->ap_Current->an_Lock. You would typically set the current directory to this lock, then access this object, then revert the lock. This lock *shall not* be released; this is performed by the pattern matcher itself as needed.

If the full path of the matching object is needed, an additional buffer shall be allocated at the end of the AnchorPath, and the size of the buffer shall be placed into ap_Strlen. The function then fills in the path into ap_Buf.

If the matching object is a directory, i.e. ap->ap_Info.fib_DirEntryType is positive and not equal to ST_SOFTLINK, the caller may request to enter this directory by setting APF_DODIR in ap->ap_Flags.

7.1.2 Continuing a Directory Scan

The MatchNext() function continues a directory scan initiated by MatchFirst(), returning the next matching object, or an error.

```
error = MatchNext(AnchorPath)
D0                D1

LONG MatchNext(struct AnchorPath *)
```

This function takes an existing AnchorPath structure, as prepared by a previous MatchFirst() or MatchNext() function, and finds the next matching object. Unlike most other functions of the *dos.library*, this function returns an error code on failure and 0 for success. It does *not* return a boolean success indictor. In particular, if ERROR_BREAK is returned in case any of the signal bits in ap->ap_BreakBits have been received.

As for MatchFirst(), this call fills ap->ap_Info with meta information on the found object, in particular its file name, and ap->ap_Current->an_Lock the lock of the directory containing the object. As for MatchFirst(), APF_DODIR can be set to enter directories recursively, and ap->ap_Buf will be filled with the full path of the found object if ap->ap_Strlen is non-zero.

7.1.3 Terminating a Directory Scan

The `MatchEnd()` function terminates a running scan, and releases all resources associated with the scan. It does not release the `AnchorPath` structure.

```
MatchEnd(AnchorPath)
             D1

VOID MatchEnd(struct AnchorPath *)
```

This function ends a directory scan started by `MatchFirst()` and releases all resources associated to the scan. This function shall be called regardless whether the scan is aborted due to exhaustion (i.e. `ERROR_NO_MORE_ENTRIES`, by error, or by choice of the scanning program (i.e. the desired object has been detected and no further matches are required).

7.2 Matching Strings against Patterns

While the prime purpose of the pattern matcher is to scan directories, it can also be used to check whether an arbitrary string matches a wildcard, for example to scan for a pattern within a text document. This requires two steps: In the first step, the wildcard is preparsed, generating a tokenized version of the pattern. The second step checks whether a given input string matches the pattern. You would typically tokenize the pattern once, and then use it to match multiple strings to the pattern.

Two versions of the tokenizer and pattern matcher exist: One pair that is case-sensitive, and the second pair is case-insensitive. Note that AmigaDOS file names are case-insensitive, so the `MatchFirst()` and `MatchNext()` functions internally only use the second piar.

The buffer for the tokenized version of the pattern shall be allocated by the caller. It requires a buffer that is at least `2 + (n « 1)` bytes large, where n is the length of the input wildcard.

7.2.1 Tokenizing a Case-Sensitive Pattern

The `ParsePattern()` function tokenizes a pattern for case-sensitive string matching. This tokenized version is then later on used to test a string for a match.

```
IsWild = ParsePattern(Source, Dest, DestLength)
d0                    D1     D2      D3

LONG ParsePattern(STRPTR, STRPTR, LONG)
```

This function tokenizes a wildcard pattern in `Source`, generating a tokenized version of the pattern in `Dest`. The size (capacity) of the target buffer is `DestLength` bytes. This size shall be at least `2 + (n « 1)` bytes large, where `n` is the length of the input pattern. However, as future implementations can require larger buffers, the result code shall be checked nevertheless for error conditions. The result code `IsWild` is one of the following:

`1` is returned if the source contained wildcards.

`0` is returned if the source contains no wildcards. In this case, the tokenized pattern may still be used to match a string against the pattern, though a simple string comparison would also work.

`-1` is returned in case of an error, either because the input pattern is ill-formed, or because `DestLength` is too short. In such a case, `IoErr()` should be used to obtain the reason of the failure.

7.2.2 Tokenizing a Case-Insensitive Pattern

The ParsePatternNoCase() function tokenizes a pattern for case-insensitive string matching. This tokenized version is then later on used to test a string for a match. This version is suitable for matching file names, but is otherwise similar to ParsePattern().

```
IsWild = ParsePatternNoCase(Source, Dest, DestLength)
d0                          D1     D2      D3

LONG ParsePatternNoCase(STRPTR, STRPTR, LONG)
```

This function tokenizes a wildcard pattern in Source, generating a tokenized version of the pattern in Dest. The size (capacity) of the target buffer is DestLength bytes. This size shall be at least 2 + (n « 1) bytes large, where n is the length of the input pattern. However, as future implementations can require larger buffers, the result code shall be checked nevertheless for error conditions. The result code IsWild is one of the following:

1 is returned if the source contained wildcards.

0 is returned if the source contains no wildcards. In this case, the tokenized pattern may still be used to match a string against the pattern, though a simple case-insensitive string comparison would also work.

−1 is returned in case of an error, either because the input pattern is ill-formed, or because DestLength is too short. In such a case, IoErr() should be used to obtain the reason of the failure.

7.2.3 Match a String against a Pattern

The MatchPattern() function matches an input string against a tokenized pattern, in a case sensitive way.

```
match = MatchPattern(pat, str)
D0                   D1   D2

BOOL MatchPattern(STRPTR, STRPTR)
```

This function matches the string str against the tokenized pattern pat, returning an indicator whether the string matches the pattern. This function is case-sensitive. The pattern pat shall have been tokenized by ParsePattern().

The result code match is non-zero in case the string matches, or 0 in case either the string did not match, or the function run out of stack. The latter two cases can be distinguished by IoErr(). In case the string did not match, IoErr() returns 0, or a non-zero error code otherwise. A possible error code is ERROR_TOO_MANY_LEVELS indicating that the pattern matcher run out of stack due to too many levels of recursion.

The caller shall have at least 1500 bytes of stack space available to avoid race conditions, despite the function checking for out-of-stack conditions.

7.2.4 Match a String against a Pattern ignoring Case

The MatchPatternNoCase() function matches an input string against a tokenized pattern ignoring the case.

```
match = MatchPatternNoCase(pat, str)
D0                         D1   D2

BOOL MatchPatternCase(STRPTR, STRPTR)
```

This function matches the string `str` against the tokenized pattern `pat`, returning an indicator whether the string matches the pattern. This function is case-insensitive. The pattern `pat` shall have been tokenized by `ParsePatternNoCase()`.

The result code `match` is non-zero in case the string matches, or 0 in case either the string did not match, or the function run out of stack. The latter two cases can be distinguished by `IoErr()`. In case the string did not match, `IoErr()` returns 0, or a non-zero error code otherwise. A possible error code is `ERROR_TOO_MANY_LEVELS` indicating that the pattern matcher run out of stack due to too many levels of recursion.

The caller shall have at least 1500 bytes of stack space available to avoid race conditions, despite the function checking for out-of-stack conditions.

Chapter 8

Processes

Processes are extensions of exec *tasks*, and as such scheduled by exec. The most important extensions are that processes include a message port in the form of a *MsgPort* structure for inter-process communication to *handlers*, a current directory to resolve relative paths, and the last input/output error as returned by the `IoErr()` function.

Processes are represented by the `Process` structure documented in `dos/dosextens.h`. It reads as follows:

```
struct Process {
    struct  Task    pr_Task;
    struct  MsgPort pr_MsgPort;
    WORD    pr_Pad;
    BPTR    pr_SegList;
    LONG    pr_StackSize;
    APTR    pr_GlobVec;
    LONG    pr_TaskNum;
    BPTR    pr_StackBase;
    LONG    pr_Result2;
    BPTR    pr_CurrentDir;
    BPTR    pr_CIS;
    BPTR    pr_COS;
    APTR    pr_ConsoleTask;
    APTR    pr_FileSystemTask;
    BPTR    pr_CLI;
    APTR    pr_ReturnAddr;
    APTR    pr_PktWait;
    APTR    pr_WindowPtr;

    /* following definitions are new with 2.0 */
    BPTR    pr_HomeDir;
    LONG    pr_Flags;
    void    (*pr_ExitCode)();
    LONG    pr_ExitData;
    UBYTE   *pr_Arguments;
    struct MinList pr_LocalVars;
    ULONG   pr_ShellPrivate;
    BPTR    pr_CES;
};  /* Process */
```

The members of this structure are as follows:

pr_Task is the exec task structure defined in exec/tasks.h. It is required by the exec scheduler. The only difference between an exec Task and a Process is that pr_Task.tc_Node.ln_Type is set to NT_PROCESS instead to NT_TASK.

pr_MsgPort is a message port structure as defined in exec/ports.h. This port is used by many functions of the *dos.library* to communicate with *handlers* and *file systems*. Details of the communication protocol are given in section **??**.

pr_Pad is unused and only included in the structure to ensure that all following members are aligned to 32-bit boundaries.

pr_SegList contains an array of *segments* containing AmigaDOS functions. The first entry in this array is a 32-bit integer indicating the number of valid elements, the remaining entries are *BPTRs* to segments of AmigaDOS and the loaded binary. Some entries may be ZERO indicating that the corresponding entry is currently not used. Segments are explained in more detail in section **??**. Typically, entries 1 and 2 are system segments containing AmigaDOS functions, entry 3 is used for the loaded binary, and entry 4 the segment of the shell. This, however, only reflects the current usage of segments, and later versions of AmigaDOS may populate this vector differently. The segments contained in this vector are used by the AmigaDOS runtime binder to build the *Global Vector* of processes using BCPL linkage. As BPCL is phased out, this vector is of no particular importance today anymore, and can be ignored for almost all purposes. The only exception is the *Shell* which shall prepare this vector to ensure that commands written in BCPL function properly. More on this in section **??**.

pr_StackSize is the size of the process stack in bytes. It is always a multiple of 4 bytes long.

pr_GlobVec is another BCPL legacy. It contains the *Global Vector* of the process. For binaries using the BCPL linkages, this is a custom-build array of global data and function entry points from pr_SegList. For C and assembler binaries, the *Global Vector* is the system shared vector; it contains *dos.library* global data required by some of its functions, such as base pointers to system libraries. As no particular advantage can be taken from this vector (anymore) as all functions available in it are also available as *dos.library* entry points, it should be left alone.

pr_TaskNum is an integer allocated by the system for processes that execute a shell, or are binaries that have been launched by the shell. The number here corresponds to the integer printed by the Status command. Note that AmigaDOS does not use task numbers consistenty, i.e. processes that are started from the workbench or have been created by some other means are not identified by a task number. In such a case, this member remains 0.

pr_StackBase is a *BPTR* to the address of the lower end of the stack, i.e. the end of the C or assembler stack. As the BCPL stack grows in opposide direction, it is the start of the BCPL stack. While it is initialized, it is not used by the *dos.library* at all.

pr_Result2 is the secondary result code set by many functions of the *dos.library*. The value stored here is delivered by IoErr().

pr_CurrentDir is the *lock* representing the current directory of the process. All relative paths are resolved from this *lock*, i.e. they are relative to pr_CurrentDir. If this member is ZERO, the currrent directory is the root directory of the file system stored in pr_FileSystemTask. As the latter is (unless altered) the file system of the boot volume, this is usually identical to the directory identified by the SYS assign.

pr_CIS is *file handle* of the standard input stream of the process. It is also returned by Input(). It can be ZERO in case the process does not have a standard input stream. This is *not* equivalenqt to a NIL: input handle — in fact, any attempt to read from a non-existing input stream will crash. Processes started from the workbench do not have an input stream, unless one is installed here with SelectInput().

pr_COS is the *file handle* of the standard output stream of the process. It is also returned by the Output() function of the *dos.library*. It can be ZERO in case the process does not have a standard output stream, which is not equivalent to a NIL: file handle. Any attempt to output to ZERO will crash the

system. Processes started from the workbench do not have an output stream, unless one is installed with `SelectOutput()`.

`pr_ConsoleTask` is the *MsgPort* of the console within which this process is run, if such a console exists. This *handler* is contacted when opening "`*`" or a path relative to `CONSOLE:`. Processes started from the workbench do not have a console, unless one is installed with `SetConsoleTask()`.

`pr_FilesSystemTask` is the *MsgPort* of the file system that is contacted in case a relative path is to be resolved relative to the `ZERO` lock. This member is initialized to the *MsgPort* of the file system the system was booted from, but can be changed by `SetFileSysTask()`. This member is also returned by `GetFileSysTask()`.

`pr_CLI` is a *BPTR* to the `CommandLineInterface` structure containing information on the Shell this process is running in. If this process is not part of a Shell, this member is `ZERO`. This is for example the case for programs started from the workbench, or *handler* or *file system*.

`pr_ReturnAddr` is another *BCPL* legacy and should not be used by new implementations. It points to the BCPL stack frame of the process or the command overloading the process, and used there to restore the previous stack frame for the `Exit()` function. This is typically the process cleanup code for processes initialized by `CreateProc()` or `CreateNewProc()`, or the shell command shutdown code placed there by `RunCommand()`. This cleanup process does not, however, release any other resources obtained by user code. BCPL code or custom startup code could deposit here pointer to a BCPL stack frame for a custom shutdown mechanism.

The BCPL stack frame is described by the following (undocumented) structure:

```
struct BCPLStackFrame {
       ULONG bpsf_StackSize;
       APTR  bpsf_PreviousStack;
};
```

where `bpsf_StackSize` is the stack size of the current (active) stack, and `bpsf_PreviousStack` the stack of the caller; to restore the previous stack, this value is placed in the CPU register `A7`.

`pr_PktWait` is a function that is called when waiting for inter-process communication, in particular when waiting for a returning packet set out to a handler. If this is `NULL`, the system default function is used. The signature of this function is

```
msg = (*pr_PktWait)(void)
D0

struct MsgPort *(*pr_PktWait)(void)
```

that is, no particular arguments are delivered, the process must be obtained from `exec`, and the message received shall be delivered back into register `D0`. The returned pointer shall not be `NULL`, rather, this function shall block until a message has been received. For details, see the `DoPkt()` function and section **??**.

`pr_WindowPtr` is, unlike what the name suggests, a pointer to an *intuition* `Screen` structure, see `intuition/screens.h`, on which error requesters will appear. If this is `NULL`, error requesters appear on the workbench screen, and if this is set to `(APTR)(-1L)`, error requesters will be suppressed at all, and the implied reponse to them is to cancel the operation. This error requester is specified in more detail with the `ErrorReport()` function.

`pr_HomeDir` is the *lock* to the directory containing the binary that is currently executed as this process, if such a directory exists. It is `ZERO` if the binary is resident. This *lock* is filled in by the Shell or the Workbench when loading and starting a process. It is used to resolve paths relative to the `PROGDIR` pseudo-assign, see section 5. If this lock is `ZERO`, any attempt to resolve a path within `PROGDIR:` will create a request to inserted a volume `PROGDIR:`, which is probably not a very useful reaction of AmigaDOS.

pr_Flags are system-use only flags that shall not be used or interpreted. They are used by the system process shutdown code to identify which resources need to be released, but future systems may find additional uses for this member.

pr_ExitCode() is a pointer to a function that is called by AmigaDOS as part of the process shutdown code, and as such quite more useful that pr_ReturnAddr. Thie function prototype is as follows:

```
returncode = ExitFunc(rc,exitdata)
D0                    D0 D1

LONG ExitFunc(LONG,LONG)
```

The value of rc is the return code process, i.e. the value left in register D0 when the code drops off the final RTS, and exitdata is taken from pr_ExitData. The returncode is a modified version of the process return code that is, however, ignored.

pr_ExitData is used as argument for the pr_ExitCode() function, see above.

pr_Arguments is a pointer to the command line arguments of the process if it corresponds to a command started from the Shell. This is a NUL terminated string. This argument string can also be found in register A0 for programs started from the Shell, or in the buffer of pr_CIS. The ReadArgs() function takes it from the latter source, and not from pr_Arguments. Otherwise, this member remains NULL.

pr_LocalVars is a MinList structure, as defined in exec/lists.h, that contains local variables specific to the shell within which the process is executed, if any. The structure of such variables is defined in dos/var.h. This structure is specified in section **??**.

pr_ShellPrivate is reserved for the Shell and its value shall not be used, modified or interpreted. It is currently unused, but can be used by future releases.

pr_CES is the *file handle* to be used for error output. This stream gues usually to the console the process runs in, if such a console exists. This handle can be changed by SelectError(). If pr_CES is NULL, processes should fall back to pr_COS for printing errors. Preferably, processes should use the ErrorOutput() function to obtain an error stream, though.

8.1 Creating and Terminating Processes

AmigaDOS provides several functions to create functions: CreateNewProc() is the revised and most flexible function for launching a process, taking many parameters in the form of a tag list. The legacy functionCreateProc() supports less options, but available under all Os versions. Shells as created by the System() function implicitly also create processes, but are not discussed here, but in section **??**. Therefore, System() shares a couple of options with CreateNewProc().

There is surprisingly not a single function to delete processes. Processes die whenever their execution drops off at the end of the main() function, or whenever execution reaches the final RTS instruction of the main program function. The Exit() function also terminates a process, but shall be called from within the process, and is typically not suitable as it does not release resouces accquired by the program itself, but only those allocated by the system itself.

8.1.1 Creating a New Process from a TagList

The CreateNewProc() function takes a TagItem array as defined in utility/tagitem.h and launches a new process from this list. The tags this function takes are defined in dos/dostags.h.

```
process = CreateNewProc(tags)
D0                      D1
```

```
struct Process *CreateNewProc(struct TagItem *)

process = CreateNewProcTagList(tags)
D0                             D1

struct Process *CreateNewProcTagList(struct TagItem *)

process = CreateNewProcTags(Tag1, ...)

struct Process *CreateNewProcTags(ULONG, ...)
```

The above functions are all equivalent, just the calling conventions are different. For `CreateNewProcTags()`, the `TagList` is created by the compiler on the stack and a pointer is then implicitly passed into the function. The following tags are recognized by the function:

`NP_Seglist` takes a *BPTR* to a segment list as returned by `LoadSeg()` and launches the process at the first byte of the first segment of the list.

`NP_FreeSeglist` is a boolean indicator that defines whether the segment provided to `NP_Seglist` is released when the process terminates. Unlike what the official documentation claims, the default value of this tag is `DOSFALSE`, i.e. the segment is *not* released.

`NP_Entry` is mutually exclusive to `NP_Seglist` and defines an absolute address (and not a segment) as entry point of the process to be created. If this tag is provided, then `NP_FreeSeglist` shall *not* be set a non-zero value. Either `NP_Entry` or `NP_Seglist` shall be included.

`NP_Input` sets the input file handle, i.e. `pr_CIS` of the process to be created. This tag takes a *BPTR* to a *file handle*. The default is *not* to set the input file handle, e.g. to leave it `ZERO`.

`NP_CloseInput` selects whether the input file handle, if provided, will be closed when the process terminates. If non-zero, the input file handle will be closed, otherwise it remains opened. The default is to close the input file handle.

`NP_Output` sets the output file handle, i.e. `pr_COS` of the process to be created. This tag takes a *BPTR* to a *file handle*. The default is to leave the output at `ZERO`.

`NP_CloseOutput` selects whether the output file handle, if provided, will be closed when the process terminates. If non-zero, the output file handle will be closed, otherwise it remains open. The default is to close the output file handle.

`NP_Error` sets the error file handle, i.e. `pr_CES` of the process to be created. This tag also takes a *BPTR* to a *file handle*. The default is to leave the error output handle at `ZERO`.

`NP_CloseError` selects whether the rror file handle, if provided, will be closed when the process terminates. If non-zero, the error file handle will be closed, otherwise it remains open. The default is *not* to close the error file handle. This (different) default is to ensure backwards compatibility.

`NP_CurrentDir` sets the current directory of the process to be created. The argument is a *Lock*. The default is to duplicate the current directory of the caller with `DupLock()` if the caller is a process, or leave the current directory at `ZERO`. The current directory of the process, i.e. `pr_CurrentDir`, is released when the process terminates, unless `NP_CurrentDir` is set to `ZERO`.

`NP_StackSize` sets the stack size of the process to be created in bytes. The default is a stack size of 4000 bytes.

`NP_Name` is a pointer to a `NUL` terminated string to which the task name of the process to be created is set. This string is copied before the process is launched, and the copy is released automatically when the process terminates. The default process name is "`New Process`".

`NP_Priority` sets the priority of the process to be created. The tag value shall be an integer in the range -128 to 127, though useful values are in the range of 0 to 20. The default is 0.

NP_ConsoleTask specifies a pointer to a *MsgPort* to the handler that is responsible for the console of the process to be created. That is, if the created process opens "*" or a path relative to CONSOLE:, it will use the specified handler. The default is to use the console handler if the caller is a process, or NULL if the caller is only a task.

While not explicitly available as a tag, the default file system of the created process, i.e. pr_FileSystemTask, is set to the default file system of the calling process if the caller is a process, or otherwise use the default file system from the *dos.library*. This file system is contacted to resolve paths relative to the ZERO lock.

NP_WindowPtr specifies a pointer to a Screen on which error requesters will be displayed, 0 to display requesters on the workbench, or −1 to suppress error requesters. It will be installed in the pr_WindowPtr of the process to be created. The default is to copy the pointer from the calling process if the window pointer of the parent is 0 or −1. The tag does not copy any other value of pr_WindowPtr from the parent. To set the pr_WindowPtr of the created process to the value of the calling process, the tag must be explicitly provided. If called from a task and not a process, the default is NULL. The reason why pr_WindowPtr is not explicitly copied is that the caller shall ensure that the screen is not closed while any pointers are still pointing to its structure.

NP_HomeDir sets the pr_HomeDir *lock* which is used to resolve paths relative to the PROGDIR: pseudo-assign. The default is to copy pr_HomeDir of the calling process, or ZERO in case the caller is a task. This *lock* is released when the process terminates, i.e. the *lock* provided as argument here remains available to the caller, and shall be released by the caller in one way or another.

NP_CopyVars determines if the local shell variables in pr_LocalVars of the calling process are copied into the variables of the process to be created. If set to non-zero, a copy of the variables of the calling process are made, otherwise the new process does not receive any shell variables by itself. The latter also happens if the caller is a task and not a process. The variables are automatically released when the new process terminates.

NP_Cli determines whether the new process will receive a new shell environment in the form of a CommandLineInterface structure. If non-zero, a new CLI structure will be created and a *BPTR* to this structure will be filled into the pr_CLI member of the process to be created. The new shell environment will be a copy of the shell environment of the caller if one is present, or a shell environment initialized with all defaults. This means that the prompt, the path, and the command name will be copied over. If 0, no such environment will be created. The latter is also the default.

NP_Path provides a chained list of *locks* within which commands are searched. This is the same list the PATH command adjusts, see section **??** for details on this structure. This tag only applies if NP_Cli is non-zero to create a shell environment. This chained list is *not* copied, and will be released when the created process terminates; hence, the locks provided here are *no longer* available to the caller if CreateNewProc() succeeds. If CreateNewProc() fails, the entire lock list remains a property of the caller and thus needs to be potentially released there. The default, if this tag is not provided, is to copy the paths of the caller if the calling process has a non-zero pr_CLI structure.

NP_CommandName provides the name of the command being executed within the shell environment if NP_Cli indicates that one is to be created. The default is to copy the command name of the shell environment of the calling process if one exists, or to leave the command name empty if none is provided. The command name is copied into the shell environment of the process being created and thus remains available to the caller. More on the shell environment is found in section **??**.

NP_Arguments provides command line arguments for the process to be created. This is a NUL terminated string that is copied into the process to be created, and will also be released there. If provided, the arguments are copied in pr_Arguments of the process to be created, and will also be loaded into registers A0 and its length into D0. If NP_Arguments are non-zero, a non-ZERO NP_Input file handle shall also provided. This is because the arguments are also copied into the buffer associated to the input *file handle* to make them available to ReadArgs(), or any other function that performs buffered read from pr_CIS, see section 3.8 for details.

NP_ExitCode determines a pointer to function that is called when the created process terminates. This pointer is filled into pr_ExitCode. See section 8 for the description and the signature of this function.

NP_ExitData provides an argument that will be passed into the NP_ExitCode function in register D1 when the process terminates.

While the official documentation also mentiones the tags NP_NotifyOnDeath and NP_Synchronous, these tags are currently ignored and do not perform any function.

The CreateNewProc() function returns on success a pointer to the Process structure just created. At this stage, the process has already been launched and, depending on its priority, may already be running. On failure, the function returns NULL. Unfortunately, it does not set IoErr() consistently on failure.

8.1.2 Create a Process (Legacy)

The CreateProc() function creates a process from a segment list, a name, a priority and a stack size. It is a legacy call that is not as flexible as CreateNewProc(), and only exists for backwards compatibility reasons.

```
process = CreateProc( name, pri, seglist, stackSize )
D0                    D1    D2   D3       D4

struct MsgPort *CreateProc(STRPTR, LONG, BPTR, LONG)
```

This function creates a process of the name name running at priority pri. The process starts at the first byte of the first segment of the segment list passed in as seglist, and a stack size of stackSize bytes will be allocated for the process.

The process is initialized as follows: pr_ConsoleTask and pr_WindowPtr are copied from the calling process, or are set to NULL respective 0 if called from the task. The member pr_FileSysTask is also copied from the calling process, or is initialized from the default file system from the *dos.library* if called from a task.

Input, output and error file handles are set to ZERO, and no shell environment is created either. The current directory and home directory are also left at ZERO. No arguments are provided to the called function, and no shell variables are copied.

If the call succeeds, the returned value process is a pointer to the *MsgPort* of the created process. It is *not* a pointer to a process itself.

On failure, the function returns NULL. Unfortunately, it does not set IoErr() consistently in case of failure, thus the cause of the problem cannot be easily identified.

8.1.3 Terminating a Process

The Exit() function terminates the calling process or the calling command line executable. In the latter case, control is returned to the calling shell, in the former case, the process is removed from the exec scheduler.

However, tis function does not release any resources except those implicitly allocated when creating the process through CreateNewProc(), CreateProc() or RunCommand() and the calling shell. As it misses to release resources allocated by you or the compiler startup code, this function *should not be used* and rather a compiler or language specific shutdown function should be preferred. The C standard library provides exit() which releases resources allocated through this library.

```
Exit( returnCode )
      D1

void Exit(LONG)
```

This call either terminates the calling process, in which case the argument is ignored, or returns to the calling shell, then delivering `returnCode` as result code. It uses the BCPL stack frame pointed to by `pr_ReturnAddr`, removes this stack frame, initializes the new stack from the stack frame there and then returns to whatever created the stack frame. This is typically either the process shutdown code of AmigaDOS, or the shell command shutdown code installed by `RunCommand()`. In the former code, `pr_ExitCode()` may be used to implement additional cleanup activities.

This function is a BCPL legacy function that is also part of the *Global Vector*; BCPL programs would typically overload its entry in this vector to implement a custom shutdown mechanism.

8.2 Process Properties Accessor Functions

The most important members of the process structure described in section 8 are accessible through getter and setter functions. They implicitly relate to the calling process, and are the preferred way of getting access to the `Process` structure. The functions listed in this section do not touch `IoErr()` except explicitly stated.

8.2.1 Retrieve the Process Input File Handle

The `Input()` function returns the input file handle of the calling proces if one is installed. If no input file handle is provided, the function returns `ZERO`.

```
file = Input()
D0

BPTR Input(void)
```

This function returns a *BPTR* to the input *file handle* of the calling process, or `ZERO` if none is defined. This is approximately identical to `stdin` of ANSI-C. Depending on process creation, this file handle can be closed by the process shutdown code or the calling shell and thus should in general not be closed explicitly. It can be changed through `SelectInput()`.

8.2.2 Replace the Input File Handle

The `SelectInput()` function replaces the input *file handle* of the calling process with its argument and returns the previously used input handle.

```
old_fh = SelectInput(fh)
D0                   D1

BPTR SelectInput(BPTR)
```

This call replaces the input *file handle* of the calling process with the file handle given by `fh` and returns the previously used input *file handle*.

8.2.3 Retrieve the Ouput File Handle

The `Output()` function returns the output file handle of the calling proces if one is installed. If no output file handle is provided, the function returns `ZERO`.

```
file = Output()
D0

BPTR Output(void)
```

This function returns a *BPTR* to the output *file handle* of the calling process, or ZERO if none is defined. This is approximately identical to stdout of ANSI-C. Depending on process creation, this file handle can be closed by the process shutdown code or the calling shell and thus should in general not be closed explicitly. It can be changed through SelectOutput().

8.2.4 Replace the Output File Handle

The SelectOutput() function replaces the output *file handle* of the calling process with its argument and returns the previously used output handle.

```
old_fh = SelectOutput(fh)
D0                    D1

BPTR SelectOutput(BPTR)
```

This call replaces the output *file handle* of the calling process with the file handle given by fh and returns the previously used output *file handle*.

8.2.5 Retrieve the Error File Handle

The ErrorOutput() function returns the file handle through which diagnostic or error outputs should be printed. It uses either pr_CES if this handle is non-ZERO, or pr_COS if the former is ZERO. If neither an error ouptut nor a regular output is provided, this function returns ZERO.

```
file = ErrorOutput()
D0

BPTR ErrorOutput(void)
```

This function returns a *BPTR* to the error *file handle* of the calling process, or falls back to the *BPTR* of the output *file handle* if the former is not available. This is the file handle through which diagnostic output should be printed and is therefore approximately identical to stderr of ANSI-C. Depending on process creation, this file handle can be closed by the process shutdown code or the calling shell and thus should in general not be closed explicitly. It can be changed through SelectError().

8.2.6 Replace the Error File Handle

The SelectError() function replaces the error *file handle* of the calling process with its argument and returns the previously used error handle.

```
old_fh = SelectError(fh)
D0                   D1

BPTR SelectError(BPTR)
```

This call replaces the error *file handle* of the calling process with the file handle given by fh and returns the previously used error *file handle*.

8.2.7 Retrieve the Current Directory

The GetCurrentDir() function retruns the current directory of the directory, indicated by a *lock* on this object. This *lock*, and the *file system* that created the lock are used to resolve relative paths, see also section 3.3.

```
lock = GetCurrentDir(void)
D0

BPTR GetCurrentDir()
```

This function returns the *lock* to the current directory, unlike the `CurrentDir()` function which also changes it.

8.2.8 Replace the Current Directory

The `CurrentDir()` selects and retrieves the current directory of the calling process. The directory is indicated by a *lock* to this object. This *lock*, and the *file system* that created the lock are used to resolve relative paths, see also section 3.3.

```
oldLock = CurrentDir( lock )
D0                    D1

BPTR CurrentDir(BPTR)
```

This function sets the current directory to `lock` and returns in `oldLock` the previously installed current directory. The passed in `lock` then becomes part of the process and shall not be released by `UnLock()` until another *lock* is installed as current directory.

If the current directory is `ZERO`, paths are relative to the root directory of the *file system* set in the `pr_FileSysTask` member of the calling process. It may be changed by `SetFileSysTask()` described in section 8.2.12. AmigaDOS installs there the *file system* of the boot volume, unless a user installs a different default *file system*.

8.2.9 Return the Latest Error Code

The `IoErr()` function returns the secondary result code of the most recent AmigaDOS operation. This code is, in case of failure, typically an error code indicating the nature of the failure.

```
error = IoErr()
D0

LONG IoErr(void)
```

This function returns the secondary result code of the last call to the *dos.library* that provides such result. Unfortunately, not all functions set `IoErr()` consistently; all unbuffered operations in section 3.7 provide an error code in case of failure, or deliver 0 as secondary result in case of success. The buffered functions in section 3.8 generally only set a secondary result code in case an I/O operation is required, but do not touch `IoErr()` if the call can be satisfied from the caller. Whether a function of the *dos.library* touches `IoErr()` is stated in the description of the corresponding function — unfortuantely, the *dos.library* does not handle `IoErr()` consistently.

Some functions provide a secondary result code different from an error code, and thus make such additional return value available through `IoErr()`. Such additional return values are also explicitly mentioned in the description of the corresponding function. A particular example is `DeviceProc()`, which returns the (first) lock of a regular assign in `IoErr()`, but additional functions exist.

Most error codes are defined in `dos/dos.h`, with some additional error codes only used by the pattern matcher (see section 7) in `dos/dosasl.h`. Generally, *handlers* and *file systems* can select error codes as they seem fit, the list below provides a general indication how the codes are used by the *dos.library* itself, or what their suggested usage is:

ERROR_NO_FREE_STORE: This error code is set if the system run out of memory. Actually, this error code is not set by the *dos.library*, but rather by the *exec.library* memory allocation functions.

ERROR_TASK_TABLE_FULL: This error code is no longer in use. Previous releases of AmigaDOS created it if more than 10 shell processes were about to be created. As this limitation was removed, the error code remains currently unused.

ERROR_BAD_TEMPLATE: This error code indicates that the command line template for `ReadArgs()` is syntactical incorrect. It is also set by the pattern matcher in case the pattern is syntactically incorrect.

ERROR_BAD_NUMBER: This error code indicates that a string could not be converted to a number.

ERROR_REQUIRED_ARG_MISSING: This error code is set by `ReadArgs()` if a non-optional argument is not provided.

ERROR_KEY_NEEDS_ARG: This error code is also used by the argument parser `ReadArgs()` if an argument key is provided on the command line, but a corresponding argument value is missing.

ERROR_TOO_MANY_ARGS: This error code can also be set by `ReadArgs()`; it indicates that more arguments are provided than indicated in the template.

ERROR_UNMATCHED_QUOTES: This error code indicates that a closing quote is missing for at least one opening quote. It is also set by the argument parser and `ReadItem()`.

ERROR_LINE_TOO_LONG: This error code is a general indicator that a user provided buffer is too small to buffer a string. It is for example used again by the argument parser and the path manipulation functions in section 5.3.

ERROR_FILE_NOT_OBJECT: This error code is generated by the *Shell* if an attempt is made to execute a file that is neither a script, nor an executable nor a file that can be opened by a viewer.

ERROR_INVALID_RESIDENT_LIBRARY: While this error code is not in use by the *dos.library*, several *handlers* and other Os components use it to indicate that a required library or device is not available.

ERROR_NO_DEFAULT_DIR: This is error code is also not in use. Its intended purpose is unclear.

ERROR_OBJECT_IN_USE: This error code is used by multiple Os components to indicate that a particular operation cannot be performed because the object to be modified is in use. AmigaDOS uses it, for example, to indicate that a *lock* was obtained on an object that is supposed to be modified or deleted, and thus cannot be modified or removed.

ERROR_OBJECT_EXISTS: This error code is a generic error indicator that an operation could not be performed because another object already exists in place, and is used as such by multiple Os components. AmigaDOS *file systems* use it, for example, when attempting to create a directory, but a file or a directory of the requested name is already present.

ERROR_DIR_NOT_FOUND: This error code indicates that the target directory is not found. Of the AmigaDOS ROM components, only the shell uses it on an attempt to change the working directory to a non-working target directory.

ERROR_OBJECT_NOT_FOUND: This is a generic error code that indicates that the object on which a particular operation is to be performed does not exist. It is for example generated on an attempt to open a non-existing file or to lock a file or directory that could not be found.

ERROR_BAD_STREAM_NAME: This error code is currently not in use by AmigaDOS ROM components. Its purpose is unclear.

ERROR_OBJECT_TOO_LARGE: This error could be used to indicate that an object is beyond the size a *handler* or *file system* is able to handle. Note that a full disk (or full storage medium) is indicated by ERROR_DISK_FULL, and not this error. However, currently no AmigaDOS component uses this error, even though the FFS should probably return it on an attempt to create or access files larger than 2GB.

ERROR_ACTION_NOT_KNOWN: This is a generic error code that is returned by many *handlers* or *file systems* when an action (in the form of a *packet*) is requested the handler does not support or understand. For example, this error is created when attempting to create a directory on a console handler.

ERROR_INVALID_COMPONENT_NAME: This is an error that is raised by file systems when providing an invalid path, or a path that contains components that are syntactically incorrect. For example, the colon (“:”) shall only appear one in a path as separator between the device name and the path within the device. A colon within a component is therefore a syntactical error. Also, all Amiga ROM file systems do not accept code points below 0x20, i.e. ASCII control characters.

ERROR_INVALID_LOCK: This error is raised if a value is passed in as a *lock* that is, in fact, not a valid lock of the target *file system*. For example, an attempt to use a *file handle* as a lock will result in such an error condition. Note, however, that *file systems* can, but do not need to check locks for validity. Passing incorrect objects to *file systems* can raise multiple error conditions of which this error code is probably the most harmless.

ERROR_OBJECT_WRONG_TYPE: This error code indicates that a particular operation is not applicable to a target object, even though the target object is valid and existing. For example, an attempt to open an existing directory for reading as a file will raise this error.

ERROR_DISK_NOT_VALIDATED: This error indicates that the inserted medium is currently not validated, i.e. not checked for consistency. Such a consistency check (or validation) may be currently ongoing. This error is for example generated if a write operation is attempted on an FFS volume whose validation is still ongoing. In such a case, retrying the operation later may solve the problem already.

ERROR_DISK_WRITE_PROTECTED: This indicates that an attempt was made to write to a medium, e.g. a disk, that is write-protected, or that cannot be written to, such as an attempt to write to a CD-ROM.

ERROR_RENAME_ACROSS_DEVICES: Generated if an attempt is made to move an object to a target directory that is located on a different medium or different *file system* than the source directory. This cannot succeed, instead the object (and its subobjects) need to be copied manually.

ERROR_DIRECTORY_NOT_EMPTY: Indicates that an attempt was made to delete a directory that is not empty. First, all the files within a directory must be deleted before the directory itself may be deleted.

ERROR_TOO_MANY_LEVELS: This error code is generated if too many softlinks refer iteratively to other softlinks. In order to avoid an endless indirection of softlinks refering to each other, the *dos.library* aborts following softlinks after 15 passes; application programs attempting to resolve softlinks themselves through ReadLink() should implement a similar mechanism, see also section 5.4.2.

ERROR_DEVICE_NOT_MOUNTED: This error indicates that an access was attempted to either a *handler*, *file system* or *assign* that is not known to the system, or to a volume that is currently not inserted in any known drive.

ERROR_SEEK_ERROR: This error is generated by an attempt to Seek() to a file position that is either negative, or behind the end of the file. It is also signalled if the mode of Seek() or SetFileSize() is none of the modes indicated in table 8. The FFS also sets this mode if it cannot read one of its administration blocks.

ERROR_COMMENT_TOO_BIG: This error is raised if the size of the comment is too large to be stored in in the metadata of the *file system*. Note that while *file systems* shall validate the size of the comment, it shall silently truncate file names to the maximal size possible.

ERROR_DISK_FULL: Generated by *file systems* when an attempt is made to write more data to a medium than it is possible to hold, i.e. when the target medium is full.

ERROR_DELETE_PROTECTED: This error is generated by *file systems* if an attempt is made to delete a file that is delete protected, i.e. whose FIBB_DELETE protection bit is set, see table 15 in section 5.1.

ERROR_WRITE_PROTECTED: This error is generated by *file systems* if a write is attempted to a file that is write protected, i.e. whose FIBB_WRITE bit is set.

ERROR_READ_PROTECTED: This error is generated on an attempt to read from a while whose FIBB_READ bit is set to indicate read protection.

ERROR_NOT_A_DOS_DISK: This error is generated by a *file system* on an attempt to read a disk that is not strutured according to the requirements of the *file system*, i.e. that is initialized by another incompatbile

file system different from the mounted one. Unfortunately, AmigaDOS does not have a control instance that selects file systems according to the disk layout.

`ERROR_NO_MORE_ENTRIES`: This secondary result code does not really indicate an error condition, it just reports to the caller that the end of a directory has been reached when scanning it by `ExNext()` or `ExAll()`.

`ERROR_IS_SOFT_LINK`: This error code is generated by *file systems* on an attempt to access a *soft link*. For many functions, the *dos.library* recognizes this error and then resolves the link through `ReadLink()` within the library, not requiring intervention of the caller. However, not all functions of the *dos.library* are aware of *soft links*, see section 5.4 for the list.

`ERROR_OBJECT_LINKED`: This error code is currently not used by AmigaDOS and its intended use is not known.

`ERROR_BAD_HUNK`: Generated by `LoadSeg()` and `NewLoadSeg()`, this error code indicates that the binary file includes a hunk type that is not supported or recognized by AmigaDOS. The hunk format for binary executables is documented in section **??**.

`ERROR_NOT_IMPLEMENTED`: This error code is not used by any ROM component, but several workbench components signal this error indicating that the requested function is not supported by this component. For example, the `Format` command generates it on an attempt to format a disk with long file names if the target file system does not support them.

`ERROR_RECORD_NOT_LOCKED`: Issued by *file systems* and their record-locking subsystem if an attempt is made to release a record that is, actually, not locked.

`ERROR_LOCK_COLLISION`: This error is also created by the record-locking subsystem of *file systems* if attempt is made to exclusively lock the same region within a file by two write locks.

`ERROR_LOCK_TIMEOUT`: Also generated by the record-locking mechanism of *file systems* if an attempt was made to exclusively lock a region of a file that is exclusively locked already, and the attempt failed because the region did not became available before the lock timed out.

`ERROR_UNLOCK_ERROR`: This error is curently not generated by any *file system*, though could be used to indicate that an attempt to unlock a record failed for an unknown reason.

`ERROR_BUFFER_OVERFLOW`: This error is raised by the pattern matcher and indicates that the buffer allocated in the `AnchorPath` structure is too small to keep the fully expanded matching file name, see also section **??**.

`ERROR_BREAK`: This error is also raised by the pattern matcher if it received an external signal for aborting a directory scan for objects. Such signals are raised, for example, by the user through the console by pressing `Ctrl + C` through `Ctrl + F`.

`ERROR_NOT_EXECUTABLE`: This error is generated by the workbench on an attempt to start an application icon from a file whose FIBB_EXECUTE is set, indicating that the file is not executable. Why the workbench does not use the same error code as the *Shell* remains unclear.

8.2.10 Setting IoErr

The `SetIoErr()` function sets the value returned by the next call to `IoErr()` and thus initializes or resets the next IO error.

```
oldcode = SetIoErr(code)
D0                 D1

LONG SetIoErr(LONG);
```

This function sets the next value returned by `IoErr()`; this can be necessary because some functions of the *dos.library* do not update this value in all cases. A particular example are the buffered I/O functions

introduced in section **??** that do not touch `IoErr()` in case the input or output operation can be satisfied from the buffer. A good practise is to call `SetIoErr(0)` upfront to ensure that these functions leave a 0 in `IoErr()` on success.

This function returns the previous value of `IoErr()`, and thus the same value `IoErr()` would return.

8.2.11 Select the Console Handler

The `SetConsoleTask()` function selects the *handler* responsible for the "`*`" file name and `CONSOLE:` pseudo-device.

```
oldport = SetConsoleTask(port)
D0                       D1

struct MsgPort *SetConsoleTask(struct MsgPort *)
```

This function selects the *MsgPort* of the console handler. AmigaDOS will contact this handler for opening the "`*`" as file name, or a file relative to the `CONSOLE:` pseudo-device. Note that the argument is not a pointer to the *handler* process, but rather to a *MsgPort* through which this process can be contacted. It returns the previously used console handler *MsgPort*.

This function is the setter function corresponding to the `GetConsoleTask()` getter function introduced in section 6.1.4.

8.2.12 Select the Default File System

The `SetFileSysTask()` function selects the handler responsible for resolving paths relative to the `ZERO` lock.

```
oldport = SetFileSysTask(port)
D0                       D1

struct MsgPort *SetFileSysTask(struct MsgPort *)
```

This function selects the *MsgPort* of the default *file system*. AmigaDOS will contact this *file system* if a path relative to the `ZERO` lock is resolved, e.g. a relative path name if the current directory is `ZERO`. This *file system* should be identical to the *file system* of the `SYS:` assign, and should therefore not be relaced as otherwise resolving file names may be inconsistent between processes.

Note that the argument is not a pointer to the *handler* process, but rather to a *MsgPort* through which this process can be contacted. It returns the previously used default file system *MsgPort*. This function is the setter equivalent of `GetFileSysTask()` introduced in section 6.1.5.

Chapter 9

Binary File Structure

The AmigaDOS *Hunk* format represents executable and linkable object files. While both formats are related, they are not identical; executables can be loaded from the shell or the workbench from disk to RAM, and then either overload the shell process, or a new process is created from them. Object files are created as intermediate compiler outputs; typically, each translation unit is compiled into one object file which are then, in a final step, linked with a startup code and object code libraries to form an executable.

An object or executable file in this format consists of multiple *hunks* (thus, the name). Hunks define either payload data as indivisible *segments* of code or data that is initialized or loaded from disk, or additional meta-information interpreted by the AmigaDOS loader, the `LoadSeg()` function. The meta-information is used to relocate the payload to their final position in memory, to define the size of the sections, to select the memory type that is allocated for the segment, or to interrupt or terminate the loading process.

Loaded executables are represented as singly linked list of segments in memory, by a structure that looks as follows:

```
struct LoadedSegment {
    BPTR  NextSegment; /* BPTR to next segment or ZERO */
    ULONG Data[1];     /* Payload data */
};
```

The above structure is *not* documented and is not identical to the `Segment` in `dos/dosextens.h`. The latter describes a resident executable, see section **??**, but also contains a *BPTR* to a segment in the above sense. Each segment of a binary is allocated through `AllocVec()` which is sometimes helpful as it allows to retrieve size of the segment from the size of the allocated memory block.

The hunk format distinguishes three types of *segments*, each represented by a hunk: *code hunks* that should contain constant data, most notably executable machine code and constant data associated to this code, *data hunks* that contain (variable) data, and so called *BSS* hunks that contain data that is initialized to zero. Thus, the contents of *BSS* hunks is not represented on disk.

> *Const is not enforced* While *code hunks* should contain executable code and other constant data, and *data hunks* should contain variable data, nothing in AmigaDOS is able to enforce these conventions. In principle, *data hunks* may contain executable machine code, and *code hunks* may contain variable data. Note, however, that some third party tools may require programs to follow such conventions. Many commercial compilers structure their object code according to these conventions, or at least do so in their default configuration.

Additional *hunks* describe how to relocate the loaded code and data. Relocation means that data within the hunk is corrected according to the addresses this and other hunks are loaded to. The relocation process takes an offset into one hunk, and adds to the longword at this offset the absolute address of this or any other

hunk. That is, hunks on disk are represented as if their first byte is placed at address 0, and relocation adjusts longwords within hunks to the final positions in memory.

An extension of the executable file format is the *overlay format* also supported by `LoadSeg()`. Here, only a part of the file is loaded into memory, while the remaining parts are only loaded on demand, potentially releasing other already loaded parts from memory. Overlayed executables thus take less main memory, though requires the volume containing the executable available all the time.

AmigaDOS also contains a simple run-time binder that is only used by compiled BCPL code, or by code that operates under such requirements. The purpose of this binder is to populate the BCPL *global vector* of the loaded program. While this runtime binder implements a legacy protocol, certain parts of AmigaDOS still expect. These are *handlers* or *file systems* that use the `dol_GlobVec` value of 0 or -2, or corresponding `GlobVec` entry in the mount list. While new handlers should not use this BCPL legacy protocol, the ROM file system (the FFS) and the port-handler currently still depend (or require) it, despite not being written in BCPL. A second application of this run-time binding procol is the shell which also depends on BCPL binding.

9.1 Executable File Format

The hunk format of executable files consists of 4-byte (longword) hunk identifiers and subsequent data that is interpreted by the AmigaDOS loader according to the introducing hunk identifier. The syntax of such a file, and its hunks, is here presented in a pseudo-code, in three-column tables.

The first column identifies the number of bits a syntax element takes. Bits within a byte are read from most significant to least significant bit, and bytes within a structure that extends over multiple bytes are read from most significant to least significant bit. That is, the binary file format follows the big-endian convention. If the first column contains a question mark ("?"), the structure is variably-sized, and the number of removed bits is defined by the second column, or the section it refers to. If the first column is empty, no bits are removed from the file.

The second column either identifies the member of a structure to which the value removed from the stream is assigned, or contains pseudo-code that describes how to process the values parsed from the stream. These syntax elements follow closely the convention of the C language. In particular `if (`*cond*`)` formulates a condition that is only executed if *cond* is true, `else` describes code that is executed following an `if` clause that is executed if *cond* is false, and `do ... while(`*cond*`)` indicates a loop that continues as long as `cond` is non-zero, and that may alternatively be terminated by a `break` within the body of the loop. The expression `i++` increments an internal state variable `i`, and the expression `--j` decrements an internal state variable. The value of `i++` is the value of i before the increment, and the value of `--j` is the value of j after decrementing it.

The following pseudo-code describes the top-level syntax of a binary executable file AmigaDOS is able to bring to memory:

Table 21: Regular Executable File

Size	Code	Syntax
?	`HUNK_HEADER`	Defines all segments, see section 9.1.1 for details
	$i = t_{\texttt{num}}$	Start with the first hunk, $t_{\texttt{num}}$ is defined in the `HUNK_HEADER`
	`do {`	Repeat until all hunks done
2	$\hat{m}_t[i]$	These two bits are unused, but some utilities set it identical to $m_t[i]$, the memory type of the hunk, see 9.1.1
1	a_f	Advisory hunk flag.

29	`h`	This is the hunk type
	`if (EOF) break;`	Terminate loading on end of file
	`if (`a_f`) {`	Check for bit 29, these are advisory hunks
32	`l`	Read length of advisory hunk
32 × l		l long words of hunk contents ignored
	`}`	
	`else if (h == HUNK_END) i++;`	Advance to next segment, see 9.1.11
	`else if (h == HUNK_BREAK) break;`	Terminate loading an overlay, see 9.3.4
?	`else if (h == HUNK_NAME) parse_NAME;`	See section 9.1.8
?	`else if (h == HUNK_CODE) parse_CODE;`	See section 9.1.2
?	`else if (h == HUNK_DATA) parse_DATA;`	See section 9.1.3
?	`else if (h == HUNK_BSS) parse_BSS;`	See section 9.1.4
?	`else if (h == HUNK_RELOC32) parse_RELOC32;`	See section 9.1.5
?	`else if (h == HUNK_SYMBOL) parse_SYMBOL;`	See section 9.1.9
?	`else if (h == HUNK_DEBUG) parse_DEBUG;`	See section 9.1.10
?	`else if (h == HUNK_OVERLAY) { parse_OVERLAY; break }`	See section 9.3.3
?	`else if (h == HUNK_DREL32) parse_RELOC32SHORT;`	This is a compatibility kludge for some older versions of the *dos.library*, new tools should use `HUNK_RELOC32SHORT` instead, see section 9.1.6
?	`else if (h == HUNK_RELOC32SHORT) parse_RELOC32SHORT;`	See section 9.1.6
?	`else if (h == HUNK_RELRELOC32) parse_RELRELOC32;`	See section 9.1.7
	`else ERROR_BAD_HUNK;`	Everything else is invalid
	`} while(true)`	repeat until all hunks done

In particular, every executable shall start with the `HUNK_HEADER` identifier, the big-endian long-word `0x3f3`. The following stream contains long-word identifiers of which the first 2 bits are ignored and masked out. Some tools (e.g. the `Atom` tool by CBM) places there memory requirements similar to what is indicated in the `HUNK_HEADER`. They have there, however, no effect as the segments are allocated within the `HUNK_HEADER` and not at times the hunk type is encountered.

Bit 29 (`HUNKB_ADVISORY`) has a special meaning. If this bit is set, then the hunk contents is ignored. The size of such an *advisory* hunk is defined by a long-word following the hunk type.

Loading a binary executable terminates on three conditions. Either, if an end of file is encountered. This closes the file handle and returns to the caller with the loaded segment list. Or, if a `HUNK_BREAK` or `HUNK_OVERLAY` are found. This mechanism is used for *overlayed* files. In the latter two cases, the file

remains open, and for HUNK_OVERLAY, information on the loaded file is injected into the first hunk of the loaded data. More information on this mechanism is provided in section 9.3.

9.1.1 HUNK_HEADER

The HUNK_HEADER is the first hunk of every executable file. It identifies the number of segments in an executable, and the amount of memory to reserve for each segment.

Table 22: Hunk Header Syntax

Size	Code	Syntax
32	HUNK_HEADER [0x3f3]	Every executable file shall start with this hunk
32	0	Number of resident libraries, BCPL legacy, shall be zero
32	$t_{\mathtt{size}} \in [1, 2^{31}-1]$	Number of segments in binary
32	$t_{\mathrm{num}} \in [0, t_{\mathtt{size}}-1]$	First segment to load
32	$t_{\mathtt{max}} \in [t_{\mathrm{num}}, t_{\mathtt{size}}-1]$	Last segment to load (inclusive)
	for(i=t_{num};$i \leq t_{\mathtt{max}}$;i++) {	Iterate over all hunks
2	$m_t[i]$	Read memory type of the segment as 2 bits
30	$m_s[i]$	Read memory size in long words as 30 bits
	if ($m[i]$ == 3) {	if the memory type is 3
32	$m_s[i]$	Memory type is explicitly provided
	}	End of special memory condition
	$m_a[i]$ = AllocVec(sizeof(BPTR) + $m_s[i]\times$ sizeof(LONG),$m_t[i]$| MEMF_PUBLIC) + sizeof(BPTR)	Get memory for segment
	}	End of loop over segments

The first member of a HUNK_HEADER shall always be 0; it was used by a legacy mechanism which allowed run-time binding of the executable with dynamic libraries. While first versions of AmigaDOS inherited this mechanism from TRIPOS, but it was not particularly useful as the calling conventions for such libraries did not follow the usual conventions of AmigaDOS, i.e. with the library base in register a6. Later versions of AmigaDOS, in particular its re-implementation as of Kickstart v37, removed support for such libraries. As this mechanism is no longer supported, it is not documented here. More information is found in [10].

The second entry t_{size} contains the number of segments the executable consists of. In case of overlays, it is the total number of segments that can be resident in memory at all times. See section 9.3 for more information. This value shall be consistent for all HUNK_HEADERS within an overlayed file. In regular executables, only a single HUNK_HEADER exists at the beginning of the file.

The members t_{num} and t_{max} define the 0-based index of the first and last segment to load within the branch of the overlay tree described by this HUNK_HEADER. For a regular (non-overlayed) file and for the root node of the overlay tree, t_{num} shall be 0, that is, the first segment to load is 0, the first index in the segment table.

For regular files, t_{max} shall be identical to $t_{\mathrm{size}} - 1$, that is, the last segment to load is the last entry in the segment table described by this HUNK_HEADER. For overlayed files, the number may be smaller, i.e. not all segments may be populated initially and loading may continue later on when executing the binary.

9.1.2 HUNK_CODE

This hunk should contain executable machine code and constant data. As executables are started from the first byte of the first segment, the first hunk of an executable should be a `HUNK_CODE`, and it should start with a valid opcode.

Compilers use typically this hunk to represent the `text` segment, i.e. compiled code and constant data. The structure of this hunk is as follows:

Table 23: Hunk Code Syntax

Size	Code	Syntax
	`HUNK_CODE [0x3e9]`	A hunk describing a segment of code and constant data
32	$l \leq m_s[i]$	Size of the payload
1×32	`Code`	l long words of payload

Note that the size of the payload loaded from the file may be less than the size of the allocated segment as defined in `HUNK_HEADER`. In such a case, all bytes of the segment not included in the `HUNK_CODE` are zero-initialized. AmigaDOS versions earlier than v37 skipped this initialization. Due to a bug in the loader in later versions, the initialization is also skipped of the hunk length l is 0.

9.1.3 HUNK_DATA

This hunk should contain variable data, and it should not contain executable code. Compilers typically use this hunk to represent initialized data.

The structure of this hunk is otherwise identical to `HUNK_CODE`:

Table 24: Hunk Data Syntax

Size	Code	Syntax
	`HUNK_CODE [0x3ea]`	A hunk describing a segment of data
32	$l \leq m_s[i]$	Size of the payload
1×32	`Code`	l long words of payload

Similar to `HUNK_CODE`, the size of the payload defined by this hunk may be less than the size of the segment allocated by `HUNK_HEADER`. Excess bytes are zero-initialized in all AmigaDOS releases from v37 onwards. Due to a bug in the loader in later versions, the initialization is also skipped of the hunk length l is 0.

9.1.4 HUNK_BSS

This hunk contains zero-initialized data; it does not define actual payload.

The structure of this hunk is as follows:

Table 25: Hunk BSS Syntax

Size	Code	Syntax
	`HUNK_CODE [0x3eb]`	A hunk describing zero-initialized data
32	$l \leq m_s[i]$	Size of the segment in long-words

Note that this hunk does not contain any payload; the segment allocated from this hunk is always zero-initialized.

Due to a defect in AmigaDOS prior release v37, the BSS segment will not be completely initialized to zero if the segment size is larger than 256K, i.e. if $l > 2^{16}$. Also, these releases do not initialize long words beyond the l^{th} long-word to zero, i.e. the excess bytes included if $l < m_s[i]$.

9.1.5 HUNK_RELOC32

This hunk contains relocation information for the previously loaded segment; that is, it corrects addresses within this segment by adding the absolute address of this or other segments to long words at indicated offsets of the previous segments.

The structure of this hunk is as follows:

Table 26: Hunk Reloc32 Syntax

Size	Code	Syntax
	`HUNK_RELOC32 [0x3ec]`	A hunk containing relocation information
	`do {`	Loop over relocation entries
32	c	Number of relocation entries
	`if (`c `== 0) break;`	Terminate the hunk if the count is zero
32	$j \in [0, t_{\texttt{size}} - 1]$	Read the hunk to which the relocation is relative to
	`do {`	Loop over the relocation entries
32	$r_o \in [0, m_s[i] \times 4 - 4]$	Relocation offset into this hunk as byte address
	`(UBYTE **)(`$m_a[i]$ `+` r_o`) +=` $m_a[j]$	Fixup this hunk by the start address of the selected hunk
	`} while(--c);`	until all entries are used
	`} while(true);`	until a zero-count is read.

That is, the hunk consists first of a counter that indicates the number of relocation entries, followed by the hunk index relative to which an address should be relocated. Then relocation entries follow; each long-word defines an offset into the previously loaded segment to relocate, that is, to fix up the address.

For AmigaDOS versions 37 and up (Kickstart 2.0 and later), the number of relocation entries c shall not be larger than 2^{16}. This is a known defect of the loader that has currently not yet fixed. If more relocation entries are needed, they shall be split into multiple chunks.

9.1.6 HUNK_RELOC32SHORT

This hunk contains relocation information for the previously loaded segment, and is almost similar to `HUNK_RELOC32`, except that hunk indices, counts and offsets are only 16 bits in size. To ensure that all hunks start at long-word boundaries, the hunk contains an optional padding field at its end to align the next hunk appropriately.

The structure of this hunk is as follows:

Table 27: Hunk Reloc32Short Syntax

Size	Code	Syntax
	`HUNK_RELOC32SHORT [0x3fc]`	A hunk containing relocation information
	p=1	Padding count
	`do {`	Loop over relocation entries
16	c	Number of relocation entries
	`if (`c `== 0) break;`	Terminate the hunk if the count is zero
16	$j \in [0, t_{\texttt{size}} - 1]$	Read the hunk to which the relocation is relative to
	p `+=` c	Update padding count
	`do {`	Loop over the relocation entries
16	$r_o \in [0, m_s[i] \times 4 - 4]$	Relocation offset into this hunk as byte address

	(UBYTE **)($m_a[i]$ + r_o) += $m_a[j]$	Fixup this hunk by the start address of the selected hunk
	} while(--c);	until all entries are used
	} while(true);	until a zero-count is read.
	if (p & 1) {	check whether padding is required.
16		dummy for long-word alignment
	}	

Due to an oversight, some versions of AmigaDOS do not understand the hunk type `0x3fc` properly and use instead `0x3f7`. This alternative (but incorrect) hunk type for the short version of the relocation hunk is still supported currently.

9.1.7 HUNK_RELRELOC32

This hunk contains relocation information for 32-bit relative displacements the `68020` and later processors offer. Its purpose is to adjust the offsets of a 32-bit wide PC-relative branches between segments.

The structure of this hunk is as follows:

Table 28: Hunk RelReloc32 Syntax

Size	Code	Syntax
	HUNK_RELRELOC32 [0x3fd]	A hunk containing relocation information
	do {	Loop over relocation entries
32	c	Number of relocation entries
	if (c == 0) break;	Terminate the hunk if the count is zero
32	$j \in [0, t_{\texttt{Size}} - 1]$	Read the hunk to which the relocation is relative to
	do {	Loop over the relocation entries
32	$r_o \in [0, m_s[i] \times 4 - 4]$	Relocation offset into this hunk as byte address
	(UBYTE **)($m_a[i]$ + r_o) += $m_a[j] - m_a[i] - r_o$	Fixup this hunk by the start address of the selected hunk
	} while(--c);	until all entries are used
	} while(true);	until a zero-count is read.

For AmigaDOS versions 37 and up (Kickstart 2.0 and later), the number of relocation entries c shall not be larger than 2^{16}. This is a known defect of the loader that has currently not yet fixed. If more relocation entries are needed, they shall be split into multiple chunks.

Due to another defect, the r_o entry, i.e. the relocation offset, is only read as a 16-bit displacement, which limits the usefulness of this hunk. It is therefore recommended not to depend on this hunk type at all and avoid 32-bit wide branches between segments. Luckely, the support for this hunk type is very limited.

9.1.8 HUNK_NAME

This hunk defines a name for the current segment. The AmigaDOS loader completely ignores this name, and it does not serve a particular purpose for the executable file format. However, linkers that bind object files together use the name to decide which segments to merge together to a single segment.

The structure of this hunk is as follows:

Table 29: Hunk Name Syntax

Size	Code	Syntax
	`HUNK_NAME [0x3e8]`	A hunk assigning a name to the current segment
32	`l`	Size of the name in long-words
$32 \times l$	h_n	Hunk name

The size of the name is not given in characters, but in 32-bit units. The name is possibly zero-padded to the next 32-bit boundary to fill an integer number of long-words. If the name fills an entire number of long-words already, it is *not* zero-terminated.

While the specification does not define a maximum size of the name, the AmigaDOS loader fails on names longer than 124 character, i.e. 31 long-words.

9.1.9 HUNK_SYMBOL

This hunk defines symbol names and corresponding symbol offsets or values within the currently loaded segment. Again, the AmigaDOS loader ignores this hunk, but the linker uses it to resolve symbols with external linkage to bind multiple object files together. If the symbol information is retained in the executable file, it may be used for debugging purposes.

The synax of this hunk reads as follows:

Table 30: Hunk Symbol Syntax

Size	Code	Syntax
	`HUNK_SYMBOL [0x3f0]`	A hunk assigning symbols to positions within a segment
	`do {`	Repeat . . .
8	s_t	Symbol type
24	s_l	Symbol length in long-words
	`if (`s_l `== 0) break`	Terminate the hunk
$32 \times s_l$	s_n	Symbol name, potentially zero-padded
32	s_v	Symbol value
	`} while(true)`	until zero-sized symbol

The length of the symbol name is encoded in long-words, not in characters. If it does not fill an integer number of long-words, it is zero-padded; the name is not zero-terminated if it does fill an integer number of long-words, though. The AmigaDOS loader is currently limiting the maximum size of the symbol name to 124 characters, i.e. $s_i < 32$.

The symbol type s_t defines the nature of the symbol. The symbol types are defined in `dos/doshunks.h` and shared with the `HUNK_EXT` hunk; the latter hunk type shall not appear in an executable file, but may only appear in an object file, see section 9.5.7.

The symbol type can be roughly classified into two classes: If bit 7 of the type is clear, a symbol is *defined* that may be referenced by another object file. If bit 7 is set, the symbol is *referenced* and requires resolution by a symbol definition with bit 7 cleared upon linking. Executable files, and thus symbols within `HUNK_SYMBOL`, may only contain symbol definitions as references had been resolved by the linker before.

The following table contains the symbol types for definitions and those may therefore may appear in both `HUNK_SYMBOL` as part of executables and `HUNK_EXT` as part of object files; actually, `HUNK_SYMBOL` will typically only include the first type of entry, i.e. `EXT_SYMB`:

Table 31: Symbol types in HUNK_SYMBOL and HUNK_EXT

EXT_SYMB [0x00]	Definition of a symbol, $s_v + m_a[i]$ is the address of the symbol
EXT_DEF [0x01]	Relocation definition, $s_v + m_a[i]$ is the address of the symbol. References to this symbol are converted into a relocation information to the offset s_v in hunk i.
EXT_ABS [0x02]	Absolute value, s_v is the value of the symbol which is substituted into the executable by the linker. No relocation information is created, the absolute value is just substituted.
EXT_RES [0x03]	Not longer supported as it is part of the obsolete dynamic library runtime binding interface, see [10] for more details.

Additional symbol types representing references used within HUNK_EXT are documented in section 9.5.7.

9.1.10 HUNK_DEBUG

This hunk contains debug information such as function names and line number information. Generally, the contents of this hunk is compiler or assembler specific, and the AmigaDOS loader does not interpret the contents of this hunk at all, it is just skipped over.

However, the debug information emitted by the SAS/C compiler for the "line-debug" option is also shared by other development tools such as the DevPac assembler and will be documented here. In this format, the debug hunk contains for each line of the source file an offset into the hunk to the code that was compiled from this line.

The syntax of this hunk is as follows:

Table 32: Hunk Debug Syntax

Size	Code	Syntax
	HUNK_DEBUG [0x3f1]	Hunk including debug information
32	l > 3	Size of the hunk in long-words
Compiler- and configuration specific data for line-debug data:		
32	0	This field shall be zero
32	'LINE'	These four bytes shall contain the ASCII characters 'L','I','N','E' identifying the type of the debug information
32	l_n	Size of the source file name in long-words
32 $\times l_n$	n_f	source file name that compiled to the current segment in l_n long-words
	$l- = 3 + l_n$	Remove long-words read so far
	while(l > 0) {	Repeat for all entries
8		Dummy byte
24	l_l	Line number within the source file
8		Dummy byte
24	l_v	Offset into the source file. The source file at line l_l is compiled or assembled to the code at at address $m_a[i]+l_v$ and following.
	$l- = 2$	Remove the read data
	};	Loop over the hunk.

The file name n_f is encoded in l_n long-words, and potentially padded with 0-bytes to fill an integer number of long-words. If it already is an integer number of long-words sized, it is *not* zero-terminated.

9.1.11 HUNK_END

This hunk terminates the current segment and advances to the next segment, if any. It does not contain any data.

Table 33: Hunk End Syntax

Size	Code	Syntax
	HUNK_END [0x3f2]	Terminate a segment

9.2 The AmigaDOS Loader

The *dos.library* provides service functions for loading and releasing binary executables in the *Hunk* format introduced in section 9. The functions discussed in this section load such binaries into memory, constructing a segment list from the hunks found in the files, or release such files. Overlay files are discussed separately in section 9.3 due to their additional complexity.

A segment list is a linearly linked list as defined in section 9, i.e. the first four bytes of every segment form a *BPTR* to the following segment of the loaded binary, or ZERO for the last segment.

The seglist returned from the loader functions may be, for example, passed into CreateNewProc() as argument to the NP_Seglist tag for starting a new process.

9.2.1 Loading an Executable

The LoadSeg() function loads an executable binary in the Hunk format and returns a *BPTR* to the first segment:

```
seglist = LoadSeg( name )
D0                 D1

BPTR LoadSeg(STRPTR)
```

This function loads the binary executable named name and returns a *BPTR* to its first segment in case of success, or ZERO in case of failure. The name is passed into the Open() function and follows the conventions of this function for locating the file.

The segment list shall be removed from memory via UnLoadSeg().

This function sets IoErr() to an error code in case of failure, or 0 in case of success.

9.2.2 Loading an Executable with Additional Parameters

The NewLoadSeg() function loads an executable providing additional data for loading.

```
seglist = NewLoadSeg(file, tags)
D0                   D1    D2

BPTR NewLoadSeg(STRPTR, struct TagItem *)

seglist = NewLoadSegTagList(file, tags)
D0                           D1    D2

BPTR NewLoadSegTagList(STRPTR, struct TagItem *)

seglist = NewLoadSegTags(file, ...)
```

```
BPTR NewLoadSegTags(STRPTR, ...)
```

This function loads a binary executable from `file` and returns a *BPTR* to its first segment, similar to `LoadSeg()`.

Additional parameters may be provided in the form of a TagList, passed in as `tags`. The first two functions are identical and differ only by their naming convention; the last function prototype also refers to the same entry within the *dos.library*, though uses a different calling convention where the second and all following arguments form the TagList itself. This TagList is build on the stack, and the pointer to this stack-based TagList is passed in.

While this function looks quite useful, AmigaDOS does currently not define any tags for this function, and thus no additional functionality over `LoadSeg()` is provided.

The segment list returned by this function shall be removed from memory via `UnLoadSeg()`, a specialized unloader function is not required for this call.

9.2.3 Loading an Executable through Call-Back Functions

The `InternalLoadSeg()` function loads a binary executable, retrieving data and memory through call-back functions. While `LoadSeg()` always goes through the *dos.library* and the *exec.library* for reading data and allocating memory, this function instead calls through user-provided functions.

```
seglist = InternalLoadSeg(fh,table,funcs)
D0                        D0  A0   A1

BPTR InternalLoadSeg(BPTR,BPTR,struct LoadSegFuncs *)
```

This function loads a binary executable in the hunk format from an opaque file handle `fh` through functions in the `funcs`. The `table` argument shall be `ZERO` when loading regular binaries or the root node of an overlay file, and shall be a *BPTR* to the array containing pointers to all segments when loading a non-root overlay node, see section 9.3.

The `LSFuncs` structure contains function pointers through which this function loads data or retrieves memory. It looks as follows:

```
struct LoadSegFuncs {
    LONG __asm ReadFunc(register __d1 BPTR fh,
                        register __a0 APTR buffer,
                        register __d0 ULONG size,
                        register __a6 struct DosLibrary *DOSBase);
    APTR __asm AllocMem(register __d0 ULONG size,
                        register __d1 ULONG flags,
                        register __a6 struct ExecBase *SysBase);
    void __asm FreeMem (register __a1 APTR mem,
                        register __d0 ULONG size,
                        register __a6 struct ExecBase *SysBase);
}
```

The `ReadFunc()` function retrieves `d0` bytes from an opaque file handle passed into register `d1` and places the read bytes into the buffer pointed to by register `a0`, it shall return the number of bytes read in register `d0`, or a negative value in case of error. Note that the file handle `d1` need not to be a file handle as returned by the `Open()` function, it is only a copy of the `fh` argument provided to `InternalLoadSeg()`. Register `a6` is loaded by a pointer to the *dos.library*.

The AllocMem() function allocates d0 bytes of memory, using requirement flags from exec/memory.h such as MEMF_CHIP to require chip memory or MEMF_FAST for fast memory. This function shall return a pointer to the allocated memory in register d0, or NULL in case of failure. Register a6 is loaded with a pointer to the *exec.library*.

The FreeMem() function releases a block of d0 bytes pointed to by a0. Register a6 is loaded with a pointer to the *exec.library*.

The purpose of this function is to load a segment or a binary without having access to a file or a *file system*; for example, this function could load binaries from ROM-space, or from the Rigit Disk Block of a boot partition. In particular, the fh argument does not need to be a regular *file handle*; it is rather an opaque value identifying the source. The InternalLoadSeg() function does not interpret this argument, but rather passes it into funcs->ReadFunc() in register d1.

When allocating memory, the InternalLoadSeg() function follows the conventions of the AllocVec() and FreeVec() functions and stores the number of allocated bytes in the first four bytes of the allocated memory block. In specific, the memory allocator and memory releaser functions provided in the LoadSegFuncs structure *do not need* to store the memory sizes, and the exec AllocMem() and FreeMem() functions satisfy the interfaces for InternalLoadSeg() function already.

This function does not set IoErr() consistently, unless the functions within LoadSegFuncs do. Callers should also call SetIoErr(0) upfront this function to identify all errors.

9.2.4 Unloading a Binary

The UnLoadSeg() function releases a linked list of segments as returned by LoadSeg() or NewLoadSeg().

```
success = UnLoadSeg( seglist )
D0                   D1

BOOL UnLoadSeg(BPTR)
```

This function releases all segments chained together by LoadSeg() and NewLoadSeg() and returns their memory back into the system pool. This function *also* accepts overlayed segments, see section 9.3, and releases additional resources accquired for them.

Segment lists loaded through InternalLoadSeg() require in general a more generic unloader. They shall be be released through InternalUnLoadSeg() instead, see 9.2.5.

This function returns a non-zero result in case of success, or 0 in case of error. Currently, the only source of error is passing in ZERO as segment list, all other cases will indicate success. In particular, this function does not attempt to check return codes of the function calls required to release resources assoicated to overlayed files.

9.2.5 UnLoading a Binary through Call-Back Functions

The InternalUnLoadSeg() function releases a segment list loaded through InternalLoadSeg().

```
success = InternalUnLoadSeg(seglist,FreeFunc)
D0                          D1       A1

BOOL InternalUnLoadSeg(BPTR,
                       void __asm (*)(register __a1 APTR,
                                      register __d0 ULONG,
                                      register __a6 struct ExecBase
                                                    *SysBase))
```

This function releases a segment list created by `InternalLoadSeg()` passed in as `seglist`. To release memory, it uses a function pointed to by `a1`. This function expects the memory block to release in register `a1` and its size in register `d0`. Additionally, register `a6` will be populated by a function to the *exec.library*.

This function pointer should be identical to the `FreeMem` function pointer in the `LoadSegFuncs` structure provided to `InternalAllocMem()`, or at least shall be able to release memory allocated by the `AllocMem` function pointer in this structure. Note that the `InternalLoadSeg()` stores the sizes of the allocated memory blocks itself and that `FreeFunc` does not need to retrieve them.

This function is also able to release overlayed binaries, but then closes the file stored in the root node of the overlay tree (see section 9.3) through the `Close()` function of the *dos.library*. It therefore can only release overlayed files that were loaded from regular *file handles* obtained through `Open()`.

This function returns a non-negative result code in case of success, or 0 in case of failure. Currently, the only cause of failure is to pass in a `ZERO` segment list, the function does not check of the result code of `Close()` on the file handle of overlayed files. It therefore neither sets `IoErr()` consistently in case of failure.

9.3 Overlays

While regular binary executables are first brought to memory in entity and then brought to execution, overlayed binaries only keep a fraction of the executable code in memory and then load additional code parts as required, potentially releasing other currently unused code parts and thus making more memory available.

Overlays are an extension of the AmigaDOS hunk format that splits the executable into a root node that is loaded initially and stays resident for the lifetime of the program, and one or multiple extension or overlay nodes that are loaded and unloaded on demand. Locating the overlay nodes, loading them to memory and releasing unused nodes is performed by the *overlay manager*, a short piece of program.

AmigaDOS does not provide a ROM-resident overlay manager itself, i.e. the *dos.library* does not provide an overlay manager itself, though it provides services overlay managers may use. Instead, the overlay manager is part of the root node of an overlayed binary, and thus overlay management is fully under control of the application.

However, the Amiga linker *ALink*, the Software Distillery linker *BLink* and the SAS/C linker *SLink* include a standard overlay manager, and this manager and its properties are discussed in greater detail in this section.

9.3.1 The Overlay File Format

A binary file making use of overlays consists of several nodes, one root node and several overlayed nodes. Nodes contain multiple segments, defined through `HUNK_CODE`, `HUNK_DATA` or `HUNK_BSS` as in regular (non-overlayed) binary files.

Each node, the root node and all overlayed nodes start with a `HUNK_HEADER` identifying which segments are contained in the node. The root node is terminated by a `HUNK_OVERLAY` on which loading stops; this hunk contains additonal data for the purpose of the overlay manager, and therefore the data within this hunk depends on the overlay manager.

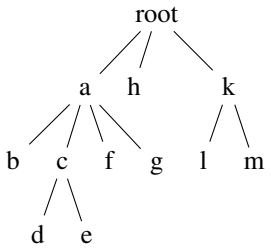
Every other overlay node terminates with a `HUNK_BREAK`, and loading stops there as well. This hunk does not contain any data. The overall structure of an overlayed binary therefore looks as follows:

Table 34: Overlay File Format

Hunk Type	Description
HUNK_HEADER	Defines segments for the root node
HUNK_CODE	Contains the overlay manager and other resident code
...	Other hunks, such as relocation information
HUNK_END	Terminates the previous segment
HUNK_OVERLAY	Metadata for the overlay manager, see 9.3.3
do {	Repeats over all overlay nodes
HUNK_HEADER	Defines the segments in this overlay node
HUNK_CODE or HUNK_DATA	First segment of the overlay node
...	Other hunks of this overlay node
HUNK_END	Terminates the last segment
HUNK_BREAK	Terminates the first overlay node, see 9.3.4
} while(!end of file);	This pattern repeats until end of file

9.3.2 The Hierarchical Overlay Manager

The overlay manager that comes with the standard Amiga linkers *ALink*, *BLink* and *SLink* structures overlay nodes into a tree such as the following:



Only those nodes that form a path from the root to one of the nodes of the tree can be in memory at a time. Thus, for the above example, the root node and nodes a, c and e can be in memory simultaneously, or the root node, and nodes k and m can be loaded at the same time, but not the nodes a, g and h because they do not form a path from the root to one of the nodes.

Thus, in the above example, if nodes a and f are in memory, and node l is required, the nodes a and f will be removed from memory, and nodes k and l are loaded. Even though k is not explicitly requested, it needs to be loaded as it is the parent of l.

Every node in the overlay tree is identified by two numbers: The depth of the node, which identifies the level within the tree where a node is located. The root node is at level 0, the nodes a, h and k forms level 1 in the above example, nodes b, c, f, g and l and m form level 2, and nodes d and e are level 3.

The second number is the ordinate number of a node. The ordinate enumerates nodes from left to right within a level, and it starts from 1 in the standard overlay manager. In the above example, a is at ordinate 1, h at ordinate 2, and k at ordinate 3. At level 2, node b has ordinate 1, node c ordinate 2 and so on.

9.3.3 HUNK_OVERLAY

This hunk terminates the loading process and indicates the end of the main (first) segments. The HUNK_OVERLAY contains meta-data — the overlay table — for the overlay manager. This table contains information where symbols within the overlayed segments are located. Section **??** provides more information on overlays. The format of the data within this hunk depends on the overlay manager which shall be included in the first segment of the executable itself as AmigaDOS does not contain a resident overlay manager.

The standard AmigaDOS linkers, `ALink` and `BLink` both include an overlay manager. Each entry in its overlay table describes a symbol that is located in one of the overlay nodes. The format of `HUNK_OVERLAY` reads as follows:

Table 35: Hunk Overlay Syntax

Size	Code	Syntax
	`HUNK_OVERLAY [0x3f5]`	Overlay table definition
32	l	Size of the overlay table, it is $l + 1$ long-words large.
Format for the standard overlay manager, $l + 1$ long-words.		
32	o_d	Number of levels in the overlay tree, including the root node
	`for(`i `= 1;`i `<` o_d`;i++) {`	For all nodes, excluding the root node
32	0	Currently loaded ordinate, shall be zero
	}	That is, $o_d - 1$ zeros
	$l-= o_d$	Count removed long-words
	$s = 0$	Start with symbol 0
	`while(`$l \geq 0$`) {`	Repeat over the overlay table
32	$o_p[s]$	Absolute file offset of the `HUNK_HEADER` of the overlay node containing the symbol.
64		Two reserved long-words.
32	$o_l[s]$	Level of the overlay node containing the symbol, the root level containing the overlay manager is level 0.
32	$o_n[s]$	Ordinate of the overlay node, enumerating overlay nodes of the same depth.
32	$o_h[s]$	Hunk index of the first hunk within the overlay node.
32	$o_s[s]$	Hunk index of the hunk containing the symbol described by this entry in the overlay node.
32	$o_o[s]$	Symbol offset within hunk o_s
	l `-=8`	Remove 8 long words.
	$s++$	Advance to next symbol.
	}	End of loop over table

Note that the overlay table is $l + 1$ and not l long-words large, i.e. a table only defining a single symbol would be indicated by a value of $l = 7$. While the payload data of `HUNK_OVERLAY` is always $l + 1$ long-words large, with l indicated in the first long-word of the hunk, the format of the subsequent data is specific to the overlay manager used.

Irrespective of the overlay manager used, the AmigaDOS loader injects overlay-specific data into the first segment loaded from disk, that is, into the root-node. The data placed there is also required to release all resources associated to overlays and is expected there by `UnLoadSeg()` and `InternalUnLoadSeg()`.

The first bytes of the root node shall therefore form the following structure:

```
struct OverlayHeader {
    UWORD          oh_Jump[2];    /* Forms a branch to the startup code */
    LONG           oh_Magic;      /* Shall be 0x0000abcd                */
    BPTR           oh_FileHandle;/* Filled by the loader with the fh    */
    struct OVTab *oh_OVTab;       /* Overlay table from HUNK_OVERLAY     */
```

```
    BPTR          oh_Segments;  /* Array of segment BPTRs             */
    BPTR          oh_GV;        /* standard Global Vector             */
};
```

As said earlier, this structure is expected to be present at the start of the first hunk of the root node. The members `oh_FileHandle` to `oh_GV` are filled in by the AmigaDOS loader, i.e. `LoadSeg()` and related functions, but `oh_Jump` and `oh_Magic` shall be part of the segment itself.

`oh_Jump` form valid `68K` opcodes, and shall contain a jump or branch branch around this structure. This is because loaded binaries are executed from the first byte of the first segment loaded. Otherwise, the CPU would run into the data of the structure which likely forms invalid or illegal opcodes. The AmigaDOS Loader itself does not interpret the values here, just expects them to be present.

`oh_Magic` shall contain the "magic" long-word `0xabcd`. This value is neither filled or interpreted by the loader, but nevertheless shall be present. It is, however, checked by `UnLoadSeg()` and used there as an identifier for the `OverlayHeader` structure. If this identifier is not present, `UnLoadSeg()` will not be able to release resources associated to overlays.

`oh_FileHandle` will be filled by the AmigaDOS loader with a *BPTR* to the *FileHandle* from which the root node has been loaded, or with the first argument of `InternalLoadSeg()`. This handle is used by the overload manager to load all subsequent overlay nodes. Also, `UnLoadSeg()` and related functions call `Close()` on the handle stored here as the file needs to stay open for the life time of the loaded program.

`oh_OVTab` is filled by the AmigaDOS loader to a pointer to the payload data of `HUNK_OVERLAY`. The standard overlay manager stores here for every externally referenced symbol in an overlay node a structure that records for each tree level the ordinate of the currently loaded overlay node, and for all externally referenced symbols the position of the symbol within the overlay tree:

```
struct OVTab {
    ULONG ot_TreeDepth;   /* Depth of the tree, including the root    */
    ULONG ot_LoadedOrd[]; /* The loaded ordinate, indexed by level-1  */
}
struct SymTab {
    ULONG ot_FilePosition;/* File position of HUNK_HEADER of the node */
    ULONG ot_Reserved[2]; /* Not in use                               */
    ULONG ot_Level;       /* Level of the overlay node                */
    ULONG ot_Ordinate;    /* Ordinate of the overlay node             */
    ULONG ot_FirstHunk;   /* First segment of the overlay node        */
    ULONG ot_SymbolHunk;  /* Segment containing the referenced symbol */
    ULONG ot_SymbolOffset;/* Offset of the segment within the segment */
} []                      /* Repeats for each symbol */
```

That is, the overlay table starts with the tree depth o_d and an array of $o_d - 1$ elements where each element stores the ordinate of the currently loaded overlay node. If an entry in this array is 0, no overlay node at this tree level is loaded, otherwise it is the 1-based ordinate of the node.

The ordinate table is followed by the symbol table. The purpose of this structure is that it allows the overlay manager on a reference to such an external symbol to find and load the overlay node containing the symbol, and then resolve references to it. How exactly it does so is explained in more detail in section **??**. The elements of this structure are already briefly introduced in table 35.

`oh_Segments` is filled by the AmigaDOS loader to a *BPTR* to the segment table of the loaded binary. The size of this table is taken from t_{size} in the `HUNK_HEADER` of the root node, see table 22. Each element in this array contains a *BPTR* to a segment of the loaded binary, and it is indexed by the segment number, counting from 0 for the first segment of the root node.

When parsing a `HUNK_HEADER`, the array entries t_{num} to t_{max} will be populated with the *BPTR*s to the segments allocated of the node described by this hunk, and when unloading an overlay node, the corresponding segments will be unlinked, released and then cleared out.

oh_GV is, finally, filled with the *Global Vector* of the *dos.library* containing all regular functions in the library, as required by BCPL code. Overlay managers implemented in C or assembler will not make use of it and instead call vectors of the *dos.library* through the *dos.library* base address loaded in register a6.

9.3.4 HUNK_BREAK

This hunk terminates the loading process and indicates the end of an overlay node. The hunk itself does not contain any data.

Table 36: Hunk Break Syntax

Size	Code	Syntax
	HUNK_BREAK [0x3f6]	Terminate a segment

9.3.5 Loading an Overload Node

The LoadSeg() function is not only able to load the root segments of an overlayed binary, it can also be used for loading an overlayed node and all segments within it. For that, the file pointer shall first be placed with Seek() to the file offset of the HUNK_HEADER of the overlayed node. This file offset may, for the standard hierarchical overlay manager, be taken from the ot_FilePosition of the overlay table.

```
seglist = LoadSeg( name ,table, fh)
D0                 D1    D2     D3

BPTR LoadSeg(STRPTR ,BPTR, BPTR)
```

For overlayed node loading, the first argument name shall be NULL, which is used as an indicator to this function to interpret two additional (usually hidden) arguments.

table is a *BPTR* to the segment table, and may be taken from oh_Segments. It contains *BPTRs* to all allocated segments, see section 9.3.3.

fh is a *BPTR* to the *FileHandle* from which the overlay node is to be loaded. This handle may be taken from oh_FileHandle, see section 9.3.3.

While this function allocates and loads the segments in the overlayed node, it does not attempt to release already allocated segments populating the same entries in the segment table; it is instead up to the overlay manager to clean up the segment table upfront, see 9.3.7. The information which segments will be populated by an overlay node may be taken from the ot_FirstHunk member of the overlay table. Due to the tree structure imposed by the hierarhical overlay manager, it has to release all segments from ot_FirstHunk onwards up to the end of the table, unlink the segments contained therein, and then load another overlay node through LoadSeg().

Note that this function populates the same offset in the *dos.library* as the regular LoadSeg() function; the function distinguishes loading regular binaries through a file name from loading overlay nodes by the first argument.

As the regular LoadSeg() call, this function returns the *BPTR* to the first segment loaded on success, links all loaded segments together, populates the segment table, and then sets IoErr() to 0. On error, it returns ZERO and installs an error code in IoErr().

9.3.6 Loading an Overlay Node through Call-Back Functions

The InternalLoadSeg() function can also load an overlay node.

```
seglist = InternalLoadSeg(fh,table,funcs)
D0                        D0  A0   A1

BPTR InternalLoadSeg(BPTR,BPTR,struct LoadSegFuncs *)
```

The `fh` argument is an opaque file handle that is suitable for the `ReadFunc()` provided by the `funcs` structure. The corresponding file pointer shall first be placed to the file offset of the `HUNK_HEADER` of the overlayed node, e.g. by a functionality similar to `Seek()` for regular *FileHandles*. This file offset may, for the standard hierarchical overlay manager, be taken from the `ot_FilePosition` within the overlay table.

The `table` shall be the *BPTR* to the segment table; this may be taken from `oh_Segments`. This argument determines whether a regular binary load is requested, or an overlay node is to be loaded. In the latter case, this argument is non-`ZERO`.

Like `LoadSeg()`, this function does not release segments in populated entries in the segment list, it is up to the overlay manager to unload these segments. The information which entries of the segment table will be populated by an overlay node may be taken from the `ot_FirstHunk` member of the overlay table, see also 9.3.5.

The `funcs` argument points to a `LoadSegFuncs` structure as defined in section 9.2.3 and contains functions for reading data and allocating and releasing memory.

This function does not set `IoErr()` consistently, unless the functions in the `LoadSegFuncs` structure do. The function returns the segment of the first segment of the overlay node on success, or `ZERO` on error.

9.3.7 Unloading Overlay Nodes

Unloading overlay nodes (and *not* the root node) of an overlayed binary requires some manipulation of the segment table as the *dos.library* does not provide a function for such operation. This algorithm is part of the overlay manager, but its implementation within the standard hierarchical overlay manager documented here for completeness. Other custom overlay managers perform potentially different algorithms.

First, it finds the previous segment upfront the segment to be unloaded, and cleans there the `NextSegment` pointer to unlink all following segments. Then these following segments are released through `FreeVec()` or whatever memory release function is appropriate.

The following sample code releases the overlay node starting at segment $i > 0$ from a segment table of an overlay header:

```
void UnloadOverlayNode(struct OverlayHeader *oh,ULONG i)
{
  BPTR *segtbl  = (BPTR *)BADDR(oh->oh_Segments);
  BPTR *segment = (BPTR *)BADDR(segtbl[i - 1]);
  BPTR  next;

  /* Release the linkage from the last loaded to
  ** the first segment to release */
  *segment = NULL;

  do {
    /* Get the segment to release */
    if (segment = (BPTR *)BADDR(segtbl[i++])) {
      next      = *segment;
      FreeVec(segment);
    } else break;
```

```
    /* Repeat until the last segment */
  } while(next);
}
```

Note that a previous segment always exists because the root node populates at least entry 0 of the segment table. The above loop makes use of the fact that the first long-word of a segment is a *BPTR* to the next segment, and this linkage is `ZERO` for the final node.

If a custom memory allocator has been used for loading overlay nodes through `InternalLoadSeg()`, the `FreeVec()` in the above function is replaced by the corresponding memory release function.

9.3.8 Unloading Overlay Binaries

To unload the root node, and thus unload the entire program including all overlay nodes, `UnLoadSeg()` on the first segment of the root node is sufficient if neither custom I/O nor a custom memory allocator has been used to load the binary, independent on which overlay manager has been used. `UnLoadSeg()` will detect the overlay manager from the magic value in `oh_Magic` and will then not only release the segments, but also close the overlay file handle and release the segment table.

If `InternalLoadSeg()` has been used for loading the root node through custom I/O functions or with a custom memory allocator, `InternalUnLoadSeg()` shall be used instead to release the root node. Unfortunately, it *always* uses `Close()` on `oh_FileHandle`, even if `oh_FileHandle` does not correspond to a *FileHandle* as returned by `Open()`, e.g. because `ReadFunc()` upon loading the overlay program pointed to a custom I/O function. The best strategy in this case is probably to close `oh_FileHandle` manually upfront with whatever method is appropriate, then zero it out manually and then finally call into `InternalUnLoadSeg()` to perform all the necessary cleanup steps. This strategy works because `Close()` on a `ZERO` file handle performs no operation and is legit.

9.3.9 Internal Working of the Overlay Manager

Several versions of the hierarchical overlay manager exist. The version described here stems from the SAS/C `SLink` utility and is designed for the *registerized parameters* configuration within which some function arguments are passed in CPU registers. Earlier versions require stack-based parameter passing.

When binding objects together to an overlay binary together, the linker checks whether a reference to a symbol crosses a boundary of overlay nodes. References that go to a parent node or the node itself can be resolved by the linker by creating a relocation entry in a `HUNK_RELOC32` hunk as it can assume that the corresponding segment is already loaded.

References to symbols within child nodes receive each a unique integer identifier, and an entry in the overlay table in `HUNK_OVERLAY` at the index given by the identifier. The actual call to a function in a child node is then replaced to call into a trampoline function that looks as follows:

```
@symX:
    jsr @ovlyMgr
    dc.w symbX
```

where `@ovlyMgr` is the entry point of the overlay manager and `symbX` is the identifier of the referenced symbol. The overlay manager reads the return PC which points to the identifer, and from the identifier finds the entry in the symbol table.

The symbol table contains both the ordinate and the level of the symbol with which the overlay manager is able to check whether the node containing the symbol is currently loaded. If this is not the case, it needs to unload the currently loaded node at this level and all its children, and then progresses to loading the required node from the file offset in the symbol table, and then progresses to updating the overlay table.

If the overlay node containing the symbol is already loaded, or just has been loaded, the symbol address is computed from the offset in the symbol table and the address of the segment containing the symbol from the segment table, and injected into the return address that contained a pointer to the symbol identifier. Thus, when returning from the overlay manager, the code will continue to execute from the target symbol. Other versions of the overlay manager use a trampoline function that loads register `d0` with the symbol identifier and thus require stack-based function calls.

Regardless of the version of the overlay manager, only symbols corresponding to function can be resolved as the overlay manager must be called to resolve a symbol. In particular, data cannot be referenced across overlay nodes — instead, an accessor function may be used that returns the object to be accessed.

The following code provides an overlay manager for register-based calls:

```
        xdef    @ovlyMgr
;***********************************************
;** Offsets in the overlay-table                **
;***********************************************
        rsreset
ot_FilePosition:        rs.l 1          ;File position
ot_reserved:            rs.l 2          ;for whatever
ot_OverlayLevel:        rs.l 1          ;Overlay-Level
ot_Ordinate:            rs.l 1          ;Overlay-Ordinate
ot_InitialHunk:         rs.l 1          ;Initial hunk for loading
ot_SymbolHunk:          rs.l 1          ;Hunk containing symbol
ot_SymbolOffset:        rs.l 1          ;Offset of symbol
ot_len:                 rs.b 0

;***********************************************
;** Other stuff                                 **
;***********************************************
MajikLibWord    =       23456

       section NTRYHUNK,CODE
;***********************************************
;** Manager starts here                         **
;***********************************************
Start:
        bra.w NextModule                ;Jump to the next segment...

;* This next word serves to identify the overlay
;* supervisor to 'unloader', i.e. UnLoadSeg()

        dc.l       $ABCD                ;Magic longword for UnLoadSeg

;* The next four LWs are filled by the loader (LoadSeg())
ol_FileHandle:  dc.l 0                  ;Overlay file handle (points to me)
ol_OverlayTab:  dc.l 0                  ;Overlay table as found in the overlay hunk
ol_HunkTable:   dc.l 0                  ;BPTR to Overlay hunk table
ol_GlobVec:     dc.l 0                  ;BPTR to global vector (what for ?)

                dc.l MajikLibWord       ;Majik library word as identifier
                dc.b 7,"Overlay"        ;Majik identifier
```

```
;* the following data is specific to this manger

ol_SysBase:     dc.l 0                  ;additional pointer
ol_DOSBase:     dc.l 0                  ;to libraries

                dc.b "THOR Overlay Mananger 1.0",0      ;another ID

@ovlyMgr:                               ;Entry-points
        saveregs d0-d3/a0-a4/a6         ;Saveback register

        moveq #0,d0
        move.l 10*4(a7),a0
        move.w (a0),d0                  ;get the overlay reference ID

        move.l ol_OverlayTab(pc),a3     ;get pointer to overlay table
        move.l a3,a4                    ;to a4
        add.l (a3),d0                   ;add length
        lsl.l #2,d0                     ;get offset
        add.l d0,a3                     ;address of overlay entry
        move.l ot_OverlayLevel(a3),d0   ;get overlay
        lsl.l #2,d0
        adda.l d0,a4
        move.l ot_Ordinate(a3),d0       ;get required ordinate level
        cmp.l (a4),d0                   ;compare with current ordinate level
        beq.s .gotsegment
                                        ;not correct level
                                        ;clear all other entries behind this
        move.l d0,(a4)+                 ;fill with new overlay entries

        moveq #0,d1
        do                              ;macros in action! ;-)
         tst.l (a4)                     ;terminate, if end of table found
         break.s eq
         move.l d1,(a4)+                ;clear this
        loop.s

        move.l ot_InitialHunk(a3),d0    ;first hunk number to load
        add.l ol_HunkTable(pc),d0       ;plus BPTR of hunk table
        lsl.l #2,d0                     ;address of entry in hunk table
        move.l d0,a4
        move.l -4(a4),d0                ;get previous hunk
        beq.s .noprevious
        lsl.l #2,d0
        move.l d0,a2
        move.l d1,(a2)                  ;unlink fields before loading

                                        ;now free all hunks
        move.l ol_SysBase(pc),a6

        do
         move.l (a4)+,d0                ;next hunk ?
```

```
         break.s eq
         lsl.l #2,d0
         move.l d0,a1                    ;->a1
         move.l -(a1),d0                 ;get length
         jsr FreeMem(a6)                 ;free this hunk
        loop.s                           ;and now the next
.retry:
        move.l ol_DOSBase(pc),a6
        move.l ol_FileHandle(pc),d1      ;get our stream
        move.l ot_FilePosition(a3),d2    ;get file position
        moveq #-1,d3                     ;relative to beginning of file
        jsr Seek(a6)                     ;seek to this position
        tst.l d0                         ;found something ?
        bmi.s .loaderror                 ;what to do on failure ?

                                         ;now call the loader
        move.l ol_HunkTable(pc),d2       ;hunk table
        moveq #0,d1                      ;no file (is overlay)
        move.l ol_FileHandle(pc),d3      ;filehandle
        jsr LoadSeg(a6)                  ;load this stuff
        tst.l d0                         ;found
        beq.s .loaderror
        move.l d0,(a2)                   ;add new chain

                                         ;found this stuff
.gotsegment:
        move.l ot_SymbolHunk(a3),d0      ;get hunk # containing symbol
        add.l ol_HunkTable(pc),d0
        lsl.l #2,d0                      ;get APTR to hunk
        move.l d0,a4
        move.l (a4),d0                   ;BPTR to hunk
        lsl.l #2,d0
        add.l ot_SymbolOffset(a3),d0     ;Offset

        move.l d0,10*4(a7)               ;Set RETURN-Address

        loadregs
        rts

;***********************************************
;** Go here if we find an error                **
;***********************************************
.loaderror:
        saveregs d7/a5
        move.l ol_SysBase(pc),a6
        move.l #$0700000C,d7
        move.l $114(a6),a5
        jsr Alert(a6)                    ;Post alert
        loadregs
        bra.s .retry                     ;Retry or die
```

```
.noprevious:
        move.l ol_SysBase(pc),a6
        move.l #$8700000C,d7            ;dead end !
        move.l $114(a6),a5
        jsr Alert(a6)                   ;Post alert
        bra.s .noprevious


;***********************************************
;** NextModule                                **
;** Open stuff absolutely necessary and       **
;** continue with main program code           **
;***********************************************
NextModule:
                                        ;why safe registers ?
        move.l a0,a2
        move.l d0,d2                    ;keep arguments (BCPL stuff is not kept)
        lea ol_SysBase(pc),a3
        move.l ExecBase,a6
        move.l a6,(a3)                  ;fill in Sysbase
        lea DOSName(pc),a1              ;get name of DOS
        moveq #33,d0                    ;at least 1.2 MUST be used (no support of antique s
        jsr OpenLibrary(a6)
        move.l d0,4(a3)                 ;Save back DOS base for loader
        beq.s .nodosexit                ;exit if no DOS here

        move.l d0,a6
        move.l Start-4(pc),a0           ;Get BPTR of next hunk
        adda.l a0,a0
        adda.l a0,a0
        exg.l a0,a2                     ;move to a2
        move.l d2,d0                    ;restore argument
        jsr 4(a2)                       ;jump in
        move.l d0,d2                    ;Save return code

        move.l ol_SysBase(pc),a6
        move.l ol_DOSBase(pc),a1
        jsr CloseLibrary(a6)            ;Close the lib

        move.l d2,d0                    ;Returncode in d0
        rts
.nodosexit:
        move.l #$07038007,d7            ;DOS didn't open
        move.l $114(a6),a5
        jsr Alert(a6)
        moveq #30,d0                    ;Something went really wrong !
        rts

DOSName:        dc.b "dos.library",0
```

9.4 Structures within Hunks

While the AmigaDOS loader, i.e. `LoadSeg()` and related functions, do not care about the contents of the segments it loaded, some other components of AmigaDOS do actually analyze their contents.

9.4.1 The Version Cookie

The `Version` command scans a ROM-resident modules or all segments of a binary for the character sequence `$VER:` and if such a sequence is found, the string following is scanned. The syntax of the string consists of the following elements:

- The version cookie `$VER:`
- one or multiple blank spaces
- a program name, which is output by the `Version` command
- a decimal number, representing the *version* of the program
- a single dot (“`.`”)
- a decimal number, representing the *revision* of the program
- one or multiple blank spaces
- an opening bracket (“`(`”)
- a decimal number, representing the day of the month of the revision
- a single dot (“`.`”)
- a decimal number, representing the month of the revision
- a single dot (“`.`”)
- a decimal number, representing the year of the revision
- a closing bracket (“`)`”)
- an optional comment that is only output if the `FULL` option of the `Version` is given.

If the number representing the year is below 1900, the `Version` command assumes a two-digit year and either adds 200 if the year is below 78, or 1900 otherwise. The command then re-formats the date according to the currently active locale and prints it to the console, along with the program name and, optionally, the comment string.

An example for the version cookie is

```
const char version[] = "$VER: RKRM-Dos 45.3 (12.9.2023) (c) THOR";
```

Note that the date follows the convention date of the month, month and year, here September 12, 2023. The version in this example is 45, the version is 3. The string behind the date is a comment and usually not printed by the `Version` command.

9.4.2 The Stack Cookie

The workbench, the shell, and also `GetDeviceProc()` when loading *handlers* scan the loaded binary for the string sequence `$STACK:`. If this string sequence is found, AmigaDOS attempts to read a following decimal number, and interprets this as stack size in bytes.

The stack of the program is then, potentially increased to the provided size. Note that AmigaDOS also scans alternative sources for a stack size: The `Stack` setting in the icon information window of the workbench, the `Stack` command of the shell, or the `STACKSIZE` entry in the mount list. The stack size indicated by the above stack cookie does not override these settings, it can only *increase* the stack size. This allows program authors to ensure that the stack of their program has a necessary minimal size, though still allows users to increase it if necessary.

An example for the stack cookie is

```
const char stack[] = "$STACK: 8192";
```

This ensures that the stack size of the program is at least 8192 bytes.

9.4.3 Runtime binding of BCPL programs

BPCL programs depend on a *Global Vector* that includes function entries and data available to all its modules. AmigaDOS includes a runtime binder functionality that creates the *Global Vector* from data found in the segments of the loaded binary and the *dos.library Global Vector*.

Even though this mechanism is deprecated and AmigaDOS has long been ported to C and assembler, some components still depend on this legacy mechanism, namely all *handlers* and *file systems* mounted with the `GLOBVEC = 0` or `GLOBVEC = -2` entry in the mount list. While newer *handlers* should not depend on this mechanism anymore, the `Port-Handler` mount entry is created in the Kickstart ROM as BCPL handler, beyond control of the user. The same goes for the `Shell`, which is also initiated as BCPL process.

If the above components — the `Port-Handler` or the *Shell* — are attemted to be customized, implementors need to be aware that these processes are not started from the first byte of the first segment, but from the *Global Vector* entry `#1`, which contains the address of the `START` function from which the process is run. All other entries of the vector are of no concern, and should not be used anymore.

The *Runtime Binder* of AmigaOs, initiated only for BCPL handlers and BCPL processes such as the *Shell*, now scans the segments of such programs for information on how to populate the *Global Vector*.

The *first long-word* of the segment, usually the entry point of the process, contains the long-word offset from the start of the process to the start of the *Global Vector* initialization data. This initialization data is *scanned backwards* from the given offset towards lower addresses, starting with the long-word before the offset.

The first long-word of the initialization data is the required size of the *Global Vector* the program requires. This value is only used to check the following initializers for validity. The standard system global vector currently requires 150 entries, which is a safe choice.

All following entries consist of pairs of long-words, scanned again towards smaller addresses, where each pair defines one entry in the global vector. The first (higher address) long-word is the offset of the function which will populate the *Global Vector*, the second (lower) address is the index of the *Global Vector* entry. An offset of 0 terminates the list.

The following assembler stub may be used as initial segment of an (otherwise C-based) handler that instructs the *Runtime Binder* to populate the `START` vector, and then calls into the `@main` function. BCPL unfortunately also uses a custom call syntax, register `a6` is the address of the BCPL return code which cleans up the BCPL stack frame and returns to the caller.

```
                SECTION text,code

                XREF    @main           ; handler or shell main

G_GLOBMAX       EQU     150             ; size of GV
G_START         EQU     1               ; BCPL START functions
G_CLISTART      EQU     133             ; shell startup

CodeHeader:     DC.L    (CodeEnd-CodeHeader)/4

*               C Startup function, called for GlobVec=-1 or -3
*               see text below why this works
CStart:         sub.l   a0,a0           ; no startup message
                bsr.w   @main           ; need to GetMsg() manually in main
                rts

                CNOP    0,4

*               BCPL startup function, called for GlobVec=0 or -2
BCPLStart:      movem.l a0-a6,-(a7)     ; save all these, BCPL uses them
                lsl.l   #2,d1           ; get startup packet
                move.l  d1,a0           ; move to a0
                bsr.w   @main           ; get the ball rolling
                movem.l (a7)+,a0-a6     ; restore everyone
                jmp     (a6)            ; BCPL-style return

BCPLTable:      CNOP    0,4

                DC.L    0               ; End of global list
                DC.L    G_CLISTART,BCPLStart-CodeHeader ;only for the shell
                DC.L    G_START,BCPLStart-CodeHeader
                DC.L    G_GLOBMAX       ; max global used (standard value)
CodeEnd:
```

Note that there are other differences for BCPL handlers the `main()` function of the handler needs to take care of, see also section **??**. BCPL handlers do not receive their startup message in the process *MsgPort*, but rather receive it as first BCPL argument in register `d1`. It is here converted to a C-style pointer and provided as first argument to the main function, assuming the SAS/C registerized parameters ABI.

The `CStart` label is not called at all if the handler is mounted with `GLOBVEC=0` or `GLOBVEC=-2`, and thus would be, in this example, not required. It is included here to demonstrate another technique, namely *dual use* handlers that can be mounted both as C and as BCPL handlers.

The above startup code also allows `GLOBVEC=-1` in the mount list. In this case, the code *is* started from the first byte, which is, actually, the long-word offset to the BCPL initializer. In this particular case, it also assembles to a harmless `ORI` instruction, provided the offset to the end of the code is short enough, i.e. below 64K. It is therefore harmless. As the calling syntax is different, the main function is now called with a `NULL` argument, and then needs to wait for the startup package itself, see again section **??**.

For the *Shell*, in particular, a second entry in the *Global Vector* shall be populated, namely the entry `G_CLISTART` at offset `#133`. It may point to the same entry code, more on this in section **??**. As this entry is not used for handler startup, it does not hurt to include it in the BCPL initializer in either case, and thus the above code may be used as universal "BCPL kludge" for both the shell and legacy handlers that depend on BCPL startup.

9.5 Object File Format

Object files are intermediate output files of a compiler or assembler, generated from one translation unit, i.e. typically one source code file. Such files can still contain references to symbols that could not be resolved within the translation unit because the corresponding symbol is defined in another unit. The linker then combines multiple object codes, resolving all unreferenced symbol, and generates an executable binary file as output.

The overall structure of an object file is depicted in table 37:

Table 37: Object File Format

Size	Code	Syntax
?	`HUNK_UNIT`	Defines the start of a translation unit, see 9.5.1
	`do {`	Multiple segments follow
?	`  HUNK_NAME`	Name of the hunk, defines hunks to merge, see 9.1.8
2	`  `m_t	Read the memory type of the next hunk
30	`  `h_t	Read the next hunk type
	`  if (`h_t `== HUNK_CODE) parse_CODE`	Code and constant data, see 9.1.2
	`  else if (`h_t `== HUNK_DATA) parse_DATA`	Data, see 9.1.3
	`  else if (`h_t `== HUNK_BSS) parse_BSS`	Zero-initialized data, see 9.1.4
	`  else if (`m_t `!= 0) ERROR_BAD_HUNK`	Upper bits shall be 0 for all other hunks
	`  else do {`	Loop over auxiliary information
	`    if (`h_t `== HUNK_RELOC32) parse_RELOC32`	32-bit relocation, see 9.1.5
	`    else if (`h_t `== HUNK_RELOC32SHORT) parse_RELOC32SHORT`	32-bit relocation, see 9.1.6
	`    else if (`h_t `== HUNK_RELRELOC32) parse_RELRELOC32`	32-bit PC-relative relocation, see 9.1.7
	`    else if (`h_t `== HUNK_RELOC16) parse_RELOC16`	16-bit PC-relative relocation, see 9.5.2
	`    else if (`h_t `== HUNK_RELOC8) parse_RELOC8`	8-bit PC-relative relocation, see 9.5.2
	`    else if (`h_t `== HUNK_DRELOC32) parse_RELOC32`	32-bit base-relative relocation, see 9.5.4
	`    else if (`h_t `== HUNK_DRELOC16) parse_RELOC16`	16-bit base-relative relocation, see 9.5.5
	`    else if (`h_t `== HUNK_DRELOC8) parse_RELOC8`	8-bit base-relative relocation, see 9.5.6
	`    else if (`h_t `== HUNK_EXT) parse_EXT`	External symbol definition, see 9.5.7
	`    else if (`h_t `== HUNK_SYMBOL) parse_SYMBOL`	Symbol definition, see 9.1.9

	`else if (`h_t `== HUNK_DEBUG) parse_DEBUG`	Debug information, see 9.1.10
	`else if (`h_t `== HUNK_END) break`	abort this segment
	`else ERROR_INVALID_HUNK`	an error
32	h_t	Read next hunk type
	`} while(true)`	Repeated until `HUNK_END`
	`} while(!EOF)`	Repeated with the next hunk until the file ends

9.5.1 HUNK_UNIT

This hunk identifies a translation unit and provides it a unique name. This hunk shall be the first hunk of an object file. A translation unit typically refers to one source code file that has been processed by the compiler or assembler. Typically, the name of the unit is ignored.

The structure of this hunk is as follows:

Table 38: Hunk Unit Syntax

Size	Code	Syntax
	`HUNK_UNIT [0x3e7]`	A hunk identifying a translation unit
32	`l`	Size of the name in long-words
$32 \times l$	h_n	Unit name

The size of the name is not given in characters, but in 32-bit units. The name is possibly zero-padded to the next 32-bit boundary to fill an integer number of long-words. If the name fills an entire number of long-words already, it is *not* zero-terminated.

Even though not enforced by the format, linkers can limit the size of the unit.

9.5.2 HUNK_RELOC16

This hunk defines relocation information of one hunk into another hunk, and its format is identical to `HUNK_RELRELOC32`, see section 9.1.7 and table 28.. Relocation offsets are therefore 32 bits long, though the elements to relocate at offset r_o within the segment are only 16 bits in size, and refer to PC-relative addressing modes, such as branches.

Table 39: Hunk Reloc16 Syntax

Size	Code	Syntax
	`HUNK_RELOC16 [0x3ed]`	16-bit PC-relative relocation information
	`...`	See table 28

This restricts possible displacements to 16 bits, and thus the segment containing the 16-bit value to be relocated and the segment the hunk is relative to shall be linked together to form a single segment in the final output. In the notation of section 28, the segments i and j shall thus be merged, and for that shall have the same names as assigned by `HUNK_NAME`.

However, it may still happen that the joined segment generated by merging two or more segments together is too long to allow 16 displacements. In such a case, the relocation can obviously not performed. In such a case, linkers either abort with a failure, or generate an *automatic link vector*. The PC-relative branch or jump to an out-of-range target symbol is then replaced by the linker with a branch or jump to an intermediate "automatic" vector, which then performs a 32-bit absolute jump to the intended target.

While such *automatic link vectors* or short *ALV*s solve the problem of changing the program flow by 16-bit displacements over distances exceeding 16-bit, *ALV*s do not work correctly for data that is addressed by 16-bit PC relative modes. Instead of referencing the intended data, the executing code would then see the *ALV* as data.

Thus, authors of compilers or assemblers should disallow data references across translation unit boundaries with 16-bit PC-relative addressing modes as those can trigger linkers to incorrectly generate *ALV*s. Linkers should also generate a warning when creating *ALV*s.

9.5.3 HUNK_RELOC8

This hunk defines relocation information of one hunk into another hunk, and its format is identical to `HUNK_RELRELOC32`, see section 9.1.7 and table 28.. Relocation offsets are therefore 32 bits long, though the elements to relocate at offset r_o within the current segment are only 8 bits in size, and thus refer to short branches.

Table 40: Hunk Reloc8 Syntax

Size	Code	Syntax
	`HUNK_RELOC8 [0x3ee]`	8-bit PC-relative relocation information
	`...`	See table 28

As for `HUNK_RELOC16`, the linker can generate *ALV*s in case the target offset is not reachable with an 8-bit offset. However, as the possible range for displacement is quite short, it is quite likely that the generated *ALV* itself is not reachable, and thus relocation during the linking phase is not possible at all. Thus, short branches between translation units should be avoided.

Otherwise, the same precautions as for `HUNK_RELOC16` should be taken, i.e. short displacements to data over translation unit boundaries should be avoided as proper linkage cannot be ensured.

9.5.4 HUNK_DRELOC32

This hunk defines relocation of 32-bit data elements within a hunk that is addressed relative to a base register. The name of this hunk shall be `__MERGED`, indicating that the hunk contains data and zero-initialized elements in the *small data model*.

The format of this hunk is identical to `HUNK_RELOC32`, see section 9.1.5 and table 26, where each 32-bit wide relocation offset r_o points to a long-word within the preceeding data or BSS hunk. The long-word at this offset is then adjusted by the position of the first byte of this hunk relative to the start of the `__MERGED` hunk into which this hunk is merged.

Table 41: Hunk DReloc32 Syntax

Size	Code	Syntax
	`HUNK_DRELOC32 [0x3f7]`	32-bit base-relative relocation information
	`...`	See table 26

The hunks named `__MERGED` are typically generated by comilers or assemblers when implementing the *small data model* within which all non-static objects are addressed relative to a base register. Typically, register `a4` is used for this purpose, and it is loaded by the compiler startup code to point either to the start of the `__MERGED` hunk, or 32K into the hunk such that both negative and positive offsets relative to `a4` can be used. The name *small data model* stems for the limitation to 64K of data as the 68000 uses only 16-bit for base-relative addressing.

9.5.5 HUNK_DRELOC16

This hunk defines relocation of 16-bit data elements within a hunk that is addressed relative to a base register, i.e. __MERGED hunks in the *small data* memory model.

The format of this hunk is identical to HUNK_RELOC32, see section 9.1.5 and table 26, where each 32-bit wide relocation offset r_o points to a 16-bit word within the preceeding data or BSS segment. The word at this offset is then adjusted by the position of the first byte of this hunk relative to the start of the __MERGED hunk into which this hunk is merged.

Table 42: Hunk DReloc16 Syntax

Size	Code	Syntax
	HUNK_DRELOC16 [0x3f8]	16-bit base-relative relocation information
	...	See table 26

9.5.6 HUNK_DRELOC8

This hunk defines relocation of 8-bit data elements within a hunk that is addressed relative to a base register, i.e. __MERGED hunks in the *small data* memory model.

The format of this hunk is identical to HUNK_RELOC32, see section 9.1.5 and table 26, where each 32-bit wide relocation offset r_o points to a byte within the preceeding data or BSS hunk. The byte at this offset is then adjusted by the position of the first byte of this hunk relative to the start of the __MERGED hunk into which this hunk is merged.

Table 43: Hunk DReloc8 Syntax

Size	Code	Syntax
	HUNK_DRELOC8 [0x3f9]	8-bit base-relative relocation information
	...	See table 26

9.5.7 HUNK_EXT

This hunk defines symbol names and corresponding symbol offsets or values within the currently loaded segment. It is quite similar to HUNK_SYMBOL except that it not only includes symbol definitions, but also symbol references. The linker uses this hunk to resolve symbols with external linkage.

The syntax of this hunk reads as follows:

Table 44: Hunk EXT Syntax

Size	Code	Syntax
	HUNK_EXT [0x3ef]	A hunk assigning symbols to positions within a segment
	do {	Repeat ...
8	s_t	Symbol type
24	s_l	Symbol length in long-words
	if (s_l == 0) break	Terminate the hunk
32 × s_l	s_n	Symbol name, potentially zero-padded
	if (s_t < 0x80) {	Symbol definition?
32	s_v	Symbol value
	} else {	Symbol reference
	if (s_t == 0x82 || s_t == 0x89)	A common block?
32	s_c	Size of the common block in bytes

	}	End of common block
32	s_n	Number of references of this symbol
	while(−− $s_n \geq 0$) {	Repeat over the references
32	$s_o[s_n]$	Offset into the hunk of the reference
	}	End of loop over symbols
	} while(true)	until zero-sized symbol

The length of the symbol name is encoded in long-words, not in characters. If it does not fill an integer number of long-words, it is zero-padded; the name is not zero-terminated if it does fill an integer number of long-words, though.

The symbol type s_t defines the nature of the symbol. The symbol types are defined in `dos/doshunks.h` and shared with the `HUNK_SYMBOL` hunk, see section 9.1.9.

Entries of a symbol type with s_t < `0x80` are symbol definitions, the symbol value is defined by s_v. See table 45 for possible types of symbol definitions.

The next table includes symbol types that identify symbol references, i.e. they are referenced within a segment of an object file, though not defined there. These types can clearly not be contained within a executable binary, but they may appear within an object file and are then resolved by corresponding symbol definitions from the above table by the linker:

Table 45: Symbol types in HUNK_EXT

EXT_REF32	[0x81]	Reference to a 32-bit symbol that is resolved by a corresponding EXT_ABS to an absolute value or by a EXT_DEF definition to a relocation information to this or another segment.
EXT_COMMON	[0x82]	Reference to a 32-bit symbol that may be resolved by a EXT_ABS or EXT_DEF definition, but if no such definition is found, a BSS hunk of the maximal size of all references to the symbol is created by the linker. Thus, this type generates a zero-initialized object if no definition is found.
EXT_REF16	[0x83]	Reference to a 16-bit PC relative offset within the same segment.
EXT_REF8	[0x84]	Reference to a 8-bit PC relative offset within the same segment.
EXT_DREF32	[0x85]	32-bit reference relative to a base register (typically a4), resolved by the linker through an entry in a HUNK_DRELOC32 hunk.
EXT_DREF16	[0x86]	16-bit reference relative to a base register, resolved by the linker through an entry in a HUNK_DRELOC16 hunk.
EXT_DREF8	[0x87]	8-bit reference relative to a base register, resolved by the linker through an entry in a HUNK_DRELOC8 hunk .
EXT_RELREF32	[0x88]	32-bit PC-relative reference for 32-bit address, this will be resolved by an EXT_DEF definition into an entry into a HUNK_RELRELOC32 hunk by the linker.
EXT_RELCOMMON	[0x89]	32-bit PC relative common reference for a 32-bit address. Similar to a EXT_COMMON definition, this will be resolved into an HUNK_RELRELOC32 entry where potentially space for the symbol will be allocated in a BSS segment if no corresponding definition is found.
EXT_ABSREF16	[0x8a]	16-bit absolute reference, resolved by the linker to a 16-bit value by an EXT_ABS definition.
EXT_ABSREF8	[0x8b]	8-bit absolute reference, resolved by the linker to an 8-bit value through an EXT_ABS definition.

For references, s_n identifies the number of times the symbol is referenced, while s_o defines the offsets into the current segment where the symbol is used and into which the symbol value will be resolved during linking. This value comes from an s_v entry in a HUNK_EXT hunk from another translation unit.

Common symbols are symbols that are defined in multiple translation units. The size of the symbol required by the translation s_c. If no corresponding symbol definition is found, the linker then allocates space of a size that is determined by the maximum of all s_c values found for the same symbol in all translation units. The symbol will then be created within a BSS segment by the linker without an explicit symbol definition. This mechanism is mostly used by FORTRAN and is therefore rarely used. It is likely that many linkers do not support such symbols.

9.6 Link Library File Format

Link Library files are collections of small compiled or assembled program sections that provide multiple commonly used symbols or functions. Unliky AmigaOs libraries, which are loaded dynamically at run time, link libraries resolve undefined symbols at link time; functions or symbols within them become a permanent part of the generated executable.

The *amiga.lib* is a typical example for a link library. It contains functions such as CreateExtIO(). While newer versions of the *exec.library* contains with CreateIORequest() a similar function, some manual work was required for creating an IORequest structure in exec versions prior v37. To ease development, it was made available in a (static) library whose functions are merged with the compiled code.

Link libraries come in two forms: Old-style non-indexed libraries, and indexed libraries that are faster to process.

9.6.1 Non-indexed Link Libraries

Old-style or non-indexed link libraries are simply a concatenation of object files in the form presented in section 9.5 and table 37. Then, of course, one translation unit as introduced by HUNK_UNIT **??** is not necessarily terminated by an EOF as specified in table 37, but possibly by the subsequent program unit, starting with another HUNK_UNIT.

Non-indexed link libraries do not require any tools beyond a compiler or assembler for buliding them. The AmigaDOS JOIN command is sufficient to build them. The drawback of such libraries is that they are slow to process as the linker needs to scan the entire library to find a specific symbol.

9.7 Miscellaneous Functions

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 $1/50^{\text{th}}$ 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