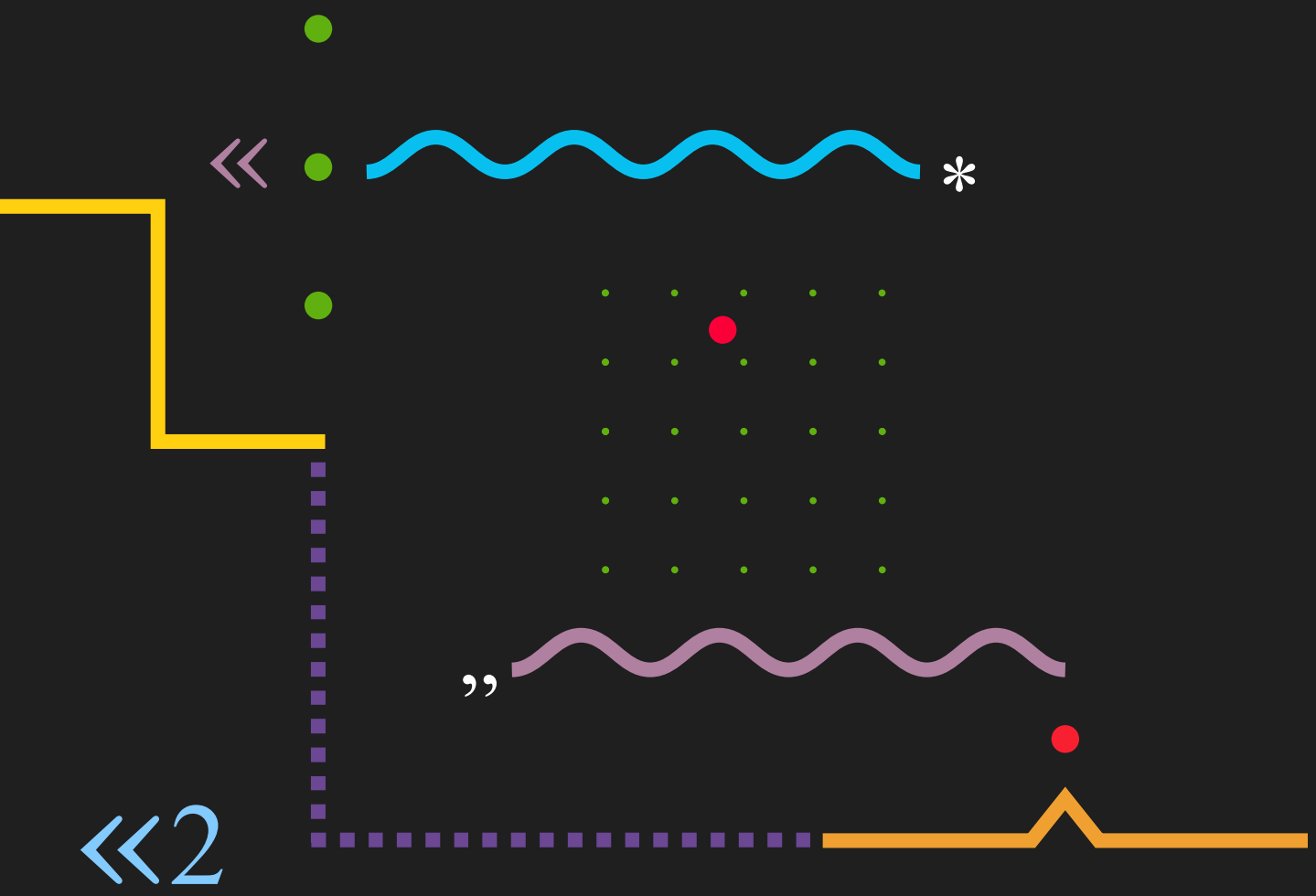
AMIGA ROM Kernel Reference Manual

AmigaDOS



AMIGA TECHNICAL REFERENCE SERIES

FIRST EDITION

Amiga ROM Kernel Reference Manual

AmigaDOS

THOMAS RICHTER

Contents

Chapter 1

Introduction

1.1 Foreword

The purpose of this manual is to provide a comprehensive documentation of the AmigaDOS subsystem of the Amiga Operation System. AmigaDOS provides elementary services similar to other contemporary operating systems, such as file systems and handlers implementing stream-based input and output, process management, volume and device management, command line execution and command line parsing. Its interface to applications is the *dos.library*, a ROM based shared library that offers an interface between such applications and AmigaDOS subsystems such as file systems or the Shell.

While the Amiga ROM Kernel Reference Manuals [**?**] document major parts of the AmigaOs, they do not include a volume on AmigaDOS and its subsystems. This is due to the history of AmigaDOS which is nothing but a port of Tripos, an experimental operating system developed at the University of Cambridge. Instead, its documentation became available as the AmigaDOS manual [**?**] separately. This book 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 now out of print, does not reflect the current state of AmigaDOS anymore, and leaves multiple parts of AmigaDOS undocumented.

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

This book attempts to fill this gap and attempts to provide a comprehensive and complete documentation of AmigaDOS and its subsystems, closely following the style of the ROM Kernel Reference Manuals.

1.1.1 Acknowledgments

A book of this size would be impossible without additional help from others. I want to thank in particular Frank Wille for helping me to compile the description of the Amiga binary format and Olaf Barthel for many fruitful discussions and for answering many questions while compiling many chapters of this volume. Special thank goes to Rainer Müller for proof-reading the manuscript and providing many fixes and corrections.

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 AmigaDOS. The words *should* and *should not* indicate best practice 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 remembering! 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 an interface of *AmigaDOS*, and are also used when new terms are introduced for the first time. Data structures and 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 Concepts

2.1 Purpose of the dos.library

AmigaDOS is part of the *Amiga Operating System* or short *AmigaOs*. The *dos.library* provides the application interface to AmigaDOS, even though it consists of components beyond this library, e.g. the *AmigaDOS Shell* and the *Fast File System*, which is the default (ROM-based) file system organizing the data on volumes such as floppy disks or hard disks.

Unlike many other operating systems, 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*. The FFS is only one of the multiple file systems AmigaDOS is able to handle; instead, the library provides an abstract interface to handlers that enables extensions through third-party file systems and handlers. Handlers and file systems are introduced in section **??**, and the interface between the *dos.library* and such handlers is specified in chapter **??**.

2.2 The DosLibrary Object

The *dos.library* is typically opened by the startup code of most compilers, and its base pointer is placed into the `DOSBase` object by such startup code:

```
struct DosLibrary *DOSBase;
```

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

The *DosLibrary* structure is defined in `dos/dosextens.h`, but most programs do not require and should not access to its elements. However, chapter **??** documents its structure and objects referenced by it. Instead, the library offers functions to application programs through its library vector offsets — or short `_LVO`s — which are made available to a C compiler by including `proto/dos.h`:

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

This book is organized around groups of functions defined through the above file, roughly by functionality and by increasing complexity.

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

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

```
#include <dos/dos.h>

struct DosLibrary *DOSBase;
struct ExecBase *SysBase;

int __saveds startup(void)
{
  SysBase = *((struct ExecBase **)(4L);
  ...
  if ((DOSBase = (struct DosLibrary *)(OpenLibrary(DOSNAME,47)))) {
    ...
    CloseLibrary((struct Library *)DOSBase);
  }
  ...
}
```

The most up to date version of the *dos.library* also discussed in this book is version 47, as indicated by the second argument of `OpenLibrary()`. Not all programs will require its most recent version, though; the version in which a particular function of the *dos.library* appeared is indicated behind the function definition. Rather than requesting the maximum version, program authors should identify which functions of the library they require, identify for each of them the minimum version number in which they became available, and then take the maximum of all the version numbers and supply this maximum as second argument of `OpenLibrary()`. Table 2.1 provides the relation between AmigaDOS versions and the corresponding library version:

Table 2.1: AmigaDOS Version Numbers

AmigaDOS version	dos.library version
1.2	33
1.3	34
1.4 (RAM-based, with Hedley support)	35
2.0	36
2.04	37
2.1	38
3.0	39
3.1	40
3.1 (unpublished, with support for Japanese locale)	41
3.1 (unpublished alpha release)	42
3.2 (unpublished release for the Walker)	43
3.5	44
3.9	45
3.1.4	46
3.2	47

Version 35 was an extension of version 34 with integrated support for the A2024 Hedley monitor that was entirely RAM-loaded. Most important changes were made in version 36 in which the library was largely extended, though the release contained many defects that were addressed in version 37. Version 38 was a pure software update that added localization support. Versions 41 to 43 were never published, with the exception of some beta versions of the FFS which shipped as version 43. Version 44 was used for AmigaOs 3.5, and 45 for AmigaOs 3.9 and most modules of 3.1.4. Some modules with extended features such as the FFS with long file name support received the version number 46.

2.3 Booleans

AmigaDOS uses a convention for Booleans that differs from the one imposed by the C programming language; it uses the following truth values defined in the file `dos/dos.h`:

Table 2.2: DOS Truth Values

Define	Value
DOSFALSE	0
DOSTRUE	-1

Note that the C language uses instead the value 1 for `TRUE`. Code that checks for zero or non-zero return codes will function normally, however it shall not compare to `TRUE` in Boolean tests. Instead, programs shall test whether a value is 0 or different from 0, avoiding this discrepancy between the conventions of AmigaDOS and the C language.

2.4 Pointers and BPTRs

AmigaDOS is a descendant of the *Tripos* operating system and as such was originally implemented in the BCPL language. As of Kickstart 2.0, i.e. version 36 of the *dos.library*, AmigaDOS was re-implemented in C and assembler, but this new implementation had to preserve the existing interface based on BCPL conventions.

BCPL is a type-less 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 expresses the positions of its objects as indices of their first 32-bit element in memory. As each 32-bit group is assigned its own index, one can obtain such an index by dividing the byte-address of an object by 4, or equivalently, by right-shifting the address as given by a conventional pointer by two bits. This also has the consequence that (most) objects 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 BCPL object, its index is multiplied by 4, or left-shifted by 2 bits.

> *Not on the Stack!* Since BCPL objects must have addresses divisible by 4, using automatic storage duration for AmigaDOS objects is inappropriate. Compilers will typically keep automatic objects on the hardware stack of the 68K processor [**?**, **?**], but usually do not ensure that their addresses are aligned to long word boundaries. In case a particular AmigaDOS object cannot be constructed by `AllocDosObject()` (see section **??**), a safe strategy is to use *exec.library* memory allocation functions such as `AllocMem()` or `AllocVec()` to obtain memory for holding them. All three functions ensure proper alignment.

Indices to 32-bit memory cells in the BCPL abstraction of computer memory are called *BCPL pointers* or short *BPTR*s, even though they are not pointers in the sense of the C language; they are 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 index */
```

The include file `exec/types.h` contains the definition of an untyped C pointer as follows:

```
typedef void *APTR;                    /* 32-bit untyped pointer */
```

This is – unlike a BPTR – a real pointer, though the data type it points to remains undefined.

Conversion from BCPL pointers to conventional C pointers and back are realized 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)
```

Luckily, 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 object. Examples for those objects are *file handles* specified in chapter **??**, and *locks*, see chapter **??**. Both are represented as BPTRs to some structure the caller usually does not need to care about.

It is certainly a burden to always allocate temporary BCPL objects from the heap through the *exec.library* or the *dos.library*, and doing so can also fragment the AmigaOs memory unnecessarily. However, allocation of automatic objects from the stack does not ensure long-word alignment in general. To work around this burden, one can use a trick and instead request from the compiler a somewhat larger object with automatic storage duration 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 a 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 aligned to a long-word boundary and thus can be safely 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 BCPL as BPTRs are indices instead. 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 conceptually something different: The first being the index to the first element of the host memory array, the later the pointer to the first address.

2.5 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 BCPL language uses a different convention. Instead, a BCPL string is 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 index of a BCPL string  */
```

Luckily, 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 them. 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 BSTRs 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 maximum value a byte-sized length value can take.

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

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

Finally, the `NUL` character — note the single "`L`" — is the name of the ASCII character with code 0 by which all C strings are terminated. A C string is therefore a `NUL`-terminated string. This notation will be used throughout this volume.

> *All Zero, but of a Different Kind* `ZERO`, `NULL` and `NUL` all encode the value 0, but the first is the name of the first BCPL memory index and indicates an invalid BPTR. Its C equivalent is `NULL`, which however denotes a pointer and not a memory cell index. `NUL` is the first code point of the ASCII code set and represented by a byte of value 0.

2.6 Elementary Conversion Functions

Functions in this section perform conversions between the elementary data types listed in this chapter; they convert between strings representing numbers as human-readable decimals and their binary machine representations. The `StrToLong()` takes such a string and converts it to an integer. There is no function in the *dos.library* to perform the inverse conversion of an integer to a string, though the *exec.library* function `RawDoFmt()` may be used to implement it as a by-product of a more general family of functions. Section **??** provides a very compact and in many cases sufficient implementation of a function that closely reassembles the `sprintf()` function of ANSI-C; it prints and formats many elementary data types, including integers, into an output buffer, and as such, can also convert an integer into a decimal number.

2.6.1 Convert a String to a Number

The `StrToLong()` function converts an ASCII encoded decimal number to a signed 32 bit integer.

```
characters = StrToLong(string,value) /* since V36 */
D0                     D1    D2

LONG StrToLong(STRPTR, LONG *)
```

This function receives a `NUL`-terminated `string` containing a decimal number encoded in ASCII and converts it to a signed integer. The return value is the number of characters it could parse from the string, or -1 if not a single valid digit could be found. The converted number is placed into the 32-bit long word pointed to by `value`.

The function skips leading spaces and tabs, they are included in `characters`. It also interprets a leading minus sign ("−") to indicate negative numbers, but *does not* accept a leading plus sign ("+"). This

function also aborts in case the conversion overflows, i.e. the absolute value of the number is larger than 2^{31}, and then returns the number of characters up to which the conversion could be performed without overflow. The result of the conversion filled into `value` is in this case meaningless.

Even in case of error, this function does not alter `IoErr()`.

2.6.2 Print Formatted into a Buffer

While not a function of the *dos.library*, the code example in this section implements a function similar to the `sprintf()` function of the ANSI C library; it converts many elementary data types to strings. It is based on the `RawDoFmt()` function of the *exec.library*, which is also patched by the *locale.library* and thus formats numbers according to the currently loaded locale.

```
#include <stdarg.h> /* for va_list macros */

static void prbuf(char c)
{
  __builtin_emit(0x16c0);   /* move.b D0,(A3)+ */
}

/*
** convert a list of arguments to a string using an
** ANSI-C format template.
*/
void vsprintf(char *buffer, const char *ctlstr, void *args)
{
  RawDoFmt((char *)ctlstr, args, prbuf, buffer);
}

/*
** Convert multiple arguments to a string, using
** an ANSI-C format template.
*/
void sprintf(char *buffer, const char *ctrl,...)
{
  va_list args;

  va_start(args,ctrl);
  vsprintf(buffer,ctrl,args);
  va_end(args);
}

/*
** Convert a signed integer to a string
*/
void LongToStr(char buffer *buf, LONG val)
{
  sprintf(buffer,"%ld",val);
}
```

The `vsprintf()` function defined above takes a target `buffer`, an ANSI-C conversion string `ctlstr` containing formatting directives, see [**?**], and a buffer of primitive integer data types or strings in `args`. It converts these arguments to strings, formats them according to `ctlstr` and places the result in the target

buffer. The full set of conversion directives is found in the description of `RawDoFmt()` in [**?**], though the most elementary directives are shown here:

`%s` The next argument in `args` is a pointer to a `NUL`-terminated string that is inserted into `buffer`. To limit the number of characters copied into the target buffer, an ANSI-C precision field should be included in the format directive, e.g. `%.30s` truncates the input string to 30 characters.

`%b` The next argument is a BPTR to a BSTR that is inserted into the `buffer`. This formatting directive was added in AmigaOs version 36. Again, it is recommended to truncate the string length by the ANSI-C precision format directive, see above.

`%ld` The next argument in `args` is a 32-bit signed integer that is converted to a decimal string.

`%lu` Convert a 32-bit unsigned integer to a decimal string — this formatting directive was added in AmigaOs 37.

`%lx` Convert a 32-bit unsigned integer to hexadecimal. If needed, a leading `0x` must be inserted manually, it is not part of the output of the conversion.

`%lc` Interpret a 32-bit integer as an ISO-Latin or ASCII code point and insert the single character this code point represents.

`%%` Insert the percent sign itself.

As also seen above, the ANSI-C flags, field width, precision and length modifiers may be included in the format directive starting with "`%`", see [**?**] for more details.

> *Think Long* The knowledgeable C developer will notice that many of the above conversion directives include an `l` length modifier which has been added here on purpose, even causing strange directives such as `%lc`. This is because the `RawDoFmt()` function assumes a 16-bit integer model, whereas most compilers operate with 32-bit integers. The length modifier ensures that the *exec.library* removes 32-bit arguments from `args`, corresponding to the integer size of many C implementations. Some older compilers use, in fact, a 16-bit integer model in which case the `l` shall be dropped (but only then!). Not following this guideline will cause hard to find bugs resulting in incorrect output.

The `vsprintf()` function from the above code segment implements the function of the same name in the ANSI-C library only approximately, and the formatting directives need to be adjusted carefully to avoid problems. There is, as in the ANSI-C standard library, no check whether the target buffer is large enough. In particular, string formatting directives should be selected carefully to avoid buffer overruns that can lead to system crashes. As the `FPrintf()` function from section **??** and the `Printf()` function from section **??** are based on the same exec function, the same peculiarities apply there as well.

The `sprintf()` function is an — albeit basic — re-implementation of the C standard library function `sprintf()` which prints all its arguments to a buffer. The same formatting directives apply. It uses the varargs macros of ANSI-C from `stdarg.h` to forward its arguments to `vsprintf()`.

The `LongToStr()` function is a simple application of `sprintf()` to convert a signed integer to a string. It does the inverse of the `StrToLong()` function of the *dos.library* introduced in section **??**.

The `prbuf()` function is hack using a built-in function of the SAS/C compiler. Its function body consists only of the `__builtin_emit()` function which injects the object code of the `MOVE.B D0,(A3)+` instruction into the function body. It takes the function argument placed in register `D0` by `RawDoFmt()` and pushes it into the target buffer pointed to by register `A3`. Other compilers will need a small assembler stub function of the same name that consists only of this instruction, and an `RTS`.

2.7 Paths

Paths are human-readable strings that uniquely identify an object on a file system, such as a file or a directory, or represent an interface of the computer system to the outside world, such as the serial or parallel port. Chapter **??** introduces paths in more detail.

2.8 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 write to. Files are explained in more detail in chapter **??**. AmigaDOS represents files through *file handles* in the form of a BPTR to a *FileHandle* structure, though most of the time, the elements of the structure are not needed and it is sufficient to pass the BPTRs around.

2.9 Directories

Directories are collections of files logically grouped together. They correspond to drawers on the workbench. Each medium such as a floppy disk, and each partition on a harddisk has at least one top-most directory; it contains all the objects that are immediately visible if the icon representing the medium is double-clicked on the Workbench. This top-most directory is also called the *root directory* of the disk, partition or medium. Often, directories such as the root directory can also contain other directories as *sub-directories*.

2.10 Locks

Locks are access rights to a particular object on a file system, such as a file or a directory. A locked file cannot be overwritten or removed by any other process, a locked directory can be altered, but not be deleted. Chapter **??** provides more details on locks. AmigaDOS represents locks through the *FileLock* structure also introduced in the above chapter, though in most cases, locks are passed around as BPTRs.

2.11 Processes

AmigaDOS is a multi-tasking system operating on top of the *exec* kernel [**?**]. As such, it can execute 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*; it includes additional state information relevant to AmigaDOS, such as a *current directory*, a default file system, a *console* it is connected to, and default input, output and error streams. Most important, each process includes a `MsgPort`, see [**?**], through which it communicates with other AmigaDOS components such as handlers or file systems. Processes are explained in more detail in chapter **??**.

2.12 Handlers and File Systems

Handlers are special processes that perform input or output operations to logical or physical devices, such as the serial port, a printer, the floppy or even the main memory. The *dos.library* delegates most operations, such as reading from a stream or file to such handlers. Handlers are explained in more detail in chapter **??**.

File systems are special handlers that organize the contents of data volumes such as hard disks, floppies or CD-Roms in the form of files and directories. File systems interpret paths (see chapter **??**) in order to locate objects such as files and directories on storage media. Thus, every file system is a handler — i.e. the Fast File System (FFS) is a handler, discussed in section **??** — but not every handler implements a file system. The console window, for example, is provided by the CON-Handler in the form of the `CON` device, even though it surely does not organize files. Section **??** provides more details on this particular handler.

2.13 The AmigaDOS Shell

The AmigaDOS Shell is a command line interpreter and implements a (basic) programming language through which the user can communicate with the system. Historically, the Shell was also called the *CLI* for *Command Line Interface*, and it is the primary user interface of AmigaDOS. Unlike the graphical user interface, the Workbench, it is purely text based and available even without a boot medium.

The Shell reads lines from the console, which is a handler of its own, and interprets them as commands, potentially along with additional arguments to them. Commands are binary executable files (see chapter **??** for their structure) that are contained in a special directory of an AmigaDOS installation, though some elementary commands are built into the Shell and do not require access to a medium.

The Shell is not restricted to reading commands from the console. Any other handler can serve as source as well, for example by providing a *Shell Script* or *Batch File* located on a partition operated by the Fast File System (FFS), the Amiga ROM file system. The most important Shell script is the *Startup-Sequence* which is interpreted by the Shell when booting AmigaOs. Chapter **??** provides more details on the AmigaDOS Shell.

AmigaDOS is not limited to its own Shell, which is also called the Boot Shell; even though this feature is rarely used, it is possible to install additional alternative shells. The AmigaDOS interface to shells is specified in section **??**.

Chapter 3

Date and Time

Due to its history, AmigaOs uses two incompatible representations of date and time. The `timer.device` represents a date as the number of seconds and microseconds since January 1st 1978. As AmigaDOS is based on Tripos as an independently developed operating system, the *dos.library* uses a different representation as `DateStamp` structure defined in `dos/dos.h`:

```
struct DateStamp {
    LONG  ds_Days;
    LONG  ds_Minute;
    LONG  ds_Tick;
};
```

The elements of this structure have the following meaning:

`ds_Days` counts the number of days since January 1st 1978. It includes intercalary days added approximately all four years at the end of February.

`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. The number of ticks per second is defined as `TICKS_PER_SECOND` in `dos/dos.h`. By dividing the number of ticks by the above constants, one can derive the number of seconds since the start of the minute. Leap seconds that are added from time to time cannot be represented by AmigaDOS, instead the clock then requires manual adjustment.

> *Ticking 50 Times a Second* A system "tick" is always $1/50^{th}$ of a second, regardless whether the system is an NTSC or PAL system. AmigaDOS detects the clock basis during setup and will scale times appropriately such that the definition of the "tick" is independent of the clocking of the system or the monitor refresh frequency.

The system date, or rather the functions to convert between the Amiga `timer.device` representation and the AmigaDOS `DateStamp` representation are currently not able to handle dates after 31th December 2045.

There is no function in AmigaDOS to set the date, this needs to be done through the `timer.device` with the `TR_SETSYSTIME` command. The Kickstart, during bootstrap, takes the current time from the real time clock, if it is present. If no real time clock is present, then all elements of the `rn_Time` structure in the `RootNode` structure (see section **??**) of the *dos.library* remain 0, corresponding to a system date of January 1st 1978. The boot file system shall check this condition and provide a better approximation in such a case. The FFS will use in this case the creation time of the boot volume recorded in the root block, see section **??** and adjusts the system time then through the `timer.device`. Section **??** contains a more detailed description of the boot process.

3.1 Elementary Time and Date Functions

The functions in this section obtain the current system time, compare two times, or delay the system for a given time. They represent times — and dates if appropriate — in the `DateStamp` structure as a triple of days, minutes and ticks.

3.1.1 Obtaining the Time and Date

The `DateStamp()` function obtains the current date and time from AmigaDOS:

```
ds = DateStamp( ds );
D0              D1

struct DateStamp *DateStamp(struct DateStamp *)
```

This function retrieves the current system time and fills it into a `DateStamp` structure pointed to by `ds`. It also returns a pointer to the structure passed in. This function cannot fail.

Unlike many other *dos.library* functions, there is no requirement to align `ds` to a long-word boundary.

3.1.2 Comparing two Times and Dates

The `CompareDates()` function compares two dates as given by `DateStamp` structures and returns an indicator which of the dates is earlier.

```
result = CompareDates(date1,date2) /* since V36 */
D0                    D1     D2

LONG CompareDates(struct DateStamp *,struct DateStamp *)
```

This function takes two pointers to `DateStamp` structures as `date1` and `date2` and returns a negative number if `date1` is later than `date2`, a positive number if `date2` is later than `date1`, or 0 if the two dates are identical.

This function does not check the dates for validity, and it also assumes that difference between the days does not exceed 2^{31} days. Note that the logic of this function is different from `strcmp()` and related functions of ANSI-C which return a positive number if its first argument is larger than its second.

3.1.3 Delaying Program Execution

The `Delay()` function delays the execution of the calling process by a specific a number of ticks.

```
Delay( ticks )
       D1

void Delay(ULONG)
```

This function suspends execution of the calling process by `ticks` AmigaDOS ticks. The delay is system-friendly and does not burn CPU cycles; instead, the process is suspended from the CPU the indicated amount of time, making it available to other processes. Thus, this function is the preferred way of delaying program execution. A tick is $1/50^{\text{th}}$ of a second.

AmigaDOS variants below version 36 could not handle delays of 0 ticks appropriately, thus passing a 0 argument should be avoided.

3.2 Conversion Into and From Strings

Functions in this section convert date and time between the (binary) AmigaDOS representation and a human-readable string. The functions in this section are patched by the *locale.library* once it is loaded. The *dos.library* then also offers and recognizes localized strings of the corresponding locale, including four-digit representation of the year.

Both the input and output of these functions are kept in the `DateTime` structure that is defined in `dos/datetime.h` and reads as follows:

```
struct DateTime {
        struct DateStamp dat_Stamp;
        UBYTE   dat_Format;
        UBYTE   dat_Flags;
        UBYTE   *dat_StrDay;
        UBYTE   *dat_StrDate;
        UBYTE   *dat_StrTime;
};
```

`dat_Stamp` contains the input or output date represented as a `DateStamp` structure.

`dat_Format` defines the format of the date string and the order in which days, months and years appear within the string. The following formats are available, all defined in `dos/datetime.h`:

Table 3.1: Date Formatting Options

Format Definition	Description
`FORMAT_DOS`	The AmigaDOS default format
`FORMAT_INT`	International (ISO) format
`FORMAT_USA`	USA date format
`FORMAT_CDN`	Canadian and European format
`FORMAT_DEF`	The format defined by the locale[1]

`FORMAT_DOS` represents the date as day of the month in two digits, followed by the month as three-letter abbreviation, followed by a two-digit year counting from the start of the century. An example of this formatting is `30-Sep-23`.

`FORMAT_INT` starts with a two-digit year, followed by the month represented as two digits starting from 01 for January, followed by two digits for the day of the month. An example of such a date is `23-09-30`.

`FORMAT_USA` places the month first, encoded as two numerical digits, followed by two digits of the day of the month, followed by two digits of the year. An example of this formatting is `09-30-23`.

`FORMAT_CDN` follows the European and Canadian convention and places the day of the month first, followed by the month represented as two numerical digits, followed by the year as two digits.

`FORMAT_DEF` uses the format defined by the locale settings of the system if the *locale.library* is installed. Otherwise, it falls back to `FORMAT_DOS`. `FORMAT_DEF` will, depending on the locale, also use four-digit years and thus should be the preferred output format.

Unfortunately, it seems that the current NDK does not seem to define this format, yet it is properly handled. Its definition should therefore be performed manually as such:

```
#ifndef FORMAT_DEF
# define FORMAT_DEF 4
#endif
```

[1] Unfortunately not defined in the official includes, see note below.

dat_Flags defines additional flags that control the conversion process. They are also defined in dos/datetime.h:

Table 3.2: Date Conversion Flags

Flag	Description
DTF_SUBST	Substitute dates by relative description if possible
DTF_FUTURE	Reference direction for relative dates is to the future

The include file dos/datetime.h define in addition also bit numbers for the above flags that start with DTB instead of DTF. The meaning of these flags are as follows:

DTF_SUBST allows, if set, the conversion to substitute dates nearby today's date by descriptions relative to today. This flag is only honored when converting a time and date in AmigaDOS representation to human-readable strings, and is for example used by the List command. In particular, the following substitutions are made:

If the date provided is identical to the system date, the output date is set to "Today", or a corresponding localized string if the *locale.library* is loaded.

If the date is one day later than the current system date, the output date is set to "Tomorrow", or to an appropriate localized version of this string.

If the date is one day before the current system date, the output date is set to "Yesterday", or a localized version of this string.

If the date is in the past week, the function substitutes it by the name of the day of the week, e.g. "Saturday", or its localized version.

A date in the future is substituted by "Future", or its localized version.

DTF_FUTURE is only only honored when converting a string to the AmigaDOS representation, that is into DateStamp structure. It indicates whether weekdays such as "Monday" are interpreted as dates in the past, i.e. "last Monday", or as dates in the future, i.e. "next Monday". If the flag is cleared, weekdays are interpreted as being in the past, as the DateToStr() function would generate them. If the flag is set, weekdays are assumed as references to the future.

All following elements are string buffers that are either parsed when converting from human-readable strings, or filled by the *dos.library* when converting to strings. In the latter case, the output buffers shall be at least LEN_DATSTRING bytes long.

dat_StrDay: This buffer is only used when converting DateStamps to strings, and — if present — is then filled by the week of the day, e.g. "Saturday".

dat_StrDate: This element points to a buffer that is either filled with the human-readable date, or is input to the conversion then containing a human-readable date. The buffer is formatted, or expected to be formatted according to dat_Format and dat_Flags.

dat_StrTime: This element points to a buffer that is either filled with a human-readable time, or is the input time to be converted. AmigaDOS expects and provides here a 24h clock, hours, minutes and seconds in this order, separated by colons, e.g. 21:47:16. If this element is NULL, then the time is not converted.

3.2.1 Converting a Time and Date to a String

The DateToStr() function converts a date and time into a human readable string. The date and time, as well as formatting instructions are given by a DateTime structure.

```
success = DateToStr( datetime ) /* since V36 */
D0                       D1

BOOL DateToStr(struct DateTime *)
```

This function takes the date and time in the AmigaDOS binary representation contained in dat_Stamp of the passed in DateTime structure introduced in section **??** and converts it into human readable strings. The elements of this structure shall be populated as follows:

dat_Stamp shall be initialized to the date and time to be converted.

dat_Format defines the format of the date string to create. It shall be a value from table **??**.

dat_Flags defines additional flags that control the conversion process. This function only honors the DTF_SUBST flag which indicates that DateToStr() is supposed to represent the date relative to the current system date if possible. That is, if possible, the date is represented as "Today", "Tomorrow", "Yesterday" or a day of the week. The latter always correspond to past days, e.g. "Friday" means past Friday, not a day in the future.

dat_StrDay: If this pointer is non-NULL, it shall point to a string buffer at least LEN_DATSTRING bytes large into which the day of the week is filled, e.g. "Saturday".

dat_StrDate: If this pointer is non-NULL, it shall point to a string buffer at least LEN_DATSTRING bytes large; this constant is defined in dos/datetime.h. This buffer will then be filled by a description for the date according to the format selected by dat_Format and dat_Flags.

dat_StrTime: This buffer, if the pointer is non-NULL, is filled by the time of the day, using a 24h clock. The format is always hours, minutes, seconds, separated by colons.

This function is patched by the *locale.library* once it is loaded, and then replaces the English output by the corresponding localized output.

The function returns 0 on error; the only source of error here is if dat_Stamp is invalid, e.g. the number of minutes is larger than 60×24 or the number of ticks is larger than 50×60. This makes this function probably unsuitable to handle leap seconds correctly. This function does not touch IoErr(), even in case of failure.

The FormatDate() function of the *locale.library* is more powerful than this function and should probably be preferred if this library is available.

3.2.2 Convert a String to a Date and Time

The StrToDate() function converts a date and time from a human-readable string to its binary AmigaDOS representation.

```
success = StrToDate( datetime ) /* since V36 */
D0                       D1

BOOL StrToDate( struct DateTime * )
```

This function takes a DateTime structure as defined in section **??** and converts the date and/or time strings in this structure to a DateStamp structure in dat_Stamp. In particular, the elements of the DateTime shall be initialized as follows:

dat_DateTime provides a default time and date that are partially or fully overwritten by output of the conversion process from dat_StrDate and dat_StrTime. In other words, this element provides the result of this function.

dat_Format shall be initialized by the format that is used by the input date. Table **??** lists the available input formats. In particular, the ROM code within the *dos.library* only accepts two-digit years and interprets the anything between 78 and 99 as 1978 to 1999, and years between 00 and 45 as 2000 to 2045. It refuses all other numbers. However, StrToDate() is patched by the *locale.library* whose replacement implementation also accepts four-digit years.

dat_Flags shall be initialized by a combination of the flags from table **??**. As StrToDate() always accepts relative dates such as "Yesterday", the DTF_SUBST flag is ignored and only DTF_FUTURE is

honored. This flag indicates whether weekdays are considered to correspond to a date in the past or in the future.

dat_StrDay is ignored by this function. If a relative date given by a day of a week is to be converted, this weekday goes directly into dat_StrDate.

dat_StrDate, if it is non-NULL, points to a string describing the date, in the format according to dat_Format. If this string is not given, ds_Days of the dat_Stamp passed in remains unaltered.

dat_StrTime, if it is non-NULL, points to a human-readable string describing the time of the day. This time shall be formatted as a 24h clock, in the order hours, minutes and seconds, each separated by colon. If this pointer is NULL, then ds_Minute and ds_Ticks remain unchanged from the time passed in.

This function returns non-zero on success, and 0 on error. It does not set IoErr() in case of error. Possible errors include ill-formatted input strings the function cannot interpret.

Also note that this function is patched by the *locale.library* once loaded. It adds date and time conventions according to the current locale when setting dat_Format to FORMAT_DEF.

The ParseDate() function of the *locale.library* is more powerful than this function and should probably be preferred if this library is available.

Chapter 4

Paths and File Names

AmigaDOS organizes data on storage media such as floppies or harddisks in the form of files and directories. However, the files AmigaDOS manages are not limited to data units stored on media carriers, also denoted as *volumes*. The console window and the serial and parallel ports are also represented as files, and can be accessed through the same interfaces as files on disks. The latter are called *interactive files*, to distinguish them from static data stored on media that only change if explicitly modified.

Files and directories are identified by *paths*, through which their content can be reached, and from which AmigaDOS also locates a process that moderates access to them. Such processes are called *handlers*, or, in case of non-interactive files organized on a data carrier, *file systems*. The latter organize storage on disk, records information where on disk a file is located, and finds files given a path. Handlers provide file-like access to system resources such as the parallel or serial port. The *dos.library* does not interact with files or directories directly, but delegates their administration to handlers and file systems.

A *path* is broken up into two parts: An optional device, volume or assign name terminated by a colon (":"), followed by a string that allows the handler to locate the file and/or defines its properties.

The first part, if present, is interpreted by the *dos.library* itself. This part is the name of a handler (or file system), responsible for a physical device or a hard disk partition (see **??**), or the name of a particular *volume* or medium inserted in a drive (see **??**), or a logical volume, denoted as *assign*. The latter type is best understood as a shortcut to a directory (or even multiple directories) located on some volume(s) known to AmigaDOS. Handlers, volumes and assigns are administrated in the AmigaDOS *device list*. This list is discussed in depth in chapter **??**.

The second part of a path, or its only part, is interpreted by the handler or file system; the *dos.library* uses the first part to find the responsible handler, or the current directory of the calling process if the first part is missing.

4.1 Case Sensitivity and Character Encoding

Device, volume or assign names are always case-insensitive. The *dos.library* uses the service of the currently loaded locale to compare the first part of a path with their names, and it assumes by default the ISO-Latin 1 encoding. Whether all other components of a path are case sensitive and how directory and file names are compared to the names of file system objects is dependent on the file system. While file systems *should* be case-insensitive, some variants of the Fast File System do not handle case-insensitive comparisons correctly on non-ASCII characters, specifically ISO-Latin code points whose most-significant bit is set, see section **??** for details. These variants of the FFS should be avoided and the "international" variants should be preferred, see table **??** in section **??**. While the string comparison performed by the library and the string comparison of the international variants of the FFS as implied by the hashing algorithm of section **??** agree for printable

ISO-Latin characters, this is *not necessarily* the case for other encodings or components containing non-printable characters.

In general, AmigaDOS support functions such as the pattern matcher functions discussed in chapter **??** also assume that file systems are case-insensitive, and use the ISO-Latin character set. Thus, what a file system considers an identical file name can, actually, be something different from what the *dos.library* considers a matching name. Code points from the control set of ISO-Latin, i.e. codes between `0x01` to `0x1f` or `0x7f` to `0x9f` should be avoided[1], and the code point `0x00` ASCII `NUL` shall not be used at all as it terminates C strings and therefore cannot be part of a file name anyhow[2].

The remaining code points from the control sets of ISO-Latin are non-printable characters and multiple modules of AmigaDOS will behave erratically if component names containing them are encountered. In particular, the pattern matcher of chapter **??** uses them as tokens for its wildcards and misbehaves if they are encountered as input. While the Workbench may attempt to display such control characters if the icon font contains glyphs at their code points, such codes form control sequences of the console and thus, in general, do not result in useful console output if printed by the Shell, regardless of the console font.

Other than the colon (“`:`”) and the slash (“`/`”), names of file system objects on FFS volumes may contain all printable ISO-Latin characters, that is, code points between `0x20` and `0x7e` and `0xa0` to `0xff`. Some file systems can, however, impose additional restrictions that origin from their native operating system. Regardless of the file system, some characters should be nevertheless avoided as the AmigaDOS Shell and some functions of the *dos.library* assign special meanings to them; paths containing such characters are hard to reach from the AmigaDOS Shell:

`*` The asterisk as stand-alone file name represents the current console of the calling process, see also section **??**. It is *in addition* also the escape character of the AmigaDOS Shell that becomes active within double quotes, see section **??**. While it can always escaped by another asterisk, this can trigger hard to find defects in Shells scripts.

`>,<,|` While the angle brackets and the vertical bar are allowed within file names, they are also operators of the AmigaDOS Shell that redirect the input, output or error stream of a command, or form compound commands as defined in sections **??** and **??**. While they can be used in file names, such names require quoting within the Shell as they could be misinterpreted as syntax elements otherwise.

`SPC` The ASCII blank space at code point `0x20` can be used within file names, though also separates arguments and commands in the AmigaDOS Shell and therefore requires proper quoting of paths within Shell scripts and within the Shell.

`#,?,[,],’,~,(,),%` These characters are syntax elements of the pattern matcher described in chapter **??**. The pattern matcher syntax also defines the apostrophe (“`’`”) as escape character and therefore allows, in principle, to use them in paths, though there is unfortunately no method to identify whether a particular shell command passes its arguments through the pattern matcher, thus requiring to escape them, or uses its arguments literally. Thus, again, for the sake of simplicity, these characters should be better avoided.

The Workbench is less critical in this regard and allows all characters in the above list without requiring escaping or quoting — the only non-working name for a Workbench disk object is the string consisting of a single asterisk (“`*`”) as the *dos.library* and not the Shell assigns a meaning to it.

> *Avoid Odd File Names* While AmigaDOS provides mechanisms to include functional characters in file names such as quoting and escaping, characters forming syntax elements of the Shell or the pattern matcher should be avoided as they can trigger hard to find defects in Shell scripts. Characters from the non-printable C0 and C1 control set of ISO-Latin 1 shall not be used at all, even if they seem to display correctly on the Workbench; they may trigger side effects of the pattern matcher.

[1] Unfortunately, the file systems in the Amiga Kickstart are currently inconsistent on which characters they accept. The FFS does not accept any control characters, the RAM-Handler (probably erroneously) accepts characters in the C1 control set from `0x80` to `0x9f`.

[2] To be very precise, it actually *could* be as this character does not have a special meaning in a BSTR, though any type of C interface as for example the one of the *dos.library* would be heavily confused by such a file name.

4.2 Maximum Path Length

The maximal length of a path that can be safely passed as argument to most functions of the *dos.library* is 255 characters. This is because the library has to convert paths internally to BSTRs to communicate them to handlers and file systems, and the lengths of BSTRs are expressed in bytes. The functions from section **??** and the `ReadLink()` function of section **??** are some notable exceptions. Section **??** demonstrates how to work around this restriction by operating with pairs of paths and directory locations.

How large the name of a file or directory can be is a matter of the file system itself. The Fast File System includes variants that limit names to 30, 54 or 106 characters. The latter is the upper limit that is imposed by the `FileInfoBlock` structure, see section **??**.

File systems typically do not report errors if the maximum file or directory name length 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 file representing a Workbench icon without ever reporting this, upon which the Workbench would not be able to identify the file as an icon. The *icon.library* and *workbench.library* of AmigaOs are aware of this problem, avoid such file names and double check created objects for correct names.

4.3 Devices, Volumes and Assigns

The first part of a path, up to the colon, identifies the device, the volume or the assign a file or directory is located in. In addition to the 255 character limit of path names and the size limit of file and directory names, the *dos.library* also imposes a 30 character limit for device, volume and assign names.

> *30 Characters Max!* For legacy reasons, device, volume and assign names can be at most 30 characters long. The AmigaDOS functions that locate handlers, most notably `GetDeviceProc()`, are not able to handle longer names, and the `Assign` command creating assigns suffers from the same restriction. This 30 character limit does not hold for file names or paths in general as the corresponding AmigaDOS components have been augmented. However, additional limits also arise, unfortunately, for paths.

4.3.1 Devices

A *device name* identifies the handler or file system directly. Handlers are typically responsible for particular hardware units, or partitions on such units, 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 in it.

Table **??** lists all device names AmigaDOS creates itself even without a boot volume available. They can be assumed to be present any time as they are created by the AmigaDOS ROM components:

Table 4.1: 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 system

If more than one floppy drive is 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 the HDToolBox. The general convention is to assign hard disk partitions the device names `DH0` and following. As for floppy disks, partitions *also* have a volume name, see section **??**, which should be different from their device names. The Workbench shows the latter, and *not* the device name, on its screen.

The devices `SER`, `PAR`, `PRT` and `PIPE` are created by the Kickstart ROM, but the corresponding handlers are disk-based and loaded as soon as the corresponding device is needed. The addition of `PIPE` to this list is relatively recent, AmigaDOS V47 (Kickstart 3.2) includes it in the ROM-mounted devices as the Boot Shell requires it. Its handler is neither included in the Kickstart.

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

Table 4.2: 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 the *dos.library* itself. Any data written into it vanishes completely, and any attempt to read data from it results in an end-of-file condition. It does not take nor allow any file name behind its name.

The `*`, if used as complete path name without a trailing colon and without a file name, identifies the current console of the process, if such a console exists. Any data output to the file named `*` will be printed on the console. Reading from `*` will wait for the user to input data on the console, and will return such data. If no console exists, for example for processes run from the Workbench, then AmigaDOS versions 36 and later fall back to opening `NIL:` which absorbs any output. Earlier versions created an error in such a case.

> *Not a wildcard!* Unlike other operating systems, the asterisk `*` is *not* a wildcard under AmigaDOS. It rather identifies the current console of a process. It is also used as escape character in AmigaDOS Shell scripts, see chapter **??**.

The `CONSOLE` device is the default console of the process. This special device name exists since AmigaDOS version 36. Unlike `*`, but like any other device name, it shall be followed by a colon. It also takes an optional name behind the colon that identifies a job, and allows the console to block the input and output of all but the active foreground job. While the AmigaDOS ROM CON-Handler does not provide job control features, some third party consoles do.

> *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 added to the system by the *Mount* command, see chapter **??**. Device names for custom mount handlers and file systems can be chosen freely as long as they do not conflict with system mounted devices, the special names listed in table **??** or the system-defined assigns in section **??**.

4.3.2 Volumes

A *volume name* identifies a particular data carrier within a physical drive. It can 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. Partitions on a harddisk also have a volume name by which they can be identified.

4.3.3 Assigns

An *assign* or *logical volume* identifies one or multiple directories 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 relocated, 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 used interchangeably with device or volume names and thus form logical volumes within partitions, disks, or even across multiple volumes.

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

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 included in 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 and multi-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 points to; whenever a file or path within the assign is requested, any volume of the name given by the assign target will satisfy the request. However, this also implies that the target of the assign is not necessarily consistent, i.e. if the assign is accessed again at a later time, another volume of the same name, but potentially different content will also satisfy the request.

Late binding assigns are a compromise between regular assigns and non-binding assigns. AmigaDOS initially only stores a symbolic path for such a late binding assign, but when the assign is accessed the first time, the assign is converted to a regular assign and then binds to the accessed volume and directory from this point on.

Table **??** 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 binding assign. `ENVARC` was added in version 36 of AmigaDOS.

Table 4.3: 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 (pseudo-)assign is handled by the *dos.library* internally and is not part of the *device list*, (see chapter **??**), it was also added in version 36 of AmigaDOS:

Table 4.4: System Defined Assigns

Assign Name	Description
`PROGDIR`	Location of the executable

PROGDIR is the directory within which the currently executed process finds the command or application code it was created from. This allows applications to locate program related data contained in the same directory as their main program code. PROGDIR does not exist[3] in case an executable was not loaded from disk, probably because it was either taken from ROM or was made resident before and located on the list of resident segments. More on resident segments 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 **??** lists some of them.

Table 4.5: Assigns Created 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
CLASSES	Boopsi GUI components

The assigns ENV, REXX, LOCALE and CLASSES became part of AmigaDOS in releases 36, 38 and 39, respectively. Additional assigns can always be made with the Assign command or the AmigaDOS functions discussed in section **??**.

4.4 Relative and Absolute Paths

As introduced in above, a path consists of an optional device, volume or assign name and a colon (“:”), followed by a string describing a file or directory within the accessed volume, device or assign. If the colon is present, such a path is said to be an *absolute path* because it identifies a location within a logical or physical volume relative to the topmost or *root directory* of the volume. If the device name and the colon are not present, such a path is called a *relative path* because it corresponds to a location in the file system relative to the *current directory* of the process using the path.

If only a colon is present but the device, volume or assign name is omitted, the path is still an absolute path and is relative to the root directory of volume that contains the current directory of the running processing. Details on processes and their properties are provided in chapter **??**.

If neither a colon nor a device, volume or assign name is present, the path is a relative path; it is relative to the current directory of the process using it.

The path is forwarded to the handler or file system identified by either the current directory or the device, volume or assign name. It is within the responsibility of the file system to interpret this path and locate the file or directory within the volume it manages, or in case of handlers, to configure an interface to the outside world according to this path. For example, while the Fast File System (FFS) — the Amiga ROM file system — interprets this path as a location within the file system directory hierarchy, the CON-Handler reads it as a specification of the dimensions and configuration of the window it is supposed to open.

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 functions workable. It is therefore advisable for file system implementers to follow the conventions of section **??** how to interpret path names.

[3]Unfortunately, attempting to access a file relative to PROGDIR if it does not exist, e.g. from within resident executables, will create a requester to insert a volume of this name. This is likely a defect as this requester is not particularly instructive.

4.5 Flat vs. Hierarchical 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 organization is clearly burdensome as files will be hard to find and hard to identify.

For this reason, all file systems provided by AmigaOs are *hierarchical* and organize files in nested *directories*, where each directory contains more files or 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. Each directory or file name in the path is denoted as a *component* in the following. AmigaDOS therefore separates components by forward slashes; this is different from some other operating system that use the backslash as component separator.

4.6 Locating Files or Directories

When attempting to locate a particular file or directory, the *dos.library* first checks if the path contains a colon ("`:`") and hence is an absolute path. If so, it locates the handler or file system responsible from the name upfront the colon and then provides the entire path to this handler.

Otherwise, it uses the *current directory* of the calling process to locate a file system responsible for the interpretation of the path name. If the current directory is `ZERO` (see section **??**), it uses the *default file system* of the process, which is, unless changed, the file system that booted the system.

The *dos.library* then provides to the file system the path, and a directory from which to start locating the requested object. Interpretation of the path continues then by the file system, outside of the *dos.library*. The following paragraph describes a recommended algorithm all AmigaDOS file system should follow:

To locate a file, the file system works iteratively through the path, component by component: A single isolated "`/`" without a preceding 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 file system to enter the directory given by the component, and to continue searching there. Scanning terminates when the file system reaches the last component. The file or directory to find is then given by the last component reached during the scan.

As interpreting relative path 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 preceding 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`".

4.6.1 Open a File From an Overlong File Name

The following code demonstrates both how to iterate through a path, and also shows how to work around the 255 character limit of paths the *dos.library* can process. It implements a lookalike of the `Open()` function introduced in section **??** that also works with paths longer than 255 characters, though requires that the handler addressed by the path is a file system.

```
BPTR OpenFromLongName(UBYTE *name,LONG mode)
{
  LONG pos  = 0;
  BPTR file = 0;
  BPTR old  = -1L; /* Never a valid lock */
  /* Long enough for a component and a device name */
  UBYTE buffer[108+32];

  do {
    pos = SplitName(name,'/',buffer,pos,sizeof(buffer));
    if (pos < 0) {
      /* No separator found, call now Open() */
      file = Open(buffer,mode);
      break;
    } else {
      BPTR lock;
      /* Lock the partial path so far and abort on error.
      ** If two slashes are next to each other,
      ** go to the parent directory.
      */
      if (!(lock = Lock(*buffer?buffer:"/",SHARED_LOCK)))
        break;

      /* Rotate directories */
      lock = CurrentDir(lock);

      /* Unlock previous directory,
      ** keep the old directory
      */
      if (old >= 0) {
        UnLock(lock);
      } else {
        old = lock;
      }
    }
  } while(1);

  /* Restore the current directory if any.
  ** None of the functions touch IoErr().
  */
  if (old >= 0)
    UnLock(CurrentDir(old));

  /* Return the opened file */
  return file;
}
```

The `SplitName()` function copies a component of a path into a buffer, splitting it at the indicated separator. More details on this function are found in **??**. The `Lock()` and `UnLock()` functions obtain and release abstract descriptors of directories, they are discussed in chapter **??**. The `CurrentDir()` function changes the current directory of the calling process and returns the previous directory. More on this function in section **??**.

Chapter 5

Files

5.1 What are Files?

Files are streams or 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 can run into an *end of file condition* upon reaching it no further data can be read from them.

For files that represent an interface of the computer system with its environment, such as the console, the end of file condition depends on factors beyond the control of the system. If more data becomes available through the interface, e.g. as being typed into the console, additional data can be read from the stream. For files stored on a disk, the condition is triggered when the file pointer reaches the last byte of the file, the *end of file position*. It can be adjusted by writing additional bytes into the file, or by setting the file size.

5.2 Interactive vs. non-Interactive Files

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

Non-interactive files are stored on a data carrier whose contents only changes due to processes within the computer system itself. They also have a defined *file size*, which is the number of bytes between the start of the file and the end-of-file position, or short *EOF position*. It is possible to open the same file by two or more processes in parallel, and in such a case, the file content and its size can change unpredictably, depending on how AmigaOs schedules the processes accessing the file. Such situations should be avoided, and AmigaDOS provides mechanisms to request exclusive access to a file (see section **??** and chapter **??**), or even parts of a file (see section **??**).

Examples for non-interactive files are data on a disk, such as on a floppy or a harddisk. Such files have a name, possibly a path within a hierarchical file system, possibly a creation date, a file comment and 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, deleted or executed; this so-called *meta-information* is discussed in more detail in section **??**.

Interactive files depend on the interaction of the computer system with the outside world, and their contents can change due to such interactions. Interactive files do not have a well-defined file size as the number of bytes that can be read from them depends on events in the environment. An attempt to read from them or write to them can 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. Another example is the serial port, where read requests are satisfied by data arriving at the serial port and written bytes are transmitted out of the port. The parallel port is a third 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 can block indefinitely if the printer runs out of paper or is turned off.

5.3 Opening and Closing Files

To open a file, an absolute or relative path name needs to be provided. The `Open()` function uses this information to construct a *file handle* through which the contents of the file can be accessed or modified. Depending on how the file is opened, multiple processes may access the same file simultaneously, or may even alter the file simultaneously.

Once done with the file, file handles shall be released again with `Close()`. This not only returns system resources, it also makes files opened for exclusive access available to other processes, and ensures that all modifications are written back to the medium the file is located, or written out through the interface the file represents.

5.3.1 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* of the file to be opened, which is interpreted according to the rules given in chapter **??**. 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 result code can be retrieved from `IoErr()`; this function is introduced in section **??**. This result code is undefined on success. In case opening the file failed, `IoErr()` delivers one of the error codes from `dos/dos.h`; section **??** lists the system defined codes.

> *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 passed into `Open()`. It is the responsibility of the caller to split oversized paths and potentially walk through the directories manually if necessary, for example by a function similar to that presented in section **??**. Note that this strategy may not be suitable for interactive files or for handlers that follow conventions for path names that are different from the conventions described in section **??**.

The access mode shall be one of modes in table **??**, they are also defined in `dos/dos.h`:

Table 5.1: 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

The access mode `MODE_OLDFILE` attempts to find an existing file. If the file does not exist, the function fails. If the file exists and `Open()` succeeds, it can be read from or written to, and 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.

Under AmigaDOS version 36 and above, 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 in the middle of the path if they do not exist. 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 simultaneously, the order in which the writes are served is undefined and depends on how processes are scheduled.

For AmigaDOS versions 34 and below, MODE_READWRITE implements a mode that is almost the reverse of what newer releases provide under the same name. There, MODE_READWRITE requested exclusive access, but required the file to be existing already; it did not create new files. Due to these inconsistencies, MODE_READWRITE should probably be best avoided. If exclusive access without deleting existing file content is required, it is best to first obtain an exclusive lock, see section **??**, and then use this lock to create a file handle from it though OpenFromLock(), see section **??**.

Finally, the access mode MODE_NEWFILE creates a new file, potentially erasing an already existing file of the same name. 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 the error code ERROR_OBJECT_IN_USE.

> *No Wildcards* The Open() function, similar to most *dos.library* functions, does not attempt to resolve wildcards. That is, any character potentially reassembling a wildcard, 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 it hard to access such files from the Shell.

5.3.2 Closing Files

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

```
success = Close( file )
D0               D1

BOOL Close(BPTR)
```

The file argument is a BPTR to a file handle identifying the file to close. The return code indicates whether the file system or handler could successfully close the file; closing a file can fail, for example, if data is still buffered in file system internal structures and the target volume is not accessible anymore.

If the result code is DOSFALSE, an error code can be obtained through IoErr() described in section **??**. On success, IoErr() will not be altered. Under AmigaDOS version 34 and below, this function does not return a result code and the contents of register d0 cannot be depended upon.

File handles that had been obtained through Input(), Output() or ErrorOutput() should, in general, not be closed as they had been created by the environment launching the process, e.g. the shell or the Workbench. These handles will automatically be closed when the running program terminates. Attempting to close the same file twice will most likely result in a crash.

If closing a file fails not much can be done unfortunately and no general advice is possible how to handle this situation. At the time Close() returns with an error, an error requester has already been shown to the user, unless it has been explicitly disabled by setting the pr_WindowPtr to -1, see chapter **??**. Thus, retrying Close() immediately again does not help to resolve the problem. If Close() fails, the file handle file remains valid and available to the caller.

Attempting to close the ZERO file handle under AmigaDOS version 36 or later returns DOSFALSE immediately[1]. Under earlier versions of AmigaDOS, attempting to close ZERO caused a crash.

[1] Returning DOSTRUE would have probably been a wiser decision as there is nothing to close in first place then.

5.4 Unbuffered Input and Output

The functions described in this section read bytes from or write bytes to opened files. These functions are *unbuffered*, that is, any request goes directly to the handler. Since a request necessarily performs a task switch from the caller to the handler managing the file, these functions are inefficient for small amounts of data and should then 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 **??**.

5.4.1 Reading Data

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

```
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 regular C pointer, not a BPTR.

The return code `actualLength` is the amount of bytes actually read, which may be less bytes than requested, 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 (but not more) data than requested by the `length` argument, either because the EOF position has been reached (see section **??**) for non-interactive files, or because the interactive source is depleted. Note that for interactive files, the function may block indefinitely until sufficient data or a fraction of the requested data becomes available. The `WaitForChar()` function introduced in section **??** may be used to probe interactive file handles for the availability of data.

5.4.2 Writing Data

The `Write()` function 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. The number of bytes written can be less than the number of bytes requested, and in extreme cases, can 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, e.g. the `PRT` device can block until an out-of-paper condition is resolved. Unlike reading, there is unfortunately no function within AmigaDOS that allows to determine upfront whether a particular handler will block on an attempt to output data.

5.4.3 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 within 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 value, defined in `dos/dos.h`:

Table 5.2: 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 (interactive) handlers may nevertheless assign to this function a particular meaning — see the handler documentation for details. The only way how to find out whether `Seek()` is supported is to call it and check its return code.

If `mode` is `OFFSET_BEGINNING`, then the new file pointer is placed `position` bytes from the start of the file. For that, the `position` shall be non-negative and smaller or equal than the size of the file.

If `mode` is `OFFSET_CURRENT`, then `position` is added to the current 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 the size of the file. This, in particular, requires that `position` shall be non-positive, and larger or equal than the negative file size.

The `Seek()` function returns the (absolute) file pointer relative to the beginning of the file 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. Unfortunately, it is not too uncommon that handlers that *do not* implement `Seek()` erroneously return 0 instead of -1 to report this problem and then set `IoErr()` to `ERROR_ACTION_NOT_KNOWN`.

> *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 succeeds 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 handle internal buffer. That is, it can be safely used by buffered and unbuffered functions.

5.4.4 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, and it is generally limited to file systems.

```
newsize = SetFileSize(fh, offset, mode) /* since V36 */
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. The file pointer is only adjusted if its position lies beyond the new end of file position.

The mode argument shall be selected from table **??**. In particular, it is interpreted as follows:

If mode is OFFSET_BEGINNING, then the file size is set to the value of offset, irrespectively 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 possibly truncated or 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 file pointer of any file handle opened on the affected file is, after a potential truncation, beyond the new end-of-file, it is clamped to the end-of-file. File pointers remain unaltered otherwise.

If the file is enlarged, the content of the file beyond the previous end-of-file position is undetermined. Some handlers set these bytes to zero, but AmigaDOS does not enforce this.

On success, the return value newsize is the size of the file after the adjustment, i.e. the updated file size, and IoErr() is set to 0. On error, this function returns -1 and an error code from dos/dos.h through IoErr(). Unfortunately, handlers that do not implement this function can erroneously also return 0, and thus callers should also check IoErr() if 0 is the primary result code. Handlers that do not implement this function will set IoErr() to ERROR_ACTION_NOT_KNOWN.

> *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, and then perform a call to SetFileSize() with the mode set to OFFSET_CURRENT. However, whether this strategy succeeds is file system dependent.

5.5 Interactive File and Handler Support

As introduced in **??**, AmigaDOS distinguishes between non-interactive files managed by file systems and interactive files that interact with the outside world. File systems create non-interactive files; all other handlers create interactive or non-interactive files, depending on the nature of the handler.

The functions in this section test whether a given file is interactive, or determine from a path whether a particular handler is a file system and therefore generates non-interactive files. Additional support functions for interactive files test for the availability of input data or change how they process input and how their input buffer operates.

5.5.1 Test whether an File Handle is Interactive

A file can be either interactive, in which case attempts to read or write data to the file may block indefinitely and the contents and state of the file depends on conditions of the outside world, 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 a non-zero result code 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()`.

The Port-Handler and the devices `PRT`, `SER` and `PAR` managed by it generates interactive file handles, and so does the CON-Handler and the `CON`, `RAW` and `AUX` devices it implements. The Queue-Handler and its `PIPE` device is another example of a handler whose files are interactive. The FFS and the RAM-Handler are file systems and thus create non-interactive files.

While reading from interactive files may block indefinitely, the `WaitForChar()` function introduced in section **??** may be used to test for the availability of data. Interactive files typically do not support repositioning the file pointer through `Seek()` (see **??**) or changing the file size through `SetFileSize()`, see section **??**.

5.5.2 Test whether a Path addresses a Handler or File System

A handler that manages volumes and allows to access named files on them is a file system. Such handlers will create non-interactive files. The `IsFileSystem()` function determines the nature of a handler from a path.

```
result = IsFileSystem(name) /* since V36 */
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 for accessing the hypothetical object, regardless whether the path corresponds to an exiting object or not.

It is advisable to provide a path that identifies the handler uniquely, i.e. a device or volume name 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 non-zero in case the handler identified by the path is a file system, and as such allows access to files on volumes, and is able to examine directories on it. Otherwise, it returns `DOSFALSE`.

This function did not exist prior to version 36 of the *dos.library*. A possible workaround to check whether a candidate handler is a file system is to attempt to lock the root directory of the handler:

```
LONG IsFileSystem34(UBYTE *path)
{
  UBYTE *colon = strchr(path,':');

  if (colon && colon > path) {
    BPTR lock;
```

```
    /* A non-empty device name */
    colon[1] = '\0';
    UnLock(lock = Lock(path,SHARED_LOCK));
    if (lock == 0)
      return DOSFALSE;
  } else if (!strcmp(path,"*")) {
    /* The console is never a file system */
    return DOSFALSE;
  }
  return DOSTRUE;
}
```

This workaround should not be used in AmigaDOS version 36 as its *dos.library* uses a dedicated packet to identify file systems, see section **??**. However, a similar workaround is also used internally by the newer library for handlers that do not support `IsFileSystem()` natively. A side effect of this workaround, but also of `IsFileSystem()`, is that they will load and start the corresponding handler if it is not already running.

5.5.3 Test Interactive Files for Availability of Data

An issue of the `Read()` function is that it can block indefinitely on an interactive file if the user or the environment does not provide any input. The `WaitForChar()` tests for the availability of at least one byte on an interactive file for limited amount of time and returns either if data becomes available or if it runs out of time.

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

BOOL WaitForChar(BPTR, LONG)
```

This function waits for a maximum of `timeout` microseconds for the availability of at least one character on `file`. If data is already available, or becomes available within or before this time, the function returns a non-zero result. In such a case, a subsequent `Read()` is able to retrieve at least one byte from the stream without blocking. Otherwise, and also in case of an error, the function returns `DOSFALSE`.

This function does not remove any bytes from the stream, it only checks for the availability of bytes at the level of the handler. This function is neither aware of any bytes buffered in the file handle the buffered IO functions of section **??** would be able to read.

A secondary return code can be obtained from `IoErr()`. If the function returns `DOSFALSE`, then `IoErr()` returns 0 if no bytes became available and the handler was able to complete the function. On failure, `WaitForChar()` returns `DOSFALSE` and `IoErr()` returns an error code.

If `WaitForChar()` returns a non-zero code, then for some handlers, in particular for the CON-Handler, a secondary result code is provided through `IoErr()`. The CON-Handler leaves here the number of lines available in its line buffer. In particular, one *shall not* depend on `IoErr()` returning 0 on success.

This function requires an interactive file to work with, file systems will typically not implement this function as they do not block. File systems rather return `DOSFALSE` and set instead `IoErr()` to the error code `ERROR_ACTION_NOT_KNOWN`.

Under AmigaDOS prior to version 36, a timeout value of 0 does not function properly. This was fixed in release 36.

5.5.4 Setting the Console Buffer Mode

The SetMode() function sets the buffer mode of an interactive handler. It is typically used in conjunction with the graphical or serial console, i.e. the CON-Handler and the AUX-Handler, and there sets the input buffer mode of the console. Depending on this mode, the console either waits for an entire line to be completed to satisfy a read request, provides each individual key as input to programs, or uses the keystrokes to implement a line editor except for some control keys that are transmitted immediately. Section **??** contains further details on the console and its modes.

```
success = SetMode(fh, mode) /* since V36 */
D0                D1   D2

BOOL SetMode(BPTR, LONG)
```

This function sets the mode of the handler addressed by the FileHandle fh to the mode provided as second argument. The meaning of the modes is specific to the handler; however, this function is typically used in conjunction with both consoles provided by AmigaDOS, the graphical console of the CON and RAW device, and the serial console corresponding to the AUX device.

For the console(s), the interpretation of the mode argument is as follows:

Table 5.3: Console Modes

Buffer Mode	Description
0	Cooked mode
1	Raw mode
2	Medium mode (V47)
All other values	Reserved for future use

In the *cooked mode*, the console buffers entire lines, provides line-editing features, but only makes the input data available to clients when the user terminates the input with the RETURN key. The CON and AUX devices operate by default in this mode, but can be switched to any other buffer mode by this function.

In the *raw mode*, every single keystroke is made available immediately, including control sequences corresponding to all cursor and function keys. That implies, however, that line editing is not available and pressed keys are not echoed on the console. If echoing is desired, the application reading from the console needs to print out each typed key itself. This mode corresponds to the RAW device which is nothing but a console operating in this mode when opening it under this name.

In the *medium mode*, the console also buffers lines, but some keystrokes are directly transmitted without requiring the user to press the RETURN key. In specific, the key combinations consisting of the up- and down cursor keys as well as the TAB key are reported immediately to the caller through control sequences. These sequences are documented in section **??**. The Amiga Shell uses this mode to offer a history and provides by it TAB expansion of commands and command line arguments. No device name corresponds to this mode; instead, the Shell switches a regular CON: window to this mode in order to offer its services.

Both the CON-Handler and the AUX-Handler implement this function, supporting all three modes. However, there is — unless explicitly mounted by the user — no device name that corresponds to the *medium mode* and no device name that corresponds to an AUX: console in the *raw mode*.

The SetMode() function returns a Boolean success code, which is non-zero if the function could change the console mode, or 0 if it failed. In case of success, IoErr() is set to 1 if the console is attached to a window of the Amiga graphical user interface, or to 0 if the window is currently closed or the console is operating on top of some other device, such as the serial port. In case of failure, IoErr() returns an error code from dos/dos.h.

As an application, the following function attempts to read a keystroke from a given file handle, supposed to be connected to the console, without waiting for the RETURN key; it returns -1 on an error or otherwise the ISO-Latin code of the pressed key:

```
LONG getch(BPTR file)
{
  UBYTE c;
  LONG err = -1;

  /* switch to raw */
  if (SetMode(file,1)) {
   if (Read(file,&c,1) == 1)
    err = 0;
   /* back to cooked */
   SetMode(file,0);
  }

  return (err < 0)?err:c;
}
```

5.6 Buffered Input and Output

AmigaDOS also offers buffered input and output functions that stores data in an intermediate buffer. It then transfers data only in larger chunks between the buffer and the handler, minimizing the task switching overhead and offering better performance if data read or written in smaller units. All file handles of AmigaDOS allow buffered input and output, and are equipped with a file buffer of a default size of 204 bytes as soon as buffered input or output is requested. The `SetVBuf()` function in section **??** should be used to request larger buffers.

Buffered I/O functions had always been part of AmigaDOS, they were only available to BCPL programs through the Global Vector prior to version 36. In version 36, the *dos.library* was rewritten in assembler and C, and as part of this re-implementation, they became available through the *dos.library* interface as well.

> *Performance Improved* While buffered I/O functions of AmigaDOS 45 and below were based on single-byte functions and thus had a massive overhead, the functions in this section were redesigned in AmigaDOS 47 and now provide significantly better performance. Unfortunately, the default buffer size AmigaDOS uses is quite small and should be increased by `SetVBuf()`. A suggested buffer size is 4096 bytes which corresponds to a disk block of modern hard drives.

The standard input file handle of a shell command provided by `Input()` is unfortunately a special case as its input buffer contains the command line parameters where the `ReadArgs()` function specified in section **??** retrieves them. Thus, any buffered read from the input stream will receive the command line parameters and not the data from the actual stream. To release these buffers and synchronize the buffer of the input stream with the actual file contents, an `Flush()` would be necessary. Unfortunately, the input buffer containing the command line parameters is allocated manually following the algorithm in section **??**, and an `Flush()` is not necessarily functional if the original stream was unbuffered. Instead, the input stream first needs to be converted to a buffered stream, and then flushed. The following function pair prepares the input stream for buffered I/O:

```
FGetC(Input()); /* convert to buffered mode        */
Flush(Input()); /* release the cmd line parameters */
```

AmigaDOS provides buffered record-oriented read and write functions similar to ANSI-C `fread()` and `fwrite()`; they are introduced in the next two sections. They read or write one or multiple records, each of the same size. There is no advantage of setting the record size to a particular value, in effect they read or write the number of bytes given by the product of record size and record count, though return only the total

number of complete records that could be received or written. Thus, a block of bytes can be read or written by setting the record size (the `blocklen` argument in the following sections) to 1.

While most buffered I/O functions closely reassemble those of ANSI-C, there is one important difference: Unlike many ANSI-C functions, the buffered I/O functions of the *dos.library* retrieve the file handle as first and not as last parameter.

5.6.1 Buffered Read From a File

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) /* since V36 */
 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. Such an incomplete record is *not* included in `count`.

From AmigaOs 47 onward, `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 inspect the return code of `IoErr()` to learn if any error occurred when the number of records read is smaller than the number of records requested.

5.6.2 Buffered Write to a File

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) /* since V36 */
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. Such an incomplete record is *not* included in `count`.

From AmigaDOS 47 onward, `FWrite()` first checks whether the file handle internal buffer contains already some data. If so, the file handle internal 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 into 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()` afterwards to learn if any error occurred if the number of records written is smaller than the number of records passed in.

5.6.3 Buffered Write to the Output Stream

The `WriteChars()` writes an array of bytes buffered to the output stream.

```
count = WriteChars(buf, buflen) /* since V36 */
D0                 D1   D2

LONG WriteChars(STRPTR, LONG)
```

This function is equivalent to `FWrite(Output(),buf,1,buflen)`, that is, it writes `buflen` bytes from `buf` to the output stream of the calling process. The number of characters written is returned. Therefore, this function has similar quirks concerning error reporting as `FWrite()`: It does not set `IoErr()` consistently, namely only when the buffer is written to the stream. It neither returns -1 on an error[2]. It is therefore recommended to reset the error upfront with `SetIoErr(0)`.

5.6.4 Adjusting the Buffer

The `SetVBuf()` function sets the buffer mode and adjust the buffer size for buffered input/output functions such as `FRead()` or `FWrite()`. As the default buffer size is with 204 characters quite small, it is for many applications advisable to enlarge it by this function.

```
error = SetVBuf(fh, buff, type, size) /* since V39 */
D0              D1   D2    D3    D4

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

This function sets the size of the internal buffer of the file handle `fh` to `size` bytes. Sizes smaller than 204 bytes will be rounded up to 204 bytes[3]. 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 or another buffer is provided.

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

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

Table 5.4: 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, for this see the specification of `Flush()` in section **??**.

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.

[2]The information in [**?**] on returning -1 on error is incorrect.

[3]While the official autodocs and [**?**] reports a default buffer size of 208 characters, this information is not correct.

The function returns 0 on success, and non-zero on error, unlike most other functions of the *dos.library*. 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.

Even though this function is documented since AmigaDOS version 36, it was only implemented from version 39 onward. AmigaDOS prior version 39 always returned 0. Due to another defect present in all versions of AmigaDOS this function cannot be safely applied to file handles opened to the `NIL` pseudo-device. As a workaround, the caller should check first the `fh_Type` element of the file handle, and should not call this function if this element is `NULL`, see also section **??**.

5.6.5 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.

```
success = Flush(fh) /* since V36 */
D0              D1

LONG Flush(BPTR)
```

This function synchronizes the file pointer to the buffer, that is, if data were written to the buffer, the buffer will be pushed out to the file. If data were read from the file and non-read bytes remained in the buffer, such bytes are dropped and the function attempts to seek back to the position of the last read byte.

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.

For flushing the command line parameters of the input stream of a process, a simple `Flush()` is not necessarily sufficient, see the code sequence at the beginning of section **??** for further details.

5.6.6 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) /* since V36 */
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 the include file `dos/stdio.h` and equals to -1.

This function does not touch `IoErr()` if the character only goes into the internal buffer; `IoErr()` is only updated when the buffer contents is written out.

5.6.7 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) /* since V36 */
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. The error code IoErr() is only updated when the buffer is flushed.

5.6.8 Write a String Buffered to the Output Stream

The PutStr() function writes a NUL-terminated string to the standard output of the calling process. No newline is appended.

```
error = PutStr(str) /* since V36 */
D0             D1

LONG PutStr(STRPTR)
```

This function is equivalent to FPuts(Output(),str), that is, it writes the NUL-terminated string pointed to by str to the output. It returns 0 on success and −1 on error. IoErr() is only adjusted when the buffer of the Output() file handle is flushed. When this happens depends on the buffer mode installed by SetVBuf(), see section **??**

5.6.9 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) /* since V36 */
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, otherwise 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 −1.

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

This function had a defect in version 36 where, after receiving an EOF once, would continue to return ENDSTREAMCH regardless whether new data became available in the (potentially interactive) input stream. This was fixed in version 37.

5.6.10 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) /* since V36 */
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. The function also returns if line-feed character is found, which is copied into the buffer. Finally, if an end of file condition or an error condition is encountered, reading also stops. In either event, the string is NUL terminated.

The function returns NULL in case an error or end-of-file condition was detected. Otherwise, the function returns the buffer passed in. If a line-feed character was read, it remains in the buffer.

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

In AmigaDOS versions prior to version 39, this function would copy one byte too much if no terminator such as a newline or an end of file is found, and it thus overflowed the input buffer by the NUL termination of the string. This was fixed in version 39. To work around this defect, the len parameter should be set to the size of the buffer minus 1.

5.6.11 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) /* since V36 */
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.

Under AmigaDOS version 36, it was not possible to push back an end of file condition. This was fixed in version 37.

5.6.12 Macros for Buffered I/O

The include file dos/stdio.h also defines a couple of additional macros for buffered I/O on the standard input and output streams that are slightly inefficient by requiring an additional call to Input() or Output(). If used in a tight loop, the target file handle should be obtained first and the functions should be called manually rather than through the following macros:

```
#define ReadChar()          FGetC(Input())
#define WriteChar(c)        FPutC(Output(),(c))
```

```
#define UnReadChar(c)         UnGetC(Input(),(c))
#define ReadChars(buf,num)    FRead(Input(),(buf),1,(num))
#define ReadLn(buf,len)       FGets(Input(),(buf),(len))
#define WriteStr(s)           FPuts(Output(),(s))
#define VWritef(format,argv) VFWritef(Output(),(format),(argv))
```

The added value of these macros is quite small, and there is rarely an advantage in using them.

5.7 Working with File Handles

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 `FileHandle` structure:

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

In the following sections, this structure and its elements are discussed.

5.7.1 The FileHandle Structure

A file is represented by a `FileHandle` structure. For example, the `Open()` function returns a BPTR to such a structure. It is allocated by the *dos.library* through `AllocDosObject()`, and then forwarded to the file system or handler for second-level initialization. It is defined by the include file `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 non-interactive and probably representing a file on a file system. `IsInteractive()` makes use of this element. The file system or handler will initialize this value when opening a file according to the nature of the handler.

`fh_Type` points to the `MsgPort` of the handler or file system that implements input and output operations through this handle. Chapter **??** provides additional information on how handlers and file systems work. If this pointer is `NULL`, no handler is associated to the file handle. AmigaDOS will deposit `NULL` here when opening a file to the `NIL` (pseudo-)device. Attempting to `Read()` from such a 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 section **??** make use of.

fh_Pos is the byte offset within fh_Buf from which the next byte will be read, or the buffer position into which the next byte will be written.

fh_End is the size of the buffer in bytes.

fh_Func1 is a pointer to a function 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 pointer to a second function that is called whenever the buffer is full and is to be written out by the handler. Users shall not call this function itself, and the function prototype is neither documented on purpose.

fh_Func3 is a pointer to a function which is called whenever the file handle is closed. This function potentially writes the buffer content out when dirty, releases the buffer if it is allocated by the *dos.library* and finally forwards the close request to the handler.

fh_Arg1 is a file-system specific value the handler or file system may use to identify the file. The interpretation of this value is up 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(), Write() and many other requests. See chapters **??** and **??** for details.

fh_Arg2 is currently unused but reserved for future use through the *dos.library*.

Additional elements have been added at the end of the FileHandle structure in AmigaDOS version 39. They are not publicly documented. For this reason, this structure shall never be allocated manually, but shall be obtained by either opening a file, or through AllocDosObject(), see section **??**.

5.7.2 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 retrieve data. One solution to this problem is to deposit the input data in a temporary file of 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 any other than the intended program accesses or modifies it.

A better alternative is to place the data to be read in a file handle and thus hide them from other programs. AmigaDOS uses a similar technique internally, for example to provide a shell script to be executed to the Run command, which is then installs a string stream as command stream of the shell; AmigaDOS also uses string streams feed command line arguments to shell commands.

The following program demonstrates this technique: It creates a dummy file handle from a NIL: stream and replaces its input buffer with a pointer to the string providing the data to be read. Since the buffer must be representable as BPTR, it is in the example program below allocated through AllocVec() which ensures proper alignment. This step is not necessary if alignment can be guaranteed by other means and the buffer remains available as long as the file handle is in use.

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

int main(int argc,char **argv)
{
  UBYTE *buf       = NULL;
  const char *test = "Hello World!\n";
  const int len    = strlen(test);
```

```
  struct FileHandle *fh;
  BPTR file;

  buf  = AllocVec(len,MEMF_PUBLIC);
  if (buf) {
    memcpy(buf,test,len);

    file         = Open("NIL:",MODE_OLDFILE);
    if (file) {
      fh         = BADDR(file);
      fh->fh_Buf = MKBADDR(buf);
      fh->fh_Pos = 0;
      fh->fh_End = len;
      /* Now read the buffer as a file */
      {
        BPTR out = Output();
        LONG ch;
        while((ch = FGetC(file)) >= 0) {
          FPutC(out,ch);
        }
      }
      Close(file);
    }
    FreeVec(buf);
  }
  return 0;
}
```

5.7.3 Cloning File Handles

Unlike locks (see chapter **??** and section **??**), file handles cannot be duplicated in general. However, if the file is interactive (see section **??**), or a `NIL:` handle, a copy can be made. Thus, for example, files of the the CON-Handler and the Port-Handler can be duplicated and then access the same console, or the same port, with identical parameters.

The following function creates a copy of a file handle. It returns `ZERO` if no copy can be made. The idea is here to temporarily install the handler serving the file handle as console of the calling process, and then open a file to the console through the "`*`" file name. See section **??** for the `SetConsoleTask()` function which modifies the console handler of the calling process:

```
BPTR CloneHandle(BPTR handle)
{
  struct MsgPort *ctask;
  struct MsgPort *fport;
  BPTR newhandle = 0;

  if (handle) {
    fport = ((struct FileHandle *)BADDR(handle))->fh_Type;
    if (IsInteractive(handle)) {
      ctask     = SetConsoleTask(fport);
      newhandle = Open("*",MODE_OLDFILE);
      SetConsoleTask(ctask);
```

```
    } else if (fport == NULL) {
      newhandle = Open("NIL:",MODE_OLDFILE);
    }
  }
  return newhandle;
}
```

The above function returns `ZERO` for files opened to a file system, they cannot be duplicated easily.

5.7.4 An FSkip() Implementation

Even though `Seek()` can be safely mixed with buffered input and output functions, it is then not very efficient as it always performs a full synchronization of the buffer with `Flush()`, loosing any buffer content. The algorithm provided in this section avoids this overhead, but can only skip forwards over bytes by manipulating the file handle buffer pointer. Buffer manipulation has the advantage that small amounts of bytes can be skipped over without requiring interaction with the file system; skipping over larger distances still requires going through `Seek()` to interact with the file system.

The following function implements an `FSkip()` function that selects the most viable strategy for skipping over bytes and is more efficient that `Seek()` on buffered streams.

```
LONG FSkip(BPTR file,LONG skip)
{
  LONG res;
  struct FileHandle *fh = BADDR(file);

  if (fh->fh_Pos >= 0 && fh->fh_End > 0) {
    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(file,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 and 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` need to be initialized for buffered reading immediately after opening the file; instead of `UnGetC()`, any buffered read function is also sufficient.

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

5.7.5 An FGet() Implementation

While the `FRead()` function already provides a buffered read function, it is not very efficient prior to version 47 of AmigaDOS. The following simple function provides in such cases a faster implementation that even allows inlining:

```
LONG FGet(BPTR f,void *buf,LONG size)
{
  struct FileHandle *cis = BADDR(f);

  if (cis->fh_Pos > 0) {
    LONG end = cis->fh_Pos + size;
    if (end < cis->fh_End) {
      memcpy(buf,(UBYTE *)BADDR(cis->fh_Buf) +
                  cis->fh_Pos,(size_t)(size));
      cis->fh_Pos = end;
      return size;
    }
  }
  return FRead(f,buf,1,size);
}
```

This code reads `size` bytes from the file `fh` into the buffer `buf` and returns the number of bytes read. Similar to all functions in section **??**, `IoErr()` is only updated if the handler is required to supply data.

As seen from this implementation, the function attempts to satisfy the read if a partially filled buffer is available. If not, the above code runs into the operating system function `FRead()` which also refills the buffer of the file handle. As for the `FSkip()` implementation presented in section **??**, the file handle requires preparation by a dummy `UnGetC()`, see there.

5.8 Formatted Output

The functions in this section print elementary data types formatted to a file, using a string that defines the types and provides formatting instructions. All functions in this section use the internal buffer of the file handle and are thus buffered.

5.8.1 Print Formatted using C-Syntax to a File

The `VFPrintf()` function prints elementary datatypes using a format string that closely reassembles the syntax of the C language. `FPrintf()` is based on the same entry point of the *dos.library*, though the prototype 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) /* since V36 */
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, see also section **??**. In particular, format strings indicating integer arguments such as %d and %u assume 16-bit integers, independent of the integer model of the compiler. On compilers working with 32-bit integer models, the format modifier l must be used, e.g. %ld for signed and %lu for unsigned integers, and %lc for writing a single character provided as argument.

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

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.

5.8.2 Print Formatted using C-Syntax to the Output Stream

The VPrintf() function prints elementary datatypes using a format string to the Output() stream of the calling process. Printf() is based on the same entry point of the *dos.library*, though the prototype is different and arguments are expected directly as function arguments.

```
count = VPrintf(fmt, argv) /* since V36 */
D0              D1    D2

LONG VPrintf(STRPTR, LONG *)

count = Printf(fmt, ...)

LONG Printf(STRPTR, ...)
```

The Printf() function is the closest analog of the ANSI-C library function printf() within AmigaDOS, but cannot be directly substituted for it as the formatting directives in fmt are subtly different. They are interpreted according to a 16-bit integer model, see also sections **??** and **??** and require the l modifier when printing integers and characters for compilers using a 32-bit integer model. Similar to FPrintf(), this function can be localized and then accepts additional formatting directives from FormatString() function of the *locale.library*.

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

5.8.3 BCPL Style Formatted Print to a File

The VFWritef() function formats its arguments according to a format string based on the syntax defined by the BCPL language. The main purpose of this function is to offer formatted output for legacy BCPL programs to which this function appears as an entry in 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.

```
count = VFWritef(fh, fmt, argv) /* since V36 */
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 conventions 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 into 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 onward. Strings that are too long to fit into the field width are not truncated.

`%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. If padding is required on the left, the number is padded with 0 digits, if the number is too long to fit into the field, it is truncated.

`%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 the `%T` format string. If padding is required, the number is 0-padded on the left. If the number is too long, it is truncated.

`%Ix` Writes a (signed) integer in decimal to the output in a field that is at most `x` characters long. The field length is again indicated by a single character. If the number is shorter than the field width, it is padded with spaces on the left. It is not truncated if the number is longer than the field width.

`%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. If padding is required, the number is padded with spaces on the left. If the number is too long, it is not truncated.

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

`%%` Prints a literal percent sign.

This function is *not* patched by the *locale.library* and therefore is not localized nor enhanced.

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

5.9 Record Locking

While locks, see chapter **??**, control access to file systems object in total, record locks provide access control on contiguous regions of a file. Unlike locks, however, the file system does not block read or write access to the locked region and does not attempt to enforce record locks during read or write operations. Instead, a record lock on a file region only prevents another conflicting record lock on a region that overlaps with the locked region. Record locks therefore require all participating processes to use the record locking functions of this section.

Record locks are a relatively modern concept not all file systems implement. The RAM-Handler and the Fast File System support it.

5.9.1 Locking a Region of a File

The LockRecord() function locks a single region of a file, potentially waiting a limited time for the region to become available.

```
success = LockRecord(fh,offset,length,mode,timeout) /* since V36 */
D0                   D1   D2     D3     D4     D5

BOOL LockRecord(BPTR,ULONG,ULONG,ULONG,ULONG)
```

This function attempts to lock the region of the file identified by fh starting from the byte offset offset and having the byte size length. The mode shall be taken from the constants defined in dos/record.h:

Table 5.5: Record Locking Modes

Record Locking Mode	Description
REC_EXCLUSIVE	Exclusive access to a region, honoring the timeout
REC_EXCLUSIVE_IMMED	Exclusive access to a region, ignoring the timeout
REC_SHARED	Shared access to a region, honoring the timeout
REC_SHARED_IMMED	Shared access to a region, ignoring the timeout

While the same byte within a file can be included in multiple regions locked through a shared record lock, only a single exclusive lock can be held on each byte of a file. Or put differently, shared regions can overlap with each other without blocking or failure, exclusively locked regions cannot overlap with shared locked regions or with each other.

For the REC_EXCLUSIVE and REC_SHARED modes, the timeout arguments provides a time limit in ticks, i.e. $1/50^{\text{th}}$ of a second, after which an attempt to obtain a record lock times out. This time limit may also be 0 in which case an attempt to lock a conflicting region fails immediately.

The REC_EXCLUSIVE_IMMED and REC_SHARED_IMMED modes ignore the timeout, i.e. they act as if the timeout is 0 and fail as soon as they can determine that the requested record cannot be locked.

This function returns 0 in case of failure and then returns a non-zero error code through IoErr(). In case the record lock cannot be obtained because the region overlaps with another locked region, the error will be ERROR_LOCK_COLLISION. If the region can be locked, the call returns a non-zero result code. Even though IoErr() is altered in case of success, its value cannot be relied upon.

5.9.2 Locking Multiple Regions of a File

The LockRecords() function locks multiple records at once, potentially within multiple files.

```
success = LockRecords(record_array,timeout) /* since V36 */
D0                    D1            D2

BOOL LockRecords(struct RecordLock *,ULONG)
```

This function attempts to lock multiple records at once that are included in the the RecordLock structure. This structure is defined in dos/record.h and looks as follows:

```
struct RecordLock {
        BPTR    rec_FH;         /* filehandle */
        ULONG   rec_Offset;     /* offset in file */
        ULONG   rec_Length;     /* length of file to be locked */
        ULONG   rec_Mode;       /* Type of lock */
};
```

The record_array is a pointer to an array of the above structure that is terminated by a RecordLock structure with rec_FH equal to NULL. The elements of this structure correspond to the arguments of the LockRecord() function:

rec_FH is the file handle to the file within which a record is to be locked. It shall be NULL for the last element in the array.

rec_Offset and rec_Length specify the region in the file to be locked as start position within the file and the number of bytes in the region.

rec_Mode specifies the type of the lock that is to be obtained. It shall be one of the modes listed in table **??**; the modes are all defined in dos/record.h.

The timeout argument specifies how long each of the attempts to obtain a non-immediate lock is supposed to wait for a record to become available; it is measured in ticks. It is ignored for records to be locked immediately. The LockRecords() works through the elements of the RecordLock array sequentially until either all records could be locked, or until locking one of the records failed. In the latter case, the function unlocks all locks obtained so far and then returns with failure. That is, the maximal time LockRecords() blocks is the sum of all timeouts.

On failure, i.e. if one of the records could not be locked, the function returns 0 and sets IoErr() to an error code. On success, the function returns a non-zero result. Even though IoErr() is altered in case of success, its value cannot be relied upon.

Unlike what the function prototype suggests, this function is *not atomic*. It attempts to lock the records sequentially one after another, applying the same timeout for each call, unless rec_Mode indicates that no timeout shall be applied to a particular record. Thus, it can happen that another task attempts for a lock of a conflicting region while the first caller is executing this function. It is therefore recommended to establish an order in which records within a file are locked, e.g. from smallest to largest start offset.

5.9.3 Unlocking a Region of a File

The UnLockRecord() function unlocks a region of a file, making it available for other record locks. The provided region shall match one of the regions locked before, i.e. it is not possible to partially unlock a region and leave the remaining bytes of the region locked.

```
success = UnLockRecord(fh,offset,length) /* since V36 */
D0                     D1   D2     D3

BOOL UnLockRecord(BPTR,ULONG,ULONG)
```

This function unlocks a region of a file locked before by LockRecord() or LockRecords(). The region starts offset bytes within the file identified by fh and is length bytes large.

This function returns 0 on failure and sets an error code that can be obtained by IoErr(). A possible error code is ERROR_RECORD_NOT_LOCKED if an attempt is made to unlock a record that has not been not locked before, or to partially unlock a record. On success, the function returns a non-zero result code. Even though IoErr() is altered in case of success, its value cannot be relied upon then.

5.9.4 Unlocking Multiple Records of a File

The UnLockRecords() function unlocks multiple records provided in an array of RecordLock structures at once, sequentially releasing one record after another.

```
success = UnLockRecords(record_array) /* since V36 */
D0                      D1

BOOL UnLockRecords(struct RecordLock *)
```

This function releases multiple records provided in an array of `RecordLock` structures. The last element of the structure is indicated by its `rec_FH` element set to `NULL`. This structure is defined in section **??**.

The function calls `UnLockRecord()` in a loop, and is therefore *not atomic*. In case unlocking any of the records fails, the function returns 0 but continues to attempt to unlock all remaining records in the array. If unlocking all regions succeeds, it returns a non-zero result code. If unlocking one of the regions fails, it returns with `DOSFALSE`, but unfortunately does not set `IoErr()` consistently as the error code is not saved on a failed unlock.

Chapter 6

Locks

Locks are access rights to file system objects, such as files or directories. Once an object has been locked, it can no longer be deleted. Locked files cannot be replaced or deleted, and depending on the type of the lock, can neither be opened for reading or writing.

Locks come in two types: *exclusive locks* and *shared locks*. Only a single exclusive lock can exist on a file system object at a time, and no other lock can exist on an exclusively locked object simultaneously. An attempt to lock an exclusively locked object results in the error `ERROR_OBJECT_IN_USE`, and attempting to exclusively lock an object that is already locked by a shared lock will also fail likewise. Unlike exclusive locks, multiple shared locks can be kept on the same object simultaneously.

Locks can also be used to identify files or directories on file systems. They help to work around the 255 character limit discussed in chapter **??** most *dos.library* functions taking path names as arguments suffer from. Long paths should be substituted by a pair of a lock on the containing directory, and a short file or directory name identifying the object within the locked directory. Even though paths are length limited, there is no restriction on the nesting depth of the directory structure within a hierarchical file system.

The `ZERO` lock, i.e. the lock of the value `ZERO`, is a special case and identifies the root directory of the boot volume. It therefore corresponds to the path name `SYS:`, see also section **??**[1].

Locks are related to file handles: From a lock, a file can be opened through by `OpenFromLock()`, see **??**, and a lock associated to an open file can be obtained from `DupLockFromFH()`, see section **??**. A typical file system design is to associate each file handle with an internal (hidden) lock on the file. A file opened for exclusive access with `MODE_NEWFILE` implies an exclusive lock, while all other modes correspond to shared locks, compare with table **??** in section **??**.

As long as at least a single lock is held on an object, the file system will keep the volume containing the locked object within the device list of the *dos.library*, which is discussed in chapter **??**. This has, for example, the consequence that the Workbench will continue to show its volume icon.

Locks cannot be held on links (see chapter **??**) because links — regardless of whether they are soft or hard links — are resolved as soon as a lock is acquired, and thus only link targets and not the links themselves can be locked.

6.1 Obtaining and Releasing Locks

Locks can be obtained either explicitly from a path, or can be derived from another lock or file handle. As locks block exclusive access to an object in a file system, locks should be released as early as possible to allow other processes to gain access to the same object.

[1] Or, at least, it should. The `ZERO` lock actually corresponds to the root directory of the file system in `pr_FileSystemTask`, and `SYS:` is an assign independent of it, though typically the two are identical, see chapter **??**.

6.1.1 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 to the current directory of the calling process, see chapter **??** for these concepts.

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

BPTR Lock(STRPTR, LONG)
```

This function locks the object identified by `name`, the path of the object to lock. 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 6.1: 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 a given object at a time.

The return code `lock` identifies the lock as a BPTR to a `FileLock` structure, see section **??**. It is non-`ZERO` (see **??**) on success, or `ZERO` on failure. In either case, `IoErr()` is set, to an undefined value on success or an error code on failure. See section **??** for a list of error codes.

6.1.2 Duplicating a Lock

The `DupLock()` function replicates a given lock, returning a copy of the lock on the same object. This requires that the original lock is a shared lock, and the function returns a shared lock on success.

```
copy = DupLock( lock )
D0              D1

BPTR DupLock(BPTR)
```

This function copies the (shared) lock passed in as `lock` and returns a copy of it in `copy`. It is not possible to copy a lock by copying the `FileLock` structure; this *does not* work and attempting to do so can crash the system.

In case of error, `DupLock()` returns `ZERO`, and then `IoErr()` returns an error code identifying the problem. On success, `IoErr()` is set to an undefined value. It is not possible to copy an exclusive lock, in such a case the error `ERROR_OBJECT_IN_USE` is generated.

Attempting to duplicate the `ZERO` lock has its pitfalls. While AmigaDOS up to its most recent version returns `ZERO` as the result of `DupLock(ZERO)`, it leaves `IoErr()` untouched, making it hard to distinguish this result from the error case. If an application program requires to cover this situation, a possible workaround is to call `SetIoErr(0)` upfront to reset the secondary return code, then call `DupLock()` and test `IoErr()` if the result was `ZERO`. If `IoErr()` is still 0, the `ZERO` lock was duplicated and the `ZERO` result of `DupLock()` does not indicate an error.

6.1.3 Obtaining the Parent of an Object

The `ParentDir()` function obtains a shared lock on the directory containing the object that is described by the lock passed in. For locks on directories, this is the parent directory, for locks on 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 determined; this lock *is not released* by this function. The function returns a shared lock on the directory containing this object, regardless of the type of the lock passed in. 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, i.e. the root directory, or the `ZERO` lock itself.

To distinguish the two cases, the caller should use the `IoErr()` function; if this function returns 0, then no error occurred and the passed in object is the topmost directory and does not have a parent directory. If `IoErr()` returns a non-zero error code, then the file system failed to lock the parent directory. Possible reasons are lack of memory, or an exclusive lock on the the parent that prevents to obtain a shared lock on it.

6.1.4 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 to be created iteratively 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, a directory whose name is given by the last component of the path (see chapter **??**) will be created. If a file or directory of the same name already exists, this function will fail with the error code `ERROR_OBJECT_EXISTS`[2]. If successful, the function returns an exclusive lock to the created directory, otherwise it returns `ZERO`.

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

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

6.1.5 Releasing a Lock

Once a lock is no longer required, it should be released as soon as possible to allow other functions or processes to access, overwrite or delete the previously 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 action. This function does not return an error code, and neither alters `IoErr()`.

[2]The information in [**?**] that in such a case a conflicting file or directory is deleted is not correct.

6.1.6 Changing the Type of a Lock or File Handle

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 locked file system object — vice versa. This function is also able to change the access mode of a file handle.

```
success = ChangeMode(type, object, newmode) /* since V36 */
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 is as in table **??**, where the types are defined in dos/dos.h:

Table 6.2: 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 **??**, i.e. SHARED_LOCK to make either the file handle 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 an undefined value. Otherwise, the function returns 0 and sets IoErr() to an appropriate error code.

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

6.1.7 Comparing two Locks

The SameLock() function compares two locks and returns information whether they refer to the same file system object, or are at least locks to objects on the same volume.

```
value = SameLock(lock1, lock2) /* since V36 */
D0               D1     D2

LONG SameLock(BPTR, BPTR)
```

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

Table 6.3: Lock Comparison Return Code

Return Code	Description
LOCK_SAME	The locks represent the same object
LOCK_SAME_VOLUME	Locks are on different objects, but on the same volume
LOCK_DIFFERENT	Locks are on objects on different volumes

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

Under AmigaDOS versions 34 and below, this function is not available. The following code may be used as a (limited) workaround:

```
#include <dos/dos.h>
#include <dos/dosextens.h>

LONG SameLockV34(BPTR lock1,BPTR lock2)
{
  struct FileLock *fl1 = BADDR(lock1);
  struct FileLock *fl2 = BADDR(lock2);

  /* This implies two ZERO locks */
  if (fl1 == fl2)
    return LOCK_SAME;

  /* Does not indicate SYS: == NULL */
  if (fl1 && fl2) {
    if (fl1->fl_Volume != fl2->fl_Volume)
      return LOCK_DIFFERENT;
    if (fl1->fl_Task   != fl2->fl_Task)
      return LOCK_DIFFERENT;
    if (fl1->fl_Key    != fl2->fl_Key)
      return LOCK_SAME_VOLUME;
    return LOCK_SAME;
  }
  /* Not fully correct */
  return LOCK_DIFFERENT;
}
```

The above function will work correctly for the AmigaDOS ROM file system, the FFS and the RAM-Handler versions 34 and below *only*. For other file systems, and newer variants of the above file systems, proper results cannot be expected as the above algorithm depends on `fl_Key` uniquely identifying the locked object. This is, however, not necessarily true as `fl_Key` is, actually, a file system internal value that cannot be safely interpreted, see also section **??**. AmigaDOS version 36 introduced a new interface to request comparing two locks from the file system, see section **??**. However, even the current *dos.library* falls back to the above algorithm in case this interface is not implemented by the target file system.

6.1.8 Compare two Locks for their Device

The `SameDevice()` function checks whether two locks refer to file system objects that reside on the same physical device, even if on potentially different partitions on the same medium.

```
same = SameDevice(lock1, lock2) /* since V36 */
D0                D1     D2

BOOL SameDevice( BPTR, BPTR )
```

The `SameDevice()` function takes two locks `lock1` and `lock2` and checks whether they were created by file systems that operate on the same physical device, even if the two locks refer to different file systems or different partitions. Only the exec device and the corresponding unit is compared, that is, this function is not able to determine whether the locks refer to file systems on the same or different physical volumes.

This function returns a non-zero result if the file systems handling the locks operate on the same unit of the same exec device, and it returns 0 otherwise. If the function is not able to identify the file systems, or cannot identify the lower level exec `device` on which the file systems operate, the function also returns 0.

Unfortunately, this function is not necessarily fully reliable as locks can be transferred from one medium to another, e.g. if a disk is removed from the first floppy drive and re-inserted into the second, and thus, the physical device the locked object is located on can change even *after* this function has been called.

A possible use case of this function is to determine whether the involved file systems can operate in parallel without imposing speed penalties due to conflicting medium accesses. Thus, copy functions may be optimized depending on the result as no intermediate buffering need to be used if source and destination are on different physical devices.

This function does not set `IoErr()`, even if it cannot determine the device a file system operates on.

6.2 Locks and Files

While it is not a necessary implementation strategy for file systems, each file handle can be considered to be associated to a lock on the file that has been opened. The functions in this section allow to copy a lock from a file handle, and to open a file handle from a lock. The type of the lock corresponds to the access mode a file has been opened with, table **??** shows this relation:

Table 6.4: 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 version 36 which was then later copied by the Fast File System implementation. In AmigaDOS versions 34 and below, this mode was associated to an `EXCLUSIVE_LOCK`. For versions 36 and above, exclusive access to a file without deleting its contents can, however, be established through the `OpenFromLock()` function by providing an exclusive lock to the file to be opened as argument.

6.2.1 Duplicate the Implicit Lock of a File

The `DupLockFromFH()` function creates a lock from a file handle of an opened file. For this function to succeed, the file must have been opened in the modes `MODE_OLDFILE` or `MODE_READWRITE`. Files opened with `MODE_NEWFILE` are based on an implicit exclusive lock that cannot be duplicated.

```
lock = DupLockFromFH(fh) /* since V36 */
D0                       D1

BPTR DupLockFromFH(BPTR)
```

This function creates a lock from the file handle `fh` and returns it in `lock`; that is, the returned lock is a lock on the file which is accessed through the handle `fh`. This file handle remains valid and usable after this call. In case of failure, `ZERO` is returned. `IoErr()` is set to an undefined value in case of success, or to an error code on failure.

6.2.2 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, this function it is roughly equivalent to first obtaining a lock on the file through `DupLockFromFH()`, and then calling `ParentDir()` on it, except that it is also able to process files opened in the `MODE_NEWFILE` mode.

```
lock = ParentOfFH(fh) /* since V36 */
D0                D1

BPTR ParentOfFH(BPTR)
```

This function returns a shared lock on the directory containing the file opened through the `fh` file handle. The file handle remains valid and usable after this call. This function returns `ZERO` in case of failure. `IoErr()` is set to an error code from `dos/dos.h` in case of failure, or to an undefined value in case of success.

6.2.3 Opening a File from a Lock

The `OpenFromLock()` function uses a lock to a file 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 anymore. It will be released by the file system upon closing the returned file handle.

```
fh = OpenFromLock(lock) /* since V36 */
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. This function always sets `IoErr()`, to an error code in case of failure, or to an undefined value in case of 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()` in a second step.

While this function is not available in AmigaDOS versions 34 and below, the following workaround may be used, which, however, only operates on shared locks and cannot open files from exclusive locks:

```
BPTR OpenFromLockV34(BPTR lock)
{
  /* Yes, really, change the directory to a file! */
  BPTR dir = CurrentDir(lock);
  BPTR fh  = Open("",MODE_OLDFILE);

  if (fh)
    UnLock(lock);

  CurrentDir(dir);

  return fh;
}
```

This function depends on the empty string describing the object locked by the current directory of the calling process, even if this locked object turns out to be a file. This is a feature every AmigaDOS file system supports, see also section **??**.

6.3 Retrieve Information on the State of the Medium

The `Info()` function returns information on the file system on which the locked object is located, and fills an `InfoData` structure with the status of the file system. If it is instead intended to retrieve information on the currently inserted volume, i.e. without requiring a lock, direct communication with the file system on the packet level is required by sending a packet type of `ACTION_DISK_INFO`, see section **??**. Since this packet fills also fills an `InfoData` structure, some information in this section applies to it as well.

```
success = Info( lock, parameterBlock )
D0              D1    D2

BOOL Info(BPTR, struct InfoData *)
```

The `lock` is a lock to an arbitrary object on the volume to be queried; its only purpose is to identify it. The function fills an `InfoData` structure that shall be aligned to long-word boundaries. It is defined in `dos/dos.h` and reads as follows:

```
struct InfoData {
   LONG   id_NumSoftErrors;
   LONG   id_UnitNumber;
   LONG   id_DiskState;
   LONG   id_NumBlocks;
   LONG   id_NumBlocksUsed;
   LONG   id_BytesPerBlock;
   LONG   id_DiskType;
   BPTR   id_VolumeNode;
   LONG   id_InUse;
};
```

The elements of this structure are interpreted as follows:

`id_NumSoftErrors` counts the number of read or write errors the file system detected during its life-time. It is not particularly bound to the currently inserted medium.

`id_UnitNumber` is the unit number of the exec device on which the file system operates, and hence into which the volume identified by the lock is inserted.

`id_DiskState` identifies the status of the file system, whether the volume is writable and whether it is consistent. Disk states are defined in `dos/dos.h` and set according to the following table:

Table 6.5: Disk States

Disk State	Description
`ID_WRITE_PROTECTED`	The volume is write protected
`ID_VALIDATING`	The volume is currently validating
`ID_VALIDATED`	The volume is consistent and read- and writable

A volume in the state `ID_WRITE_PROTECTED` has been identified as consistent, but does not accept modifications, either because the medium is physically write-protected, or because it has been locked by software, see section **??**.

A volume is in the state `ID_VALIDATING` if its file system detected inconsistencies; some file systems, including the Fast File System, then trigger a consistency check of the volume. The Fast File System rebuilds the bitmap of the volume that describes which blocks are allocated and which are free. It cannot fix more severe errors; if some are detected, it presents a requester to the user indicating the problem. During validation, file systems typically refuse to accept write requests. If validation cannot bring the volume into a consistent state, the disk state remains `ID_VALIDATING`.

A volume in state ID_VALIDATED is consistent and read- and writable.

id_NumBlocks is the total number of blocks into which the medium is divided. This includes both free and occupied blocks, and thus indicates the total capacity of the volume. This number is not necessarily constant. The RAM-Handler adjusts this value according to the available memory; RAM-Handler versions prior to 45 set this to 0. This means, in particular, that care needs to be taken when the disk fill state in percent is computed by a dividing the number of used blocks by this number.

id_NumBlocksUsed is the number of blocks occupied by the file system on the disk. As it is dependent on the file system how many blocks are needed in addition to the actual payload data, no conclusion can be derived from this number whether a particular file fits on the volume. RAM-Handlers prior to release 45 did not even fill this with a useful value. This element does not contain a useful number for file systems that are currently in the state ID_VALIDATING.

id_BytesPerBlock is the number of bytes available for payload data in a physical block of the medium, and not necessarily the physical block size into which the storage medium is divided. Some file systems require additional bytes of the physical block for administration. Even the RAM-Handler segments data into blocks and provides in this element the number of data bytes stored there.

id_DiskType identifies whether the file system can identify the disk structure and claims responsibility for it. If the Info() function succeeds, then this is always the case; if the medium is not available or its structure is unknown, Info() returns DOSFALSE and indicates an error in IoErr(). However, if instead direct packet communication and ACTION_DISK_OBJECT is used to query the state of the currently inserted volume and thus no lock is provided, then all other entries of table **??** can be supplied as well, see also section **??** and chapter **??** how to use this alternative mechanism to fill an InfoData structure.

Unlike what the name suggests, id_DiskType is *not* a general identifier of the file system type and shall not be used to identify a particular file system. For legacy reasons, the various flavors of the Fast File System also leave their identifier here, though this principle should not be carried over to new designs. Instead, a file system should rather return the generic ID_DOS_DISK if it finds a medium for which it claims responsibility. Even if the file system recognizes the disk structure as one of its own, it is possible that the structure is considered inconsistent by setting id_DiskState to ID_VALIDATING.

AmigaDOS currently defines the following disk types in dos/dos.h:

Table 6.6: Disk Types

Disk Type	Description
ID_NO_DISK_PRESENT	No disk is inserted
ID_UNREADABLE_DISK	Reading disk data failed at exec device level
ID_DOS_DISK	The disk is in a format the file system is able to interpret
ID_NOT_REALLY_DOS	While physically accessible, the disk structure is invalid
ID_KICKSTART_DISK	A disk containing an A1000 Kickstart
'BUSY'	The file system is currently inhibited
'CON\0'	Not a file system, but returned by the CON-Handler, see **??**
'RAW\0'	Not a file system, but returned by the CON-Handler, see **??**
All others	The first long word of boot block of the medium or partition

Not the DosType While mountlists include a DOSTYPE field that identifies the file system uniquely, the id_DiskType element *does not* represent this DOSTYPE. That it coincides with the DOSTYPE for variants of the FFS is a historical error that shall not be mirrored by new file system designs. It is therefore advisable to check the first 3 bytes of the id_DiskType for the characters DOS, and if so, assume that the disk is valid and can be interpreted by the file system. Unfortunately, some third-party designs do not follow this convention.

As mentioned above, ID_DOS_DISK is the id_DiskType file systems should return in case they rec-

ognize the structure and attempt to interpret them. Despite this fact, the Fast File System returns erroneously the DOSTYPE of its mountlist entry as reported in table **??** of section **??**.

ID_NOT_REALLY_DOS and ID_UNREADABLE_DISK both indicate disks the file system cannot make use of. The first value indicates that the logical structure of the disk content cannot be interpreted, and the second that the underlying exec device cannot gain access to the contents of the blocks, i.e. the physical layer of the disk is not readable. The Info() function cannot return either of these disk types as the corresponding file system interface fails then with the error code ERROR_NOT_A_DOS_DISK. However, the packet type ACTION_DISK_INFO not requiring a lock but analyzing the inserted volume directly can return them, see section **??**.

ID_NO_DISK_PRESENT indicates that the drive currently does not contain any disk. Similar to the above, Info() cannot return this disk type as the file system interface fails early with ERROR_NO_DISK, but the packet type ACTION_DISK_INFO of the packet level interface filling the same InfoData structure can. See again section **??**.

'BUSY' is a four-character constant that is not documented in dos/dos.h, but returned whenever a file system has been inhibited, i.e. its access to the physical layer has been blocked. Thus, any[3] attempt to access this file system is currently suspended, probably because some program attempts to modify the medium on the device level. Disk editors or disk salvage programs will typically block file systems from touching the medium while they work on it. The Inhibit() function of section **??** performs this, and thus leaves 'BUSY' as disk type.

'CON\0' and 'RAW\0' are indicators left by the CON-Handler (or console-type handlers) which use the InfoData structure for other purposes, see section **??**. As such handlers do not (in general) hand out locks, the Info() function cannot return these two types, but only direct handler communication with a packet type of ACTION_DISK_INFO can.

All other types are returned in case the file system cannot interpret the disk structure; the Fast File System copies them from the first 4 bytes of the boot block, see **??**, into id_DiskType. In case these bytes are all 0, the disk type ID_NOT_REALLY_DOS is returned instead. One special case is ID_KICKSTART_DISK which identifies a disk containing the A1000 Kickstart. This is, however, just a side effect of the first four bytes of the floppy containing the characters 'KICK'. As above, the Info() function cannot return such disk types, but instead aborts with the error code ERROR_NOT_A_DOS_DISK, but the similar packet level interface based on ACTION_DISK_INFO can.

id_VolumeNode in the InfoData structure is a BPTR to the DosList structure corresponding to the volume on which the object identified by the lock is located. The DosList structure is discussed in great detail in chapter **??**. For ACTION_DISK_INFO, this identifies the currently inserted volume, if any. It is ZERO if the file system does not claim any volume at this moment.

id_InUse counts the number of locks and files currently open on the medium identified by lock, or on the currently identified medium if filled by ACTION_DISK_INFO.

The Info() function returns a non-zero result code on success and sets then IoErr() to an undefined value. On failure, it returns 0 and sets IoErr() to an error code.

6.4 The FileLock structure

Locks have been discussed so far as being opaque identifiers. They are, however, BPTRs to FileLock structures as defined in dos/dosextens.h, as obtained by the following code fragment:

```
BPTR lock = Lock("S:Startup-Sequence",SHARED_LOCK);
struct FileLock *fl = BADDR(lock);
```

[3] Almost any, actually. Low-level initialization and serialization the disk still works and even requires this state, see sections **??** and **??**.

The definition of the structure is as follows:

```
struct FileLock {
    BPTR                fl_Link;
    LONG                fl_Key;
    LONG                fl_Access;
    struct MsgPort *    fl_Task;
    BPTR                fl_Volume;
};
```

Locks are not allocated nor released by the *dos.library* but by file systems when locking or unlocking files or directories. A file system may therefore allocate objects that are somewhat larger than the officially documented structure and that may carry additional elements that are not shown here.

The structure elements are described in the following; the first two elements are file system internal and should not be used by application programs.

The `fl_Link` element has no practical value for users; file systems use it to keep locks on objects located on the same volume in a singly linked list. Such lists does not exist permanently, but are created once a volume is ejected; all links on an ejected volume are queued up in the `dol_LockList` element of the `DosList` structure describing the volume. Another file system responsible for the same medium, but potentially managing a different drive will pick the locks up from there once the volume is reinserted, see also chapter **??**, and then claims responsibility for them by updating their `fl_Task` elements, see below. This mechanism of transferring file system access rights from one medium to another is rather unique to AmigaDOS.

The `fl_Key` element 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 as its meaning is file system dependent. The Fast File System stores here the block number (the "key") of the file header block or directory block that has been locked, see sections **??**, **??** and **??** for the structure of these blocks.

The `fl_Access` element keeps the access type of the lock and reflects the `accessMode` parameter of the `Lock()` function. It is either `SHARED_LOCK` or `EXCLUSIVE_LOCK`.

The `fl_Task` element points to the message port of the file system to be contacted for processing requests on the lock. Any activity on the lock goes through this port, see also section **??**. In case the medium containing the locked object has been ejected and the lock has not yet been reclaimed by any other file system, this is the pointer to the port of the most recent file system responsible for the lock.

The `fl_Volume` is a BPTR to the `DosList` structure on the device list identifying the volume the locked object is located on. Chapter **??** provides further information on this list and its entries.

Chapter 7

Working with Directories

Hierarchical file systems organize their contents in *directories* where each directory contains files, and possibly directories which can contain further directories and files, and so on. The structure of the data on a file system is thus organized as a *tree* where the directories are nodes, and the files form the leaves of tree. The root of this tree is the root directory of the medium or partition. In contrast, a flat file system contains only a single directory that can only contain files, see also section **??**.

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[1].

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

7.1 Examining Objects on File Systems

Given a lock on a file or a directory, information on the locked object can be requested by the `Examine()` function of the *dos.library*. `ExNext()` iterates through the contents of a directory entry by entry, starting from a lock on a directory. The `ExAll()` function is an advanced interface that allows to read multiple directory entries at once, potentially filter them, and thus minimize the call overhead.

> *May go away while you look!* As AmigaDOS is a multitasking operating system, directory contents may change anytime while scanning. In particular, entries 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 the locked directory goes away, it does *not* prevent other processes from adding or removing objects, so beware.

While `ExAll()` seems to provide an advantage by reading multiple directory entries in one go, the AmigaOs ROM file system — the FFS — does usually not profit from this feature, at least not unless a variant of the FFS is used that utilizes a directory cache. The latter has, however, other drawbacks and should be avoided as it creates overhead when creating or deleting objects, see section **??**. Actually, `ExAll()` is (even more) complex to implement, and it is probably not surprising that its implementation in multiple file systems have issues. The `dos.library` provides an `ExAll()` fall-back implementation for those file systems that do not implement it, but even this (ROM-based) implementation had (and still has) issues. Therefore, `ExAll()` has probably less to offer than it seems.

[1] This turns, strictly speaking, the tree structure into a graph.

Examine() and ExNext() fill a FileInfoBlock structure that provides all 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;
   char   fib_FileName[108];
   LONG   fib_Protection;
   LONG   fib_EntryType;
   LONG   fib_Size;
   LONG   fib_NumBlocks;
   struct DateStamp fib_Date;
   char   fib_Comment[80];
   UWORD  fib_OwnerUID;
   UWORD  fib_OwnerGID;
   char   fib_Reserved[32];
};
```

The semantics of the elements 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. The Fast File System stores here the disk key, see section **??**, administrating the object. Note that this key need not to be identical to the fl_Key element of the FileLock structure, for example if the directory entry is a link as it will be transparently resolved by Lock(). The RAM-Handler stores here a pointer to a structure administrating the object. Some file systems do not even fill this element at all, and thus, it cannot be used to uniquely identify a file system object.

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

Table 7.1: 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
> 0	A regular directory
< 0	A regular file

Note that Lock() resolves soft and hard links transparently, and thus, Examine() will never identify such types, only ExNext() specified in section **??** and ExAll() of section **??** can when iterating over the contents of a directory. While the FFS implements hard links through separate types, other file systems cannot identify them at all as hard links are implemented as duplicate directory entries pointing to the same file system object; hard links then only appear as regular directory entries. This applies for example to all variants of the Linux ext file system. Thus, one cannot depend on hard links being easily identifiable. The external links implemented by the RAM-Handler can neither be identified by the fib_DirEntryType and they also appear as regular files or directories.

All other values larger than zero indicate directories, and all other values smaller than zero indicate files. Section **??** provides more details on all types of links.

fib_FileName is the name of the object as NUL terminated string. For the FFS, this name can be at most 30, 56 or 106 characters long, depending on the flavor of the FFS, see table **??** in section **??**.

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

Table 7.2: 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 file 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. if a particular protection function is *not* active.

If the FIBB_DELETE is set, then the file or directory is delete protected. This not only prevents erasing a file completely through DeleteFile(), but also erasing it by creating a new file on top through the Open() mode MODE_NEWFILE. Attempting to delete a protected file system object creates the error code ERROR_DELETE_PROTECTED.

The FIBB_EXECUTE flag is only interpreted by the Shell (see chapter **??**). If the bit is set, the Shell refuses to load the file as a command. This bit is respected by Shell versions 34 and up. It is ignored for directories and by the Workbench.

If the FIBB_WRITE bit is set, then the file system refuses to modify the file, including to overwrite it by opening it with the mode MODE_NEWFILE. An attempt to modify or overwrite the file will instead return the error code ERROR_WRITE_PROTECTED. Deleting the file is, however, still permitted. This flag was ignored by the ROM file system of AmigaDOS versions 34 and below, i.e. by the OFS, and the flag is meaningless for directories.

If the FIBB_READ bit is set, then opening the file for reading fails with ERROR_READ_PROTECTED. This flag is also unsupported by the AmigaDOS ROM file system versions 34 and below, i.e. the OFS, and it is meaningless for directories.

The FIBB_ARCHIVE flag is typically used by archival software. Such software will set this flag when archiving a file, whereas the file system will reset the flag when writing to or modifying a file, or when creating new files. The archiving software is thus able to learn which files had been altered since the last backup. While this flag is also cleared for directories whenever one of the files contained in is modified, modifications in any of its sub-directories are *not* reflected in the parent directory. In other words, this bit is not updated recursively. As a result, archival software still needs to work through directories whose FIBB_ARCHIVE is set because some of its (indirect) children could have been modified.

The FIBB_PURE flag indicates reentrant 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 multiple processes in parallel; in other words, such code is *reentrant*. This implies, in particular, that the binary must allocate all its modifiable objects from the stack or the heap. As most C compilers will, however, place objects with external linkage in DATA or BSS sections and will thus cause the compiled executable to modify its own segments, such code is typically *not pure*. Some compilers provide, however, options where such objects are manually allocated through the compiler startup code, and thus the generated code will be pure.

The Shell command Resident loads pure binaries into RAM for future usage and thus reduces loading times, see sections **??** and **??**. Unfortunately, nothing in the Shell actually *checks* whether a binary is actually pure, or whether its segments have been modified. Thus, the AmigaDOS Shell depends on the user setting this flag properly. This flag is supported and recognized from version 34 of the AmigaDOS Shell and up.

The FIBB_SCRIPT flag indicates whether a file is a Shell, ARexx or any other kind of script. If this flag is set, and the script is used as command, the Shell will forward this file to a suitable script interpreter, such as ARexx or Execute. AmigaDOS versions 45 and up also allow additional script interpreters, to be indicated in the first line of a script, see section **??** for the details.

The FIBB_HOLD flag indicates whether a command is made resident upon loading it the first time. If the flag is set, 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. This bit was introduced in version 36 of AmigaDOS, but stripped again from the sources in release 40 due to ROM space limitations. It was reintroduced in AmigaDOS version 45.

Protection bits of links are not necessarily synchronized with the bits of the corresponding link target. The FFS and the RAM-Handler, however, keep the protection of the link and the link target identical for hard links. For soft links, the situation is more complicated as the FFS does not allow to set protection bits of the link and rather updates the bits of the link target, whereas the RAM-Handler keeps separate protection bits for the link itself. Separate protection bits for links have only a questionable value as they cannot be easily checked without iterating through a directory — this is because links are resolved automatically if an object is locked for inspection. For external links as provided by the RAM-Handler, linked objects are equipped with their own set of protection bits as the handler creates a copy of the linked object on an attempt to lock or open it, see also section **??**.

The fib_EntryType element shall not be used; it can be identical to the fib_DirEntryType, but its use is not documented and it therefore should be left alone.

The fib_Size element indicates the size of the file in bytes. It should have probably been defined as an unsigned type, and it is unclear how files larger than 2GB (or 4GB) could be represented. Its value is undefined for directories, hard links to directories or soft links.

The fib_NumBlocks element indicates now many blocks a file occupies on the storage medium, if such a concept applies at all. Disks and harddisk organize their storage in 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 or file systems that organizes their contents differently, e.g. network file systems. The Fast File System includes in the block count also the file extension blocks, see section **??**, and not only counts the data blocks. Unfortunately, not much can be concluded from the block count as it need not to be consistent across different file systems, or even different flavors of the same file system. For some file systems, it need not to be consistent even for identical file contents.

The fib_Date element indicates when the file was changed last; it is a DateStamp structure as specified in section **??**. Depending on the file system, the date may also indicate when the last modification was made for a directory, such as creating or deleting files or directories within. For the Fast File System modifications of file or directory metadata, such as file comments or protection bits are not reflected here. Which operations exactly trigger an update of the modification date of a directory is file system dependent.

As for protection bits, the AmigaDOS ROM file systems (FFS and RAM) synchronize the date of hard links and their link targets, but keep separate creation dates for soft links. The FFS will, however, update the date of the soft link target upon attempting to modify the date of the link. This is not necessarily true for other file systems but rather an implementation choice.

The fib_Comment element contains a NUL terminated string to a comment on the object. 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. Hard- and Soft Links on the FFS and the RAM-Handler are equipped with file comments of their own and do not mirror the file comment of their link target. However, this is an implementation detail of the file system and not necessarily true for other file systems.

The fib_OwnerUID and fib_OwnerGID elements are filled in by some multi-user aware file systems. They are opaque values representing the owner and group ID owning the file or directory. While the FFS stores such information, no provision is made to moderate access to a particular file according to its owner or its group. The two concepts are alien to AmigaDOS itself. These elements were introduced in AmigaDOS version 43.

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

7.1.1 Retrieving Information on an Directory Entry

The `Examine()` function fills information on an object identified by a lock into a `FileInfoBlock`.

```
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 **??** in more detail. As the `Lock()` function always resolves links, this function will not return a `fib_DirEntryType` that corresponds to a link.

If the `lock` is `ZERO`, this function will examine the root directory of the default file system of the calling process, i.e. typically `SYS:`.

The function returns non-zero in case of success, and 0 for failure. In either case, `IoErr()` is set, to an undefined value on success, or to 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 **??** provides some additional hints on how to allocate such structures.

7.1.2 Retrieving Information from a File Handle

While `Examine()` retrieves information a locked object, `ExamineFH()` retrieves the same information from a file handle.

```
success = ExamineFH(fh, fib) /* since V36 */
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 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 **??**.

Currently, throughout all versions, the *dos.library* is not able to handle an attempt to examine a `NIL` file handle gracefully; it will just crash. Callers should therefore check the `fh_Type` element of the file handle upfront and fail if it is `NULL`.

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

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

7.1.3 Scanning through a Directory Step by Step

The `ExNext()` function iterates through the 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 represented by 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 **??**. The `lock` shall be a lock on a directory, in particular, and shall have been prepared for a directory scan by calling `Examine()` before. The `ZERO` lock to identify the root directory of a volume is typically *not sufficient* here.

Unlike `Examine()` and `ExamineFH()`, this function can be able, depending on the file system, to identify soft- and hard links and does not attempt to resolve them. Whether file systems are actually able to distinguish between a link and its target is another matter, see also section **??** for more information.

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 an undefined value in case of success, or to an error code otherwise.

At the end of the directory, `ExNext()` returns `DOSFALSE`, and error code returned by `IoErr()` will be `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 it, which is not only complex, but also slows scanning the directory down. 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. Similarly, the same lock should not be used by more than one directory scan at the same time. This is particularly important for locks `GetDeviceProc()` returns for assigns. As the same lock will be provided to each caller, the returned lock shall be duplicated first by `DupLock()` before initiating a directory scan on the assign target.

The following example code lists the contents of the directory on the console, where the directory to list is represented by a lock. It uses the `D_S` macro of section **??** to allocate a temporary `FileInfoBlock` structure on the stack, and returns a Boolean success indicator.

```
LONG ScanDirectory(BPTR lock)
{
  D_S(struct FileInfoBlock,fib);

  if (Examine(lock,fib)) {
    while(ExNext(lock,fib)) {
      Printf("%s\n",fib->fib_FileName);
    }
    if (IoErr() == ERROR_NO_MORE_ENTRIES)
      return DOSTRUE;
  }
  return DOSFALSE;
}
```

7.1.4 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 through a single call.

```
continue = ExAll(lock, buffer, size, type, control) /* since V36 */
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. The `buffer` shall be aligned to a word boundary. It is filled by a linked list of `ExAllData` structures, see below for their elements. The `type` argument determines which elements of the `ExAllData` is filled, and `size` is the byte size of the buffer into which such structures will be filled. It should be large enough to hold at least one such structure, a file name and a comment.

The `lock` shall be a lock on the directory to be examined. It shall not be `ZERO`.

To start a directory scan, first allocate a `ExAllControl` structure through `AllocDosObject()`, see **??** and the example below. This structure looks as follows:

```
struct ExAllControl {
  ULONG        eac_Entries;
  ULONG        eac_LastKey;
  UBYTE       *eac_MatchString;
  struct Hook *eac_MatchFunc;
};
```

`eac_Entries` is provided by the file system upon returning from `ExAll()` and then contains the number of entries that were 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 element shall not be interpreted nor modified in any way. This element shall be set to zero upfront a scan. As it is already initialized by `AllocDosObject()`, re-initialization is only necessary if the `ExAllControl` structure is re-used for another scan.

`eac_MatchString` filters the directory entry names, and fills only those into `buffer` that match the wildcard pointed to by this element. This entry shall be either `NULL`, or a pre-parsed pattern as generated by `ParsePatternNoCase()`, see section **??** for more details on this function.

`eac_MatchFunc` is a even more flexible option to filter directory entries. It shall be either `NULL` or point to a `Hook` structure 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 hook structure being used for the call, register `a1` to the data buffer that is part of the `buffer` supplied by the caller of `ExAll()` and which is already filled in with a candidate `ExAllData` structure to be checked for acceptance. 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 provisionally filled 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 elements are NULL, all directory 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;
    UWORD   ed_OwnerUID;
    UWORD   ed_OwnerGID;
};
```

The elements of this structure are as follows:

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

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

ed_Type identifies the type of the entry. It identifies directory entries according to table **??** in section **??** 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 **??** in section **??** 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 **??** defines these elements more rigorously.

ed_Comment points to a potential comment of the file system element and therefore corresponds to fib_Comment.

ed_OwnerUID and ed_OwnerGID contain potential user and group IDs if the file system is able to provide such information. The FFS stores such information on disk, but does not check it or makes any other use of it.

Which elements of the ExAllData structure are filled in is selected by the type argument. It shall be selected according to table **??**. The values are defined in dos/exall.h:

Table 7.3: Type Values

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

The return code `continue` of `ExAll()` 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 new lock on the same directory.

The return code `continue` is `DOSFALSE` in case the scan result fit entirely into `buffer` or in case an error occurred. The `IoErr()` function then provides an error code identifying the reason for the termination. If this error code is `ERROR_NO_MORE_ENTRIES`, then `ExAll()` terminated because all entries have been read and scanning the directory completed. The error code `ERROR_BAD_NUMBER` is generated in case the `type` argument passed in is not supported by the file system. Additional errors can be generated if the file system is corrupt or the medium is not accessible. If `ExAll()` terminates either because the end of the directory has been reached, or an error had been detected, `ExAllEnd()` shall not be called.

Not all file systems support `ED_OWNER`; while the FFS supports it, it does not interpret its value. 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 target file system.

The following example code lists the directory contents through `ExAll()`; for the purpose of demonstrating filtering, the matcher here removes all soft links from the listing.

```
ULONG __asm Matcher(register __a0 struct Hook *h,
                    register __a2 LONG *type,
                    register __a1 struct ExAllData *ead)
{
  if (ead->ed_Type != ST_SOFTLINK)
    return DOSTRUE;
  return DOSFALSE;
}

LONG ExamineDirectory(BPTR lock)
{
  struct ExAllControl *eac;
  struct ExAllData    *ead;
  static const LONG buffersize = 2048;
  struct Hook hk;
  LONG result = ERROR_NO_FREE_STORE;

  hk.h_Entry = (ULONG (* )())&Matcher;

  if ((ead = AllocVec(buffersize,MEMF_PUBLIC))) {
    if ((eac = AllocDosObject(DOS_EXALLCONTROL,NULL))) {
      BOOL cont;
      eac->eac_LastKey   = 0;
      eac->eac_MatchFunc = &hk;
      do {
        struct ExAllData *ed = ead;
        cont   = ExAll(lock,ead,buffersize,ED_NAME,eac);
        result = IoErr();
        while(eac->eac_Entries) {
          Printf("%s\n",ed->ed_Name);
```

```
          ed = ed->ed_Next;
          eac->eac_Entries--;
        }
      } while(cont);
      if (result == ERROR_NO_MORE_ENTRIES)
        result = 0;
      FreeDosObject(DOS_EXALLCONTROL,eac);
    }
    FreeVec(ead);
  }

  return result;
}
```

Unfortunately, the `ExAll()` implementation had defects. In AmigaDOS version 36 where this function was introduced, `eac_MatchString` was used incorrectly by the RAM-Handler which also failed to initialize `ed_Next` properly. Similar issues existed in the fall-back implementation of version 36 of the *dos.library*. The FFS did not fill in `ed_Comment` correctly prior to version 39, and the FFS flavors supporting directory caches broke `ExAll()` up to release 40 where the defect of older FFS versions is addressed by `SetPatch`. AmigaDOS versions up to version 37 cannot handle `ED_OWNER` and then fail with the error code `ERROR_BAD_NUMBER`. As a workaround, a smaller value should be passed as `type` if this error is detected. The fallback implementation within the *dos.library* up to version 45 did not provide useful arguments to the error requester in case a failure was detected. `SetPatch` also provides here a workaround for earlier versions.

There was no possibility to abort a running directory scan prior to version 39 of AmigaDOS which introduced `ExAllEnd()` (see section **??**). Unfortunately, the fallback implementation of this function in the *dos.library* in case the file system does not support abortion is corrupt in currently all known AmigaDOS versions, including the patch included in `SetPatch`.

Of all the AmigaDOS file systems, only the directory-cache enabled flavors of the FFS, see table **??** in section **??**, and the RAM-Handler profit noticeably from this function. Thus, despite the well-motivated attempt to provide a better design for a directory scanner, `ExAll()` should probably be better left alone.

7.1.5 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) /* since V39 */
          D1     D2     D3    D4     D5

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

This function aborts an `ExAll()` driven directory scan before it terminated due to an error or due to reaching the end of the directory. That is, `ExAllEnd()` shall be called whenever a scan is to be aborted even though `ExAll()` returned a non-zero result code.

`ExAllEnd()` is also be the fastest way to terminate a running directory scan, for example of a network file systems where the scan can proceed offline on a separate server. The arguments to `ExAllEnd()` shall be exactly those supplied to the `ExAll()` call 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.

This function first attempts to terminate a directory scan through the file system; however, if the file system performing a scan does not support abortion directly, the *dos.library* provides a fall-back implementation.

Unfortunately, even the most recent implementation of this fall-back function is corrupt and not reliable in all situations.

This function will modify `IoErr()` and sets it to an undefined value.

7.2 Modifying Directory Entries

While the functions in section **??** read directory entries, the functions listed in this section modify the directory and its entries, that is, they delete entries, or adjust metadata of file system objects.

7.2.1 Delete 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 **??**), and the object must not be locked (see section **??**) or accessed through a file handle[2]. To be able to delete a directory, the directory also needs to be empty.

```
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 to an error code on failure and an undefined value on success.

7.2.2 Rename or Relocate an Object

The `Rename()` function changes the name of an object, or relocates it from one directory to another provided both directories are on the same volume.

```
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 location 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 an error code on failure or to an undefined value otherwise.

Versions 34 and below of the OFS allowed to rename directories into itself, and thus made them vanish completely without, actually, deleting them. This was fixed in version 36.

All versions of AmigaDOS up to version 47 cannot handle soft links in the old or new path and instead fail with `ERROR_IS_SOFT_LINK` if one is detected.

[2]This typically implies that a file system internal lock is held on the object, thus is equivalent to the first condition.

7.2.3 Set the File Comment

The `SetComment()` function sets the comment of a file system object, i.e. a file or directory, provided the file system supports comments.

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

BOOL SetComment(STRPTR, STRPTR)
```

This function sets or replaces 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.

Whether or to which degree soft or hard links support comments is up to the file system. For the FFS and the RAM-Handler, links can carry their own comments that are separate from the comment on the corresponding link target.

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

7.2.4 Setting Protection Bits

The `SetProtection()` function modifies the protection bits of a file system object, i.e. either a file or a directory.

```
success = SetProtection( name, mask )
D0                       D1    D2

BOOL SetProtection(STRPTR, LONG)
```

This function sets the protection bits of the file system object identified by the path `name` to the combination given by `mask`. The protection bits are defined in `dos/dos.h` and their function is listed in table **??** of section **??**. The `mask` value corresponds to what `Examine()` returns in the `FileInfoBlock` structure in `fib_Protection`, see also section **??**. In particular, remember that the four least significant bits are shown inverted by Shell commands and the Workbench.

The current versions of the AmigaDOS file systems allow changes of the protection bits of locked or open files.

It is file system dependent whether links carry protection bits, even though they are there not very useful as upon locking an object through a link resolves the link and thus allows only to retrieve the protection bits of the link target. The FFS and the RAM-Handler mirror any protection bits of a hard link target in the link. For soft links, the RAM-Handler supports a separate set of protection bits.

This function returns a non-zero result code on success, or zero on error. In either case, `IoErr()` is altered, either to an undefined value on success or to an error code otherwise.

7.2.5 Set the Modification Date

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

```
success = SetFileDate(name, date) /* since V36 */
D0                    D1    D2

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

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

For links, it is file system dependent whether a link carries a modification date different from the modification date of the link target. The Amiga DOS Fast File System and the RAM-Handler synchronize the modification date of a hard link with that of the link target. For soft links, the FFS updates the link target if the modification date of link is updated, whereas the RAM-Handler keeps a separate date for the link. The modification date of a link is, however, of limited use as the `Lock()` function implicitly resolves links and thus cannot gain access to the link itself.

This function returns 0 on error or non-zero on success. In either case, `IoErr()` is set, either to an undefined value 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 refuse to set the modification date if an object is exclusively locked, this is unfortunately not handled consistently.

Even though AmigaDOS releases 34 and earlier supported setting the modification date of file system objects through the packet interface, see section **??**, a convenient interface on the level of the *dos.library* did not exist.

7.2.6 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 offer some multi-user capability, this is not the case. User and group ID are pure metadata that are returned by the functions discussed in section **??**. AmigaDOS has no concept of the current user of a file system and thus cannot decide whether a user is privileged to access an object on a file system. While versions 43 and later of the FFS support this function, the RAM-Handler does not.

```
success = SetOwner( name, owner_info ) /* since V39 */
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 an undefined value on success or to an error code otherwise.

7.3 Working with Paths

The *dos.library* contains a couple of support functions that help working with paths, see also chapter **??**. What is different from many other functions in the library 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 (see section **??**) as the path is never communicated to a file system and thus never converted to a BSTR. 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 need to follow these conventions.

7.3.1 Find the Path From a Lock

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

```
success = NameFromLock(lock, buffer, len) /* since V36 */
D0                     D1     D2    D3

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

This function constructs in buffer an absolute path that identifies the object locked by lock. This lock remains usable after the call. 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.

The returned path is not necessarily identical to the path through which lock has been obtained; first, the created path is always absolute, and second, if links were involved in the original path, the created path refers to the link target(s) and not the link itself.

If the path cannot be constructed due to an error, success is also set to 0 and IoErr() is set to an error code. In such a case, the buffer content is not useful. In case of 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. The ZERO lock is resolved into the string SYS:, which is not necessarily correct if the SYS assign or the default file system of the calling process had been modified.

7.3.2 Find the Path from a File Handle

The NameFromFH() function constructs a path name from a file handle, i.e. it finds a path that is suitable to open the file corresponding to the passed in file handle.

```
success = NameFromFH(fh, buffer, len) /* since V36 */
D0                   D1    D2    D3

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

This function takes a file handle in fh and from that constructs an absolute path of the file corresponding to fh in the supplied buffer capable of storing len bytes, including a terminating NUL byte. This function is also able to construct a path if the file is opened for exclusive access.

The constructed path is not necessarily identical to the path through which the file was opened; first, the created path is always absolute, and second, if links were involved in the original path, the created path refers to the link target(s) and not the link itself.

On success, the function returns a non-zero return code and sets IoErr() to 0. On error, it returns 0 and sets IoErr() to an error code. In particular, if the supplied buffer is not large enough, it is set to ERROR_LINE_TOO_LONG. Even in the latter case, the created path is NUL terminated, though is not useful.

7.3.3 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 interprets them according to the rules explained in chapter **??**; the colon identifies the root of the volume, and a leading slash the parent directory which removes the trailing component of the directory path.

```
success = AddPart( dirname, filename, size ) /* since V36 */
D0                  D1        D2      D3

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

This function attaches to the directory 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 file systems and does not check whether the paths passed in correspond to accessible objects. The output path is constructed purely based on the AmigaDOS path syntax. Version 36 of AmigaDOS did not handle leading slashes and absolute paths in `filename` correctly, this was fixed in release 37.

7.3.4 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 (or link) 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 ) /* since V36 */
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()`.

7.3.5 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 forms a path that corresponds to the directory containing the last component of the original 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 ) /* since V36 */
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 `FilePath()` is that the latter advanced over a trailing slash if it present. That is, if the last character of the input path of `PathPart()` is a slash, then `PathPart()` would return a pointer to this slash, but `FilePart()` would advance beyond this slash. Therefore, 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.

7.3.6 Extract a Component From a Path

The `SplitName()` function extracts a component starting at a given offset from a path and delivers the component in a buffer. It also returns a new position at which to continue parsing the path for the next component. By iteratively calling `SplitName()`, a path can be resolved directory by directory, walking the file system tree from the root to the leaves. Section **??** provides an example how to use it for iterating through a path.

```
newpos = SplitName(name, separator, buf, oldpos, size) /* since V36 */
D0                 D1      D2      D3     D4     D5

WORD SplitName(STRPTR, UBYTE, STRPTR, WORD, LONG)
```

This function scans a path as given by `name` starting from position `oldpos`. It copies all characters starting from this position into the buffer `buf` which is `size` bytes large, terminating either at the end of the path, or at `separator`, or when `buf` runs full. The component string constructed in `buf` is `NUL`-terminated in either case. If the provided separator is found, the separator is not copied into `buf`.

If no separator is found, the function returns -1 as `newpos` indicating that the entire path has been scanned. Otherwise, it returns the offset into `name` at which the next component starts, i.e. the offset behind the found separator. The return value is also valid in case the found component was too large to fit into `buf` and had to be truncated.

This function does not set `IoErr()`, even in case `buf` was too small.

The intended purpose of this function is to walk a path component by component, extracting the components as scanning proceeds. That is, if result code `newpos` is not negative, it should be passed back into this function as `oldpos` for a follow-up scan which then extracts the next component of the path. Typical users of this function are file systems that locate objects in the file system tree. For most (if not all) AmigaDOS file systems the separator is therefore the forwards slash ("`/`"). Another use case is to identify the device name of an absolute path by splitting it at the colon ("`:`").

In AmigaDOS version 37 and below, components larger or equal than `size` characters terminated by a separator are truncated one character too early, causing the last character to be lost. The returned index then points at the separator rather than behind it. This was fixed in release 39.

7.4 Links

Links escape the tree-like hierarchy of directories, sub-directories and files. A *link* mirrors an object of a file system to another location such that if the object is modified through 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 remain always identical. The storage for the file (or directory) content 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 the link target, the change also appears in the link and vice versa.

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 objects on the level of the file system, and thus the two objects must be located on the same volume. That is, whenever a link is accessed, the file system resolves the link, transparent to its user and transparent to the *dos.library*. While for the Amiga Fast File System and the RAM-Handler a hard link is a distinct directory entry type, some other file systems do not distinguish between the original object and a hard link to it. For such file systems, the same file content is just referenced by two directory entries. If the target 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 an internal reference counter is decreased, and the file content or directory is removed only if this counter indicates that no further references exist.

Soft links work differently and can also be established between file systems, or between different volumes. Here, the soft link is a type of its own that contains the path of the referenced object. Unlike hard links, soft links are resolved through an interaction of the file system and the *dos.library*. Due to defects in the Fast File System, soft links did not work correctly prior to AmigaDOS version 45, and soft links containing two components of which the first (directory) component is a soft link do not work correctly in the RAM-Handler even in its latest version. Unfortunately, soft links accessed through a multi assign also fail to work correctly due a defect even in the latest version of the *dos.library*.

The *dos.library* supports soft links through the functions listed in Table **??**:

Table 7.4: Soft link aware functions

Function	Purpose
`CreateDir()`	Create a directory
`DeleteFile()`	Delete an object on a file system
`MakeLink()`	Create a link to an object
`Lock()`	Obtain access rights to an object
`Open()`	Open a file
`SetComment()`	Modify object comment
`SetFileDate()`	Set the modification date of a file
`SetOwner()`	Set User and Group ID
`SetProtection()`	Modify protection bits
`DeleteVar()`	Delete an environment variable
`Execute()`	Execute a shell script (legacy)
`GetVar()`	Read an environment variable
`LoadSeg()`	Load an executable
`MatchFirst()`	Resolve a lock to a path[3]
`NewLoadSeg()`	Load an executable with parameters
`SetVar()`	Set an environment variable
`System()`	Execute a shell script

All of the above functions take a path as one of their arguments. If the path consists of multiple components, i.e. identifies an object in a nested directory tree, and one of the intermediate components is a soft link, the *dos.library* will automatically resolve such an intermediate link and constructs a resolved path to the link destination. Whether a soft link as the last component of a path is resolved is file system and function dependent. For example, `Open()` and `Lock()` will always resolve soft links, but `SetProtection()` may not and may instead affect the link, not the target object. Here, the Fast File System resolves soft links, whereas the RAM-Handler does not. `DeleteFile()` will never resolve a link at the final component of the path, and will therefore delete the link and not the link target.

Note that `Rename()` is currently not on the list of functions supporting soft links as part of the path to the object to be renamed, or as part of the target path. Thus, if the source or destination argument of `Rename()` contain a soft link, the function will fail. Section **??** provides source code for a (partial) workaround.

If the target of a soft link is deleted, soft links pointing to it become invalid, even though they remain in the file system. Any attempt to resolve such a link fails, and AmigaDOS does not attempt to identify invalid links upon deletion of the link target. The same issue does not exist for hard links as in such a case one of the links will take the role of the link target.

Soft link resolution works as follows: Functions of the *dos.library* create a packet of a type that corresponds to the called function, and send a packet to the handler responsible for the target volume; the packet mechanism is explained in more detail in chapter **??**, and packet types are listed in chapter **??**. If the handler

[3]This function works only partially with soft links - it fails if the pattern is not a wildcard and the soft link cannot be resolved.

addressed by the packet determines that the path provided by the user contains a soft link, it will respond with failure and the error ERROR_IS_SOFT_LINK.

The *dos.library* requests the handler to resolve the soft link via the ReadLink() function, which is explained in more detail in section **??**. It sends a packet of type ACTION_READ_LINK back to the handler to resolve the link, see section **??**. The handler then creates from the original path and the link target stored in the soft link an updated path. When the packet is returned, the updated path is received by the *dos.library*, which attempts again to perform the requested function. Details on how a file system merges a path and a soft link is provided in section **??**. The additional round-trip is unfortunately necessary as the file system addressed by the link target can be a different from the file system on which the soft link is stored.

The process of link resolution continues until either the requested action could be performed, or a maximum number of attempts failed. Currently, the *dos.library* will try at most 15 times to resolve a soft link until it finally fails. The reason for introducing such a limit is that nothing in the system prevents the user from creating a circular graph of soft links that point to each other, and the process of link resolution would never terminate.

Finally, the RAM-Handler supports a special type of hard link that goes across volumes called *external link*. Such a link copies the link target on read-access into the link, i.e. the RAM-Handler implements a *copy on access*. This feature is used for the ENV assign containing all active system settings. The 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. This process is completely transparent to the *dos.library*. External links were added to AmigaDOS in version 47.

The FileInfoBlock introduced in section **??** identifies links through the fib_DirEntryType element. As seen from table **??**, 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 the above directory entry types cannot be depended upon. In particular, external links of the RAM-Handler cannot be identified by any particular value of the fib_DirEntryType.

Table **??** also provides the fib_DirEntryType for soft links, namely ST_SOFTLINK. As the target of a soft link may not be under control of the file system containing the link, 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.

7.4.1 Creating Links

The MakeLink() function creates a hard link, soft link or external link to another file system object.

```
success = MakeLink( name, dest, soft ) /* since V36 */
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, i.e. the link target, is given by dest.

The third argument, soft, identifies the type of the link to be created. It shall be taken from table **??**, defined in dos/dos.h:

Table 7.5: Link Types

Link Types	Description
LINK_HARD	Hard link, or external link
LINK_SOFT	Soft link

If soft is LINK_HARD, dest is a BPTR to a FileLock, i.e. a lock. For most file systems, dest shall be on the same volume as the one identified by the path in name. The only exception is currently 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 file system object representing the external link will be created immediately, 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 creating intermediate directories, on attempts to access the link contents. Thus, the link becomes a mirror of the link target whenever an object within the link or the link itself is accessed. From that point on, the copied object is detached from its original, and may be overwritten without affecting the link target. External links were added to AmigaDOS in version 47.

If soft is LINK_SOFT, 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 **??**. In case of soft links, the target does not even need to exist. Soft links did not work reliable prior to version 45 of AmigaDOS.

The MakeLink() function returns non-zero in success 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 to an undefined value on success.

7.4.2 Resolving Soft Links

The ReadLink() function locates the target of a soft link and constructs from the path and directory containing the link a new path that resolves the link within the original path. In most cases, namely for the functions listed in table **??**, the *dos.library* calls this function itself internally and their users do not need to worry about soft links.

If, however, a *dos.library* function or a packet in direct communication with a handler (see chapter **??**) returns the error ERROR_IS_SOFT_LINK, then this function helps to resolve a soft link within the path. The access is then typically retried with an updated path constructed by this function. Note well that the updated path can contain yet another soft link, requiring recursive resolution of the link. To avoid endless recursion, resolving links should be aborted after a maximum number of attempts, and should then generate the error ERROR_TOO_MANY_LEVELS. A suggested maximum level of nested soft links, also used by the *dos.library*, is 15 links.

```
success = ReadLink( port, lock, path, buffer, size) /* since V36 */
D0                  D1    D2    D3     D4     D5

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

This function creates an updated path in buffer from an input path whose resolution failed due to a soft link as one of its components. The input path is relative to the directory represented by lock. The output path constructed in buffer, which is size bytes large, is again relative to lock, but can also be an absolute path.

The port is the message port of the file system containing the soft link and which is requested to resolve this link. In typical cases, this port and lock are obtained from dvp_Port and dvp_Link of the DevProc structure returned by GetDeviceProc() called on path, see section **??** for this structure and the function.

Due to various inconsistencies in file system implementations, the return code is hard to interpret. If the return code is positive, the function completed with success and its value is then the string length of the updated path in buffer, not including the terminating NUL. On success, IoErr() is always set, but to an undefined value.

If size is too small to hold the adjusted path, the function should[4] return -2 and the IoErr() function indicates ERROR_LINE_TOO_LONG. Unfortunately, some file system implementations set the return code

[4]The information in [**?**] and [**?**] that the result code is a Boolean success indicator is incorrect.

erroneously to 0 or −1 in this case instead. The FFS returns 0 for versions 46 and below, and −1 for version 47 to indicate a buffer overflow.

If some other error was detected, then the function should return −1 and `IoErr()` is set to an error code identifying the source of the error; error codes are listed in section **??**. Again, some file systems erroneously return 0 to indicate an error, for example FFS versions 46 and below.

Due to these problems, it is recommended to interpret all positive values as success, and 0 and negative values as failure. To identify the cause of the failure in the latter case, the error code from `IoErr()` should be checked, and if it is `ERROR_LINE_TOO_LONG`, then the buffer size needs to be increased and the function should be called again with the enlarged buffer.

Section **??** provides further details on the file system side of the implementation of this function, in particular how to combine the `path` with the link target stored in the soft link.

The following source demonstrates how to use `ReadLink()` to implement a `Rename()` function that accepts source paths that contain soft links. Unlike the implementation in the *dos.library*, this code also works correctly in case the soft link is accessed through a multi-assign:

```
#define BUFSIZE 256
#define MAX_LINKS 15

LONG RenameWithLinks(UBYTE *srcpath,UBYTE *dstpath)
{
  UBYTE *buf         = NULL;
  UBYTE *newbuf      = NULL;
  ULONG bufsize      = BUFSIZE;
  struct DevProc *dp = NULL;
  LONG levels        = MAX_LINKS;
  LONG error         = 0;
  LONG success;

  while (!(success = Rename(srcpath,dstpath)) &&
           IoErr() == ERROR_IS_SOFT_LINK) {
    /* If too many levels passed, fail */
    if (--levels < 0) {
      SetIoErr(ERROR_TOO_MANY_LEVELS);
      break;
    }
    /* Allocate a target buffer for the new name */
    if (!(newbuf = AllocVec(bufsize,MEMF_PUBLIC)))
      break;
    /* Find the device and lock                  */
    do {
      if ((dp = GetDeviceProc(srcpath,dp))) {
        /* Ok, attempt to resolve the link */
        if ((success = ReadLink(dp->dvp_Port,dp->dvp_Lock,
                                srcpath,newbuf,bufsize)) > 0) {
          /* rotate buffers */
          srcpath  = newbuf;
          newbuf   = buf;
          buf      = srcpath;
          error    = 0;
          break;
```

```
        } else {
          error = IoErr(); /* get the error */
          /*
          ** Check whether the supplied buffer was too short.
          ** Redo with larger buffer next round.
          */
          if (error == ERROR_LINE_TOO_LONG) {
            bufsize <<= 1;
            success   = 1;
            break;
          }
          /* If the error was ERROR_OBJECT_WRONG_TYPE or
          ** ERROR_OBJECT_NOT_FOUND, then the source is no link,
          ** or the source was a multi-assign and the source was
          ** not contained in the assign, so re-iterate potentially.
          ** Unfortunately, RAM does not return the correct error.
          */
          if (error != ERROR_OBJECT_WRONG_TYPE &&
              error != ERROR_OBJECT_NOT_FOUND)
            break;
          /* Make this a more sensible error */
          error = ERROR_IS_SOFT_LINK;
        }
      } else {
        /* Get the error why GetDeviceProc() failed */
        error = IoErr();
        break;
      }
      /* Retry if the path is a multi-assign */
    } while(dp && (dp->dvp_Flags & DVPF_ASSIGN));
    /* Release the DeviceProc, NULL is fine  */
    FreeDeviceProc(dp);
    dp = NULL;
    /* Release the old or the new buffer */
    if (newbuf)
      FreeVec(newbuf);
    /* Fixup or reset the error          */
    SetIoErr(error);
    /* Abort if link resolution failed   */
    if (success <= 0)
      break;
  }

  /* Release the temporary/last buffer */
  if (buf)
    FreeVec(buf);

  return success;
}
```

The above code does not check whether the target goes into a soft linked directory — this would need to be added.

7.5 Notification Requests

Notification requests allow programs to monitor file or directory changes. If a change is detected, either a signal or a message can be send to a specific task or port. Notification requests were added in version 36 of AmigaDOS, and while the RAM-Handler supported it right from the start, the Fast File System included support for it from version 37 onward. However, be aware that not all file systems implement this mechanism; network file systems, for example, need to operate on the basis of existing protocols that do not provide a mechanism to monitor file or directory changes on the server side.

If issued on a file, the notification request is only triggered after the modified file is closed to avoid sending too many requests at once for each single change made.

If issued on a directory, attempts to add or remove files or links will trigger the request. Renaming files within the monitored directory are also recognized and can therefore be monitored. Whether changes of metadata such as protection bits or comments are considered modifications is not clearly defined and not all versions of all AmigaDOS file systems handle such cases. The most recent version of AmigaDOS will consider them sufficient to trigger a notification.

A typical user of notification requests is the `IPrefs` program which uses them to monitor changes of the preferences files. If it detects a changed preferences definition, it reloads the contents of the affected settings file and re-installs the preferences into the components it serves, most importantly the *intuition.library*.

7.5.1 Request Notification on File or Directory Changes

The `StartNotify()` function starts monitoring a file or directory for changes, and if such modifications are found, a signal is send to a task or a message is put into a port.

```
success = StartNotify(notifystructure) /* since V36 */
D0                          D1

BOOL StartNotify(struct NotifyRequest *)
```

This function starts a notification request as described by the `notifystructure` argument. This structure shall be initialized by the caller, and is then enqueued in the file system until the notification request is canceled by `EndNotify()`. Once issued, the structure shall not be touched anymore as the file system can access it any time. As some elements require zero-initialization, it is advisable to allocate it from the exec memory pool with the `MEMF_CLEAR` flag set.

The `NotifyRequest` structure is defined in `dos/notify.h` and reads as follows:

```
struct NotifyRequest {
    UBYTE *nr_Name;
    UBYTE *nr_FullName;
    ULONG nr_UserData;
    ULONG nr_Flags;
    union {
        struct {
            struct MsgPort *nr_Port;
        } nr_Msg;

        struct {
           struct Task *nr_Task;
           UBYTE nr_SignalNum;
           UBYTE nr_pad[3];
        } nr_Signal;
```

```
    } nr_stuff;
    ULONG nr_Reserved[4];
    /* internal use by handlers */
    ULONG nr_MsgCount;
    struct MsgPort *nr_Handler;
};
```

The elements of this structure shall be initialized as follows:

`nr_Name`: The path to the object to be monitored, relative to the current directory. While it seems plausible to issue a notification request on an object that does not yet exist to get notified once it is created, this feature is currently not supported by AmigaDOS.

The file system will neither inform notification requests when an assign is created to a monitored directory, nor if the volume containing the object is removed, reinserted or access to it is inhibited (see section **??**).

`nr_FullName` is initialized by the *dos.library* for the file system and shall be left alone by the caller. It stores the full path of the object to monitor. This element is supposed to be used by the file system only, see section **??**.

`nr_UserData` is free for the calling application. It may be used to distinguish multiple notification requests that have been issued in parallel. This is most useful for NRF_SEND_MESSAGE requests as the `NotifyMessage` structure contains a pointer back to the request.

`nr_Flags` defines how and when the issuer of a notification request shall be be informed by the monitoring file system. Currently, the following flags are defined in `dos/notify.h`:

Table 7.6: Notification Flags

Flag	Purpose
`NRF_SEND_MESSAGE`	Send a message on a file system change
`NRF_SEND_SIGNAL`	Set a signal on a change
`NRF_WAIT_REPLY`	Wait for a reply before notifying again
`NRF_NOTIFY_INITIAL`	Notify immediately when queuing the request

All other bits are currently reserved. In specific, bits 16 upwards are free for the file system to use and shall not be set by the caller.

The flags `NRF_SEND_MESSAGE` and `NRF_SEND_SIGNAL` are mutually exclusive. Exactly one of the two shall be included in the request to identify the activity that is performed when the monitored object changes. As the names suggest, in the first case a message is send to `nr_Port`, and in the second case a signal bit is set in the task `nr_Task`.

`NRF_WAIT_REPLY` indicates to the file system that it should not continue to send a notification message until the last one send has been replied. Thus, setting this flag prevents notification requests to pile up at the recipient. However, if one or multiple changes were detected while the first request was triggered but not yet responded, replying to this first notification message will immediately trigger a *single* subsequent request for the file system object monitored.

`NRF_NOTIFY_INITIAL` instructs the file system to trigger a notification message or signal immediately after the request has been issued. This allows applications to roll both the initial action and the response of the notification into a single function — for example, for reading an initial version of a file or update from a modified version of it.

`nr_Port` is only used if the `NRF_SEND_MESSAGE` flag is set in `nr_Flags`. It points to a `MsgPort` structure to which a `NotifyMessage` is send when a change has been detected. This structure is specified at the end of this section.

nr_Task and nr_SignalNum are only used if the NRF_SEND_SIGNAL flag is set in nr_Flags. nr_Task is a pointer to the Task that will be signaled, and nr_SignalNum the bit number of the signal that is set. It is not a bit mask. Clearly, NRF_WAIT_REPLY does not work in combination with signal bits.

nr_Pad is only present for alignment and shall be left alone.

nr_Reserved shall be zero-initialized; it is reserved for future extensions.

nr_MsgCount shall not be touched by the caller and is reserved purely for the purpose of the file system, see section **??**. It is there used to count the number of messages that have been send out to the client, but have not yet been replied. The client, i.e. the caller, shall not interpret or modify this element.

nr_Handler shall neither be touched by the caller; it is used by AmigaDOS to store the MsgPort of the file system responsible for this notification request, and in particular, the file system to contact for ending a notification request.

If NRF_SEND_MESSAGE is set, then the file system sends a NotifyMessage to nr_Port upon detection of a change of the monitored object; this structure is also defined in dos/notify.h and looks as follows:

```
struct NotifyMessage {
    struct Message nm_ExecMessage;
    ULONG  nm_Class;
    UWORD  nm_Code;
    struct NotifyRequest *nm_NReq;
    ULONG  nm_DoNotTouch;
    ULONG  nm_DoNotTouch2;
};
```

nm_ExecMessage is a standard exec message as documented in exec/ports.h.

nm_Class is always set to NOTIFY_CLASS, also defined in dos/notify.h, to identify this message as a notification message.

nm_Code is always set to NOTIFY_CODE, again defined in dos/notify.h. It may also be used to identify a notification message.

nm_NReq is a pointer to the NotifyRequest through which this message was triggered. This allows clients to identify the source of the request and by that the object that has been changed.

nm_DoNotTouch and nm_DoNotTouch2 are strictly for use by the file system and shall not be touched nor interpreted by the issuer of the request.

By design, the NotifyMessage reassembles the layout of an IntuiMessage and thus allows reusing an IDCMP port (see [**?**]) of an intuition window for receiving messages — as long as it can be ensured that the port remains available as least as long as notification request remains active.

StartNotify() returns a Boolean success indicator. It returns a non-zero result code on success and then sets IoErr() to an undefined value. On error, the function returns 0 and sets IoErr() to a non-zero error code.

7.5.2 Canceling a Notification Request

The EndNotify() function cancels an issued notification request.

```
EndNotify(notifystructure) /* since V36 */
                 D1

void EndNotify(struct NotifyRequest *)
```

This function cancels the notification request identified by `notifystructure`. This function shall only be called on notification requests that have been successfully issued by `StartNotify()`[5]. If the issuer of a request did not yet reply all `NotifyMessage` messages and some are still piled up in the `nr_Port`, the file system will manually dequeue them from `nr_Port`. However, callers should make sure that they have replied all `NotifyMessage`s they already dequeued from their port themselves before terminating the request.

Afterwards, the `notifystructure` is again available for the caller, for example to either release its memory, or to start another notification request.

[5]The information in [**?**] on this is incorrect, `EndNotify()` is *not* safe to call on requests that failed to start.

Chapter 8

Administration of Volumes, Devices and Assigns

To the *dos.library*, a handler or a file system is represented by an exec `MsgPort` (or short "port") to which it sends requests to the handler, e.g. to read bytes from a file or to iterate through a directory (see also chapter **??** and section **??**). The library, however, also needs to obtain these ports for a given path somehow; this is the responsibility of the `GetDeviceProc()` function discussed in section **??**.

For relative paths, the lock representing the current directory, or the default file system of the process if this lock is `ZERO`, is used as source for such a port, see section **??**, and chapter **??** for details. For absolute paths, the source is the device, volume or assign name upfront the colon in the path. For example, if the file "`S:Startup-Sequence`" is to be opened, `GetDeviceProc()` needs to determine from the name "`S`" of the assign the port of the handler that is responsible for the target directory of the assign.

To this end, each volume, assign, handler or file system is represented by a `DosList` structure which provides both the name as used in an absolute path, e.g. `S`, and the `MsgPort` responsible for it. All `DosList` structures are queued in a global list within the *dos.library* (see also section **??**) named the *device list*. The `GetDeviceProc()` function, when called with an absolute path, searches this list for a matching name and provides a suitable port for the caller. Thus, the relation between a name and a port is established through this function and the *Device List*.

Entries on the device list are created in multiple ways: The `Mount` command adds entries representing handlers and file systems from a mountlist stored typically in `DEVS:DosDrivers`, see section **??**. Any non-ROM resident handler, such as for example `CD0` for the ISO Rock Ridge file system is announced to AmigaDOS in this way. Some devices, such as `DF0` for the first floppy drive are created by the ROM itself through the System-Startup process as explained in section **??**, or — in AmigaDOS versions 45 and before — through the Initial CLI and the *dos.library*.

The `Assign` command creates assigns, i.e. logical volume names bound to one or multiple directories, which utilizes the functions listed in section **??** and by that also creates a `DosList` structure representing the assign. Some assigns, namely those listed in table **??** of section **??**, are also already created by the System-Startup process when booting.

File systems itself also announce their volumes, i.e. the media, data carriers and partitions they handle, through entries in the device list. `DosList` structures representing volumes are added and removed by file systems as media are inserted or removed.

Finally, auto-booting hardware expansions also mount their boot partitions here through the *expansion.library* and its `MakeDosNode()` and `AddBootNode()` functions. The System-Startup process picks them up from the *expansion.library* and adds them to *dos.library* when booting the system. This mechanism is explained in more detail in [**?**], and AmigaDOS specific details on the boot process are found in section **??**.

Some special names such as NIL or * from table **??** in **??** are hard-coded into GetDeviceProc() and are not represented by a DosList structure. They form an exception because the relation between handler and device name is dependent on the calling process, or the target device is not representable as a handler at all.

Thus, in short, the DosList structure is central to the *dos.library* and establishes the link between the assign, volume or device name as it appears on an absolute path upfront the colon (":") and the MsgPort of the handler or file system implementing AmigaDOS functions on such a path. This structure, defined in dos/dosextens.h, reads as follows:

```
struct DosList {
    BPTR                dol_Next;
    LONG                dol_Type;
    struct MsgPort     *dol_Task;
    BPTR                dol_Lock;
    union {
      struct {
        BSTR    dol_Handler;
        LONG    dol_StackSize;
        LONG    dol_Priority;
        ULONG   dol_Startup;
        BPTR    dol_SegList;
        BPTR    dol_GlobVec;
      } dol_handler;

      struct {
        struct DateStamp        dol_VolumeDate;
        BPTR                    dol_LockList;
        LONG                    dol_DiskType;
      } dol_volume;

      struct {
        UBYTE             *dol_AssignName;
        struct AssignList *dol_List;
      } dol_assign;

    } dol_misc;

    BSTR                dol_Name;
};
```

Confusingly, the same data is also accessible through the DeviceList structure, which is nothing than a DosList in disguise, but only represents volumes, with elements at the same byte offsets under different names. Thus, for example dl_VolumeDate in the DeviceList structure accesses the same elements as dol_volume.dol_VolumeDate and the two may be used interchangeably for volumes.

The DevInfo structure is another incarnation of a DosList that is only able to represent handlers; e.g dvi_Handler and dol_handler.dol_Handler access the same element. The DevInfo and the DeviceList structures are defined in dos/dosextens.h.

Finally, dos/filehandler.h contains a third structure, named DeviceNode, which is a fourth representation of a DosList, and only to be used for file systems, and thus equivalent to a DevInfo structure as well. For example, dn_Handler and dvi_Handler are equivalent elements, just available under different names.

In the following, these alternative structures will not be discussed as they offer no new functionality and just add confusion. It is suggested to just leave them alone and access entries of the device list consistently through the DosList structure. If necessary, pointers to these structures can be casted to each other without loss of information[1].

The elements of a DosList have the following interpretation:

dol_Next is a BPTR to the next entry in the singly linked list of DosList structures forming the device list, or ZERO for the last entry on the list. The order of the entries has no particular meaning. The head of this list is in the DosInfo structure specified in section **??**. However, this list should not be walked manually, but instead FindDosEntry() (see **??**) 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 elements of the unions are used. The following types are defined in dos/dosextens.h:

Table 8.1: *DosList* Entry Types

dol_Type	Description
DLT_DEVICE	A file system or handler, see **??**
DLT_DIRECTORY	A regular or multi-assign, see **??**
DLT_VOLUME	A volume, see **??**
DLT_LATE	A late binding assign, see **??**
DLT_NONBINDING	A non-binding assign, see **??**
DLT_PRIVATE	Used internally for resolving late binding assigns

dol_Task is the MsgPort of the handler to contact for the particular device, assign or volume name. If the DosList represents a handler or file system, this element can be NULL if the handler is not running. A new handler process will be started if a port is needed, and it depends on the handler whether this element will ever be populated. The CON-Handler leaves this element at NULL as it requires a new process for each window it manages, but file systems such as the FFS deposit here its MsgPort as all activities on the same medium or partition go through the same single process.

Volumes keep here the MsgPort of the file system that operates the volume, but set it to NULL in case the medium goes away, e.g. is ejected, but locks or file handles on the volume are still active and thus the volume itself remains known to AmigaDOS. As a result, the Workbench keeps showing the icon representing such volumes on the desktop.

For regular assigns, dol_Task is also the pointer to the port of the file system the assign binds to; in case the assign is a multi-assign, this is the port of the first target directory. All additional ports can be reached through the locks that are are part of the dol_List discussed below. For late binding assigns this element is initially NULL, but will be filled in as soon as the assign is bound to a particular directory; it then becomes the pointer to the port of the handler managing the volume the assign is located on. Finally, for non-binding assigns this element always stays NULL.

dol_Lock is only used for assigns, and only if it is bound to a particular directory. It is then the lock (see chapter **??**) to the directory forming the assign, or for multi-assigns, the first directory within the assign. This element remains ZERO for non-binding assigns and is initially ZERO for late binding assigns. For all other types, this element stays ZERO.

dol_Name is a BPTR to a BSTR of the name under which the handler, volume or assign is accessed. That is, this string corresponds to the path component upfront the colon. In particular, it *does not include* the colon. As a courtesy to C and assembler functions, AmigaDOS ensures that this string is also NUL terminated, i.e. dol_Name+1 is a regular C string whose length is available in dol_Name[0].

[1]The reason for this confusion lies likely again in the history of AmigaDOS actually being a port of Tripos, written in BCPL. This language represent structures through long word offsets relative to a base index in manifests, and thus does not offer structures as known from C.

The elements within dol_handler are used by handlers and file systems, that is if dol_Type equals DLT_DEVICE:

dol_Handler is a BPTR to a BSTR containing the file name from which the handler or file system is loaded if dol_SegList is ZERO. As the file name is interpreted by the first process attempting to access an unloaded handler, and this path is thus relative to the current directory of this process, it should be an absolute path. Typically, handlers reside in the L assign. Clearly, the path should not be on a file system described by its own DosList structure. This element corresponds to the HANDLER, FILESYSTEM and EHANDLER entries of the mountlist, see section **??**.

dol_StackSize specifies the size of the stack for creating the handler or file system process. The unit of the stack size depends on the dol_GlobVec entry. If dol_GlobVec indicates a C or assembler handler, dol_StackSize is in bytes. Otherwise, that is, for BCPL handlers, it is in long words[2]. Table **??** indicates the type of the handler. This element corresponds to the STACKSIZE entry of the mountlist.

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 mountlist.

dol_Startup is a handler-specific startup value that is used to communicate a configuration to the handler. For example, the CON-Handler uses this element to distinguish whether it is used as CON or RAW device (see section **??** for details). The Fast File System and most other file systems expect here a BPTR to a FileSysStartupMsg structure. While this value may be whatever the handler requires, the Mount command either deposits here a BPTR to the FileSysStartupMsg structure, a BPTR to a BSTR, or a small integer. Section **??** provides more details on mounting handlers and how the startup mechanism works. Unfortunately, it is hard to interpret dol_Startup correctly as its interpretation is dependent on the specific handler, though section **??** provides additional hints and a recommended algorithm for guessing the type of this element. One way to set it is through the STARTUP keyword in the mountlist, see **??** for details.

dol_SegList is a BPTR to the chained segment list of the handler if it is loaded, see chapter **??** for additional details how binaries are represented in memory. For disk-based handlers, this element is initially ZERO. When a program attempts to access an object identified by an absolute path starting with the device name dol_Name, the *dos.library* first checks whether this element is ZERO, and if so, attempts to load a binary from the path stored in dol_Handler. Upon success, its segment is stored here. If the FORCELOAD entry of the mountlist is non-zero, the Mount command already attempts to load the handler or file system from disk and thus populates this element; otherwise, that is, without FORCELOAD, Mount attempts to reuse an existing file system by first scanning the *FileSystem.resource* for an entry whose DOSTYPE matches. If nothing is found there, loading the handler remains to the first process requiring it. Additional details depend on nature of the handler or file system and are discussed in section **??**.

dol_GlobVec identifies the nature of the handler as AmigaDOS (still) supports BCPL and C/assembler handlers. BCPL handlers use a somewhat more complex loading and linking mechanism as their BCPL-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 sections **??** and **??**. While at this point the Port-Handler and the FFS as only exceptions still support the legacy BCPL binding despite not being implemented in BCPL anymore, it is likely that support for BCPL handlers will be phased out in the future.

The dol_GlobVec element also defines how access to the device list is secured during handler loading and startup. 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 type of access while the handler is loaded and until handler startup completed. This prevents any other modification to the list, but also read access through any other process of the list. Shared access allows read accesses to the list while preventing modifications.

The value in dol_GlobVec corresponds to the GLOBVEC entry in the mountlist. It shall be one of the values in table **??**:

[2]The information in [**?**] is not accurate, the stack size is dependent on the GLOBVEC.

Table 8.2: GlobVec Values

dol_Type	Description
-1	C/assembler handler, exclusive access to the device list
-2	C/assembler handler, shared access to the device list
0	BCPL handler using system GV, exclusive access to the device list
-3	BCPL handler using system GV, shared access to the device list
>0	BCPL handler with custom GV, exclusive access to the device 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. The values 0 and -3 fill this vector with the system functions first, and then use the BCPL binding mechanism to extend or override entries in this vector with the values found in the loaded code, see section **??** for further details. Any values > 0 require a BPTR to a custom vector which is used instead for initializing the handler. This startup mechanism is only used for creating device list entries within the Kickstart and it is not quite practical otherwise as this vector needs to be communicated into the *dos.library* somehow. For new code, BCPL linkage and binding should not be used anymore.

Elements of the `dol_volume` structure are used if `dol_Type` is `DLT_VOLUME`, identifying this entry as belonging to a known specific medium or partition.

`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. This list is created by the file system when the volume is ejected, and contains all still active 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. The linkage is performed with BPTRs and the `fl_Link` element of the `FileLock` structure, see also section **??**. For inserted volumes, this element remains `ZERO`, and the information whether a volume is inserted or not comes from the `dol_Task` element being `NULL` or a valid pointer, see there.

`dol_DiskType` is indented to be an identifier of the file system type that operates (or operated, for ejected media) the volume. The value is placed here such that an alternative process of the same file system operating on a different exec device is able to pick up or refuse the locks stored in `dol_LockList` if a medium is re-inserted into a different physical device. Unfortunately, even the latest version of the Fast File System does not fill this element such that it remains 0, though other file systems leave their traces here to identify their own volumes. Note that the value placed here is *not* the identifier found in `id_DiskType` of the `InfoData` structure as the latter rather identifies the availability of a file system but not its type, see section **??**.

Elements of the `dol_assign` structure are used for all types of assigns:

`dol_AssignName` is pointer (and not a BPTR) to a `NUL`-terminated target path of non-binding and late binding assigns. It remains `NULL` otherwise. The *dos.library* uses this string to locate the target directory of the assign. For late binding assigns, this element is used only on the first attempt to access the assign at which `dol_Lock` is populated, then this element is released with `FreeVec()` and set to `NULL`.

`dol_List` contains additional locks for multi-assigns and is thus only used if `dol_Type` equals `DLT_DIRECTORY`. A regular assign is a multi-assign whose `dol_List` element is `NULL` and thus consists of only a single directory. For multi-assigns, `dol_Lock` is the lock to the first directory of the multi-assign, while `dol_List` is a regular C pointer to a singly-linked list of the locks of all subsequent directories, represented as an `AssignList` structures, defined in `dos/dosextens.h`:

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

al_Next points to the next lock that is part of the multi-assign. This is *not* a BPTR. It is NULL for the last directory in a multi-assign.

al_Lock is the lock to (another) directory participating in the multi-directory assign.

The AssignList structures shall not be allocated or released directly; rather, they shall be created through AssignAdd() and removed through RemAssignList(), see sections **??** and **??**.

8.1 The Device List and the Mountlist

One way to create entries of the type DLT_DEVICE on the device list is through a mountlist and the Mount command. Many keywords in the mountlist map directly to entries in the DosList structure, others to entries in the FileSysStartupMsg and the DosEnvec structure pointed to from there. The latter two structures will be introduced in sections **??** and **??**.

8.1.1 Keywords defining the DosList structure

The HANDLER, EHANDLER and FILESYSTEM keywords in the mountlist all define the dol_Handler element and thus the path from which the handler will be loaded. This is by convention a file within the L assign containing all handlers and file systems[3]. Which keyword is used impacts, however, other elements of the DosList structure, most notably the dol_Startup entry.

The STACKSIZE keyword sets dol_StackSize element. This is in bytes for C and assembler handlers, and in long words for BCPL handlers. The type of the handler is determined by GLOBVEC.

The PRIORITY keyword sets dol_Priority and with that the priority of the process running the handler or file system. Useful values are between 0 and 19.

The GLOBVEC keyword sets dol_GlobVec element, and by that also the type of the handler. Table **??** in chapter **??** lists the possible values and their interpretation. Positive values, even though a meaning is assigned to them, are not useful as an absolute RAM location for the BCPL global vector would then be needed.

If the HANDLER keyword is present, the STARTUP entry in the mountlist sets dol_Startup. It can be either an integer value, or a string, optionally enclosed in double quotes. In the latter case, dol_Startup is set to a BPTR to a BSTR which is, to ease handler implementation, also NUL terminated. If present, the quotes become part of the string, and the handler has to remove them when interpreting it. The NUL terminator is, even though always present, not included in the size of the BSTR. It is up to the user to ensure that the arguments in the mountlist are what the handler expects.

8.1.2 Keywords controlling the FileSysStartupMsg

If the EHANDLER or FILESYSTEM keyword is present, the dol_Startup element is instead a BPTR to a FileSysStartupMsg structure, defined in the include file dos/filehandler.h:

```
struct FileSysStartupMsg {
  ULONG       fssm_Unit;
  BSTR        fssm_Device;
  BPTR        fssm_Environ;
  ULONG       fssm_Flags;
};
```

[3]This L is probably short for "libraries", not to be confused with the exec type of shared libraries, and of course this convention goes back to Tripos.

It is again up to the user to ensure that the handler is really expecting such a structure and create the mountlist appropriately.

The elements of the above structure identify an exec type device on top of which the handler or file system is supposed to operate, thus for example the *trackdisk.device* to mount a file system on the floppy. Some extended handlers, i.e. those that accept the `EHANDLER` keyword, also use this structure; the V43 Port-Handler can be setup this way to operate on top of a third-party serial device driver, see section **??** for details.

The `DEVICE` keyword of the mountlist sets `fssm_Device` entry. This element is initialized to a BSTR containing the device name (without quotes). To avoid conversion to a C string when passing it to `OpenDevice()`, this BSTR is also always `NUL`-terminated.

The `UNIT` keyword sets `fssm_Unit` and therefore the unit number of the exec device on top of which the file system or handler should operate. The unit number enumerates several hardware units handled by the same device. For example, SCSI device drivers map it to the SCSI ID of the drive they address.

The `FLAGS` keyword sets `fssm_Flags` element, and thus the flags for opening an exec device. Its purpose and meaning is specific to the device identified by `fssm_Device`. It is typically 0.

The `fssm_Environ` element is a BPTR to another structure explained in section **??**.

8.1.3 Keywords controlling the Environment Vector

The `fssm_Environ` element of the `FileSysStartupMsg` is a BPTR to another structure that describes, among other things, the layout (or "geometry") of a file system, that is, the partition; beyond file systems, extended handlers mounted by the `EHANDLER` keyword also make use of it.

This structure is also defined in `dos/filehandler.h` and looks as follows:

```
struct DosEnvec {
    ULONG de_TableSize;
    ULONG de_SizeBlock;
    ULONG de_SecOrg;
    ULONG de_Surfaces;
    ULONG de_SectorPerBlock;
    ULONG de_BlocksPerTrack;
    ULONG de_Reserved;
    ULONG de_PreAlloc;
    ULONG de_Interleave;
    ULONG de_LowCyl;
    ULONG de_HighCyl;
    ULONG de_NumBuffers;
    ULONG de_BufMemType;
    ULONG de_MaxTransfer;
    ULONG de_Mask;
    LONG  de_BootPri;
    ULONG de_DosType;
    ULONG de_Baud;
    ULONG de_Control;
    ULONG de_BootBlocks;
};
```

All elements in this structure, except the first one which is set implicitly, are also represented by keywords in the mountlist.

`de_TableSize` defines how many elements in this structure are actually valid and thus are accessible by a handler or file system. It is *not* a byte count, but an element count, excluding `de_TableSize`. In other

words, the DosEnvec is a typical BCPL vector whose vector size is indicated in its first element — note that all elements of the structure are 32-bits wide. For AmigaDOS version 32 and below, at least all elements up to de_NumBuffers shall be present, and thus de_TableSize is at least 11. AmigaDOS version 33 added de_BufMemType, and AmigaDOS version 34 de_MaxTransfer and de_Mask. Version 36 added the remaining elements.

The keywords SECTORSIZE and BLOCKSIZE in the mountlist set both de_SizeBlock. While the mountlist keywords take a byte count, this byte count is divided by 4 to form a long-word count that is inserted into de_SizeBlock. As the name suggests, it defines the size of a storage unit on a medium. Typical values are 128 for 512 byte sectors, or 1024 for 4096 byte sectors. However, not all file systems support all sectors sizes, most of them restrict them to powers of 2, and some only support the default value of 128, i.e. 512 byte sectors. This is also what the Mount command fills in by default.

de_SecOrg is not used and shall be 0; consequently, the Mount command does not provide a keyword to set it.

The SURFACES keyword sets de_Surfaces. A possible interpretation for this element is the number of read-write heads of a magnetic disk drive. However, as the exec device trackdisk interface for magnetic storage media does not access media on such a low level and instead expects byte counts to address sectors, file systems use this value along with de_BlocksPerTrack and the number of cylinders computed from de_LowCyl and de_HighCyl to determine the capacity of the medium or partition they operate on.

The SECTORSPERBLOCK keyword sets the de_SectorsPerBlock, and thus the number of physical sectors on a disk the file system combines to one logical storage block. Not all file systems support values different from 1 here and most restrict this element to powers of 2, including the FFS. File systems read and write data in units of de_SectorsPerBlock times de_SizeBlock long words. For the exec trackdisk interface, only this product matters; however, the "direct SCSI" transfer the FFS version 45 and above offer addressing the disk in terms of physical sectors, and then de_SizeBlock defines the size of a sector in long words as accessed by SCSI commands, whereas de_SectorsPerBlock is the number of physical sectors the FFS reads per logical block. Internally, the FFS administrates partitions and mediums in terms of so-called *keys* where each key addresses one logical block, see section **??** for the details. Both the sector size as defined by de_SizeBlock and the block size in sectors defined by this keyword are required to compute from the key the byte or sector offset the exec device layer requires.

The SECTORSPERTRACK and BLOCKSPERTRACK keywords set both the de_BlocksPerTrack element of the DosEnvec structure. The first name is actually more appropriate as this element defines the number of physical sectors — and not the number of logical blocks — a track of a disk contains. Thus, the naming is a historic accident. As for the SURFACES keyword, the number of sectors per track is only in so far relevant as it defines along with the first and last cylinder the storage capacity of the medium or partition; for FFS partitions, it also defines location of the root block, see section **??**. This block represents the root directory of a medium or partition. Otherwise, the track size as defined by this values never enter the exec device layer directly.

The RESERVED keyword sets the de_Reserved element of DosEnvec and defines the number of (logical) blocks not used by the file system at the start of the disk or partition. For floppies, these reserved blocks hold a (minimal) boot procedure that initializes the *dos.library*; the boot block is described in more detail in section **??**. As the FFS reserves the block number 0 — or rather the key 0 (see **??**) — to indicate an unused or non-existing linkage, the very first block of a partition cannot be made use of anyhow; all occupied blocks are identified by a positive key. The default is to reserve 2 blocks, which is also the number of boot blocks on a floppy.

The PREALLOC keyword installs the de_PreAlloc element, which is supposed to be the number of logical blocks set aside at the end of the partition[4]. However, current FFS versions completely ignore it and no traces of its use are found in the current source. Historically, the BCPL implementation of the OFS shipped with AmigaDOS 34 and before used this value instead to compute the number of blocks the file system could

[4]This is what is also documented in [**?**], though the purpose of this entry changed over time.

pre-read from the disk and store in its internal buffers without imposing a penalty for repositioning the drive head. Thus, this value was 11 for the (ROM-created) mountlist of the floppy as a track there consists of 11 (physical) sectors and these 11 sectors are in the internal buffer of the *trackdisk.device* anyhow. At present, one can only recommend to leave this element alone as its interpretation seem to have changed over time; it could be assigned a different meaning in the future, or could be used by other (non-native) file systems for unknown purposes.

The `INTERLEAVE` keyword defines the lower 16 bits of the `de_Interleave` element. Tripos and AmigaDOS versions prior version 45 reserved the entire long word to define the interleave factor of the disk, which is the difference between two sector numbers the file system is supposed to allocate for storing contiguous data. This interleave factor historically helped to speed up data rates of slow harddisk interfaces; as data is transferred, the following sectors on the rotating disk passed under read head and the harddisk controller would have to wait for almost an entire disk rotation for the numerically next sector to become accessible again. Setting an interleave factor larger than 1 would have helped in such cases. However, neither the original BCPL implementation of the OFS nor the latest FFS make use of this mechanism and allocate sectors contiguously, i.e. with an interleave of 1, regardless of what is set here. The upper 16 bits of `de_Interleave` suit now a different purpose and are controlled by a separate set of keywords in the mountlist Thus, for all practical purposes, this keyword serves no purpose, and even the legacy BCPL implementation ignored it.

The `LOWCYL` keyword initializes the `de_LowCyl` element of the `DosEnvec` structure. It sets the lower end of the partition on the storage medium, i.e. `de_LowCyl` × `de_Surfaces` × `de_BlocksPerTrack` + `de_Reserved` × `de_SectorsPerBlock` is the first physical sector number on a disk that can carry payload data of the file system. Again, while this element seems to imply that a disk consists of equally sized cylinders, each containing `de_Surfaces` tracks, this does not need to be the case and the underlying exec device addresses media through a byte or sector offset instead.

The `HIGHCYL` keyword sets the `de_HighCyl` element; it defines the (inclusive) upper end of the partition in units of cylinders. That is, (`de_HighCyl` + 1) × `de_Surfaces` × `de_BlocksPerTrack` −1 is the last sector of the partition or medium the file system can allocate for storing data.

The `BUFFERS` keyword sets the `de_NumBuffers` element; it defines the (initial) number of file system buffers allocated for caching metadata and storing data not directly accessible through the exec device driver. The number of buffers may be changed later with `AddBuffers()`, see section **??**. Each buffer holds data of one (logical) block, though a small administrative overhead is required on top. Initially, 5 buffers are reserved for the floppies, and while a larger number of buffers generally help, especially when seeking in large files, the overhead of administrating the cache has a negative impact from a certain amount of buffers onward. For the FFS, larger blocks (rather than a larger number of buffers) are recommended as more payload and administration data can be stored per block. That is, the block size even enters inversely quadratic into the FFS overhead.

The `BUFMEMTYPE` defines the `de_BufMemType` element. It defines the memory type, i.e. the second argument of `AllocMem()`, for allocating memory keeping the file system buffers. It cannot, of course, control the memory type into which users require the file system to read data or from which data is to be written. While exec device drivers should be able read from and write data to any memory type, several legacy drivers are not able reach all memory areas; this is often due to limitations of the hardware, for example Zorro-II hardware cannot reach 32-bit memory on CPU expansion boards by means of DMA. Depending on limitations of the `fssm_Device`, it is unfortunately up to the user to provide a suitable `BUFMEMTYPE`. While it seems plausible to provide here a memory type of the RAM that sits "closer" to the physical interface in order to improve transfer speed, this is not necessarily the case as the CPU still has to access this data anyhow, in worst case over the slow Zorro-II bus. Thus, only insignificant performance improvements can be expected from playing with this value. Historically, the *trackdisk.device* could only access buffers in chip memory and required a value of 3 (= `MEMF_CHIP|MEMF_PUBLIC`) here, though this defect was fixed in AmigaOs 36. This is also the default installed by `Mount`, unless a better value is provided by the user. Fully operational device drivers will accept a value of 1 (= `MEMF_PUBLIC`) here.

Make Mask and Buffer Match It makes little sense to provide a very restrictive BUFMEMTYPE without also adjusting the MASK and vice versa. The MASK (see next paragraph) defines under which conditions a file system performs single-block I/O, and the BUFMEMTYPE the memory type from which such buffers are taken. If the latter does not fit the former, the file system would attempt to perform single block I/O into memory it cannot reach, or restrict I/O into single-block I/O for memory it would be able to reach — thus, the two should match. A mask of 0x001fffff fits to a MEMF_CHIP buffer, i.e. a value of 3 for the BUFMEMTYPE, and a mask of 0x00ffffff to Zorro-II memory, namely a value of 513 of the BUFMEMTYPE. FFS 47 onward aligns buffers it allocates to multiples of 16 bytes, suitable for a mask whose last digit is 0. Requiring even more least significant bits to 0 thus makes no sense. FFS versions 45 and below could only ensure that the least significant two bits of its own buffer memory were 0.

The MASK keyword defines the de_Mask element of the DosEnvec structure. It is a workaround for defect device drivers that cannot read from or write to all memory types. If the address of the host memory buffer has bits set in positions where the mask contains 0 bits, the file system assumes that the device cannot reach memory of such an address. Instead, it performs input or output indirectly through its buffers, allocated from BUFMEMTYPE memory, and copies the data manually (i.e. by the CPU) to or from the target memory address. Historically, the *trackdisk.device* required here a mask value restricting access to chip memory, though this requirement has been removed in AmigaOs 36. The Mount command enforces that bit 0 is always 0, i.e. that direct transfers are only possible to word-aligned buffers; the default value is 0xfffffffe, which unfortunately does not fit to the default buffer memory type. Thus, users should probably provide more useful values and override such defaults.

Masking Defects The purpose of the mask is to hide defects in device drivers and provide a working system in the absence of a fully functional device driver. If a mask is set, BUFMEMTYPE shall be set as well as it determines which (alternative) memory will be allocated for buffers in case the intended target memory is not suitable. A rather typical value for the mask is 0x00ffffff, indicating that the device cannot reach 32-bit memory. A suitable memory type would then be MEMF_24BITDMA|MEMF_PUBLIC, i.e. 513 as decimal value. This requests 24-bit memory for the buffers.

The MAXTRANSFER keyword sets the de_MaxTransfer element of the DosList. This value sets the maximum number of bytes the file system shall read or write in a single transfer. Similar to MASK, it is a workaround required to support defect device driver implementations that corrupt data if too many bytes are read or written in one go. The Mount command installs here 0x7fffffff as default value, the *scsi.device* of AmigaDOS 45 and below required here due to multiple defects a value of 0x0001fe00.

MaxTransfer is not a rate The MaxTransfer keyword or element defines a byte count that, when exceeded, requests the file system to break up transfers into smaller blocks. While that implicitly limits the throughput of the device, it does not define a rate (i.e. in units of bytes per second). It is the *amount of bytes to read or write* and not the number of bytes transferred per second that imposes a problem.

The BOOTPRI keyword sets the de_BootPri element which defines the order in which the system attempts to boot from a partition or medium. Clearly, it makes little sense to set this keyword in a mountlist that is interpreted after the system had already booted. However, auto-booting devices may install an appropriate value in the DosList they create through the *expansion.library* from the DosEnvec structure installed in the RDB. The file system itself does not make use of this value anyhow, but the auto-booting device. The default value Mount leaves here is 0.

The DOSTYPE keyword sets the de_DosType element, identifying the type and flavor of the file system to use for the medium or partition. If the FILESYSTEM keyword is present, but the FORCELOAD is either

not present or set to 0, then the `Mount` command first attempts to find a suitable file system in the *FileSystem.resource* whose `fse_DosType` matches `de_DosType`, avoiding to load the same file system again. If a match is found, the `FileSysEntry` of the resource is used to initialize the `DosEnvec`, in particular `dol_SegList` is filled from `fse_SegList`. The `Format` command uses the value deposited here to initialize a partition for a mounted file system, whereas the FFS determines its type from the boot block, i.e. the first block of a partition, see section **??**. Table **??** lists some AmigaDOS file systems:

Table 8.3: File System Types

FFS Flavor	Description
`ID_DOS_DISK`	Original file system (OFS)
`ID_FFS_DISK`	First version of FFS
`ID_INTER_DOS_DISK`	International variant of OFS
`ID_INTER_FFS_DISK`	International variant of FFS
`ID_FASTDIR_DOS_DISK`	OFS variant with directory cache
`ID_FASTDIR_FFS_DISK`	FFS variant with directory cache
`ID_LONG_DOS_DISK`	OFS variant with 106 character file name size
`ID_LONG_FFS_DISK`	FFS variant with 106 character file name size
`ID_COMPLONG_FFS_DISK`	FFS with 54 character file names
`'MSD\0'`	FAT on a disk without partition table
`'MDD\0'`	identical to the above, FAT on a floppy
`'MSH\0'`	FAT on a partition
`'FAT\0'`	FAT, used on a partition or a SuperFloppy
`'CD0\0'`	Original CD File system
`'CD0\1'`	CD File System with Joliet support
`'UNI\0'`	AT&T System-V file system
`'UNI\1'`	Dummy boot "file system" for UNIX boot
`'UNI\2'`	Berkeley file system for System-V
`'resv'`	reserved partition, e.g. swap

The first part of the table indicates various variants of the ROM file system; the original version from Tripos is here denoted as OFS, though this name mostly distinguishes it from its later re-implementation, the FFS.

The OFS variants embed additional administration information (see section **??**) into the data blocks and thus carry less payload data per block. For that, they are are more robust, but slower as the data cannot be transmitted by DMA into the host memory but require an additional copy. The FFS variant immediately below addressed this issue; due to its re-implementation in assembler, it also provided significantly faster disk access. Both first types use, however, a non-suitable algorithm for case-insensitive comparison of file names and thus do not interpret characters in the extended ISO-Latin-1 set (i.e. printable characters outside the ASCII range) correctly (see also section **??** for details on the hashing algorithm). Thus, the first two types should not be used anymore.

Proper case-insensitive interpretation of file names was added afterwards, leading to the next two flavors which are, despite the layout of data blocks, otherwise identical. All types from that point on in table **??** until its end use a proper (ISO-Latin aware) algorithm to compare file names.

The next two versions administrate an additional directory cache; while this cache typically speeds up listing directories, it also requires additional update steps when adding or renaming files, making such operations slower and more error prone. These variants unfortunately also lack a good algorithm to clean up the cache if objects are continuously added and removed from directories. These variants are not generally recommended and should be considered experimental. Sections **??** and following provide details on the directory cache.

The LONG variants of OFS and FFS allow file names of up to 106 characters by using a slightly modified block syntax which overcomes the 30 character file name limit all above variants suffer from. This variant was introduced in version 45 of AmigaDOS. They also use the ISO-Latin aware file name comparison. In some rare cases, the administration information is augmented by one additional block keeping a long comment. This block is specified in section **??**.

The last variant offers longer file names in a way that is backwards compatible to earlier versions of the FFS, i.e. no reformatting is necessary as all block types, except for those holding longer names, remain unchanged. The file name length limit is here 54 characters, though older versions of the FFS can still read the disk correctly, even though they will not be able to locate or list longer file names. This variant is also experimental. While it was already introduced in AmigaDOS 43, it was never officially announced.

The next group of types indicates various flavors of the MS-DOS FAT file system. The first two types are identical and correspond to a file system on a single disk without a master boot record (MBR) type partition table, as found on floppy disks. They are unsuitable for harddisk and thumb-drive partitions as both typically include a MBR.

The MSH\0 type indicates a FAT system on a Master Boot Record (MBR) partition as used on (legacy) PC hardware. As AmigaDOS does not natively support the MBR, the file system here (as an architectural tweak) interprets the partition table. Which partition is used depends on the last character of the device name, i.e. dol_Name. The C character indicates the first partition, adopting the convention of the operating system to which FAT is native, the second partition is indicated by the last character of dol_Name being D and so on.

The FAT\0 type indicates a file system either on a floppy or the first partition of a MBR-formatted disk, depending on the SUPERFLOPPY keyword, see below.

The next group of file system types indicates various versions of the CD-Rom — actually ISO Rock Ridge — file system. The first type is the original file system that came with Version 40 of AmigaDOS, the second the extended version that includes support for Joliet extensions and audio track support. Otherwise, the types are identical.

The last group indicates various file systems for Amiga Unix installations that are not found in mountlists, but in the RDB of the booting harddisk. These file systems do not run under AmigaDOS but rather under Unix variants, and thus do not appear in AmigaDOS installations. A UNI\1 partition appears as an entry in the boot menu, booting an Amiga Unix installation.

The BAUD keyword fills the de_Baud element; it is not used by file systems but extended handlers that are mounted by the EHANDLER keyword. For them, it provides the baud rate for an (assumed) serial connection. This keyword first appeared for serial handlers delivered with the CBM A2232 multi-serial port, but is also interpreted by the AmigaDOS Port-Handler from version 43 onward and by the AUX-Handler version 47 onward.

The CONTROL keyword sets the de_Control element of the DosList structure. Even though this element is here indicated as an ULONG, it can be either an integer or a string, optionally enclosed in double quotes, similar to dol_Startup. It is encoded either as an integer or as a NUL terminated BSTR. It is up to the user to learn from the handler documentation what the handler expects here as the Mount command has no means of checking the correctness of the value. The Port-Handler, see section **??**, and AUX-Handler (see section **??**) interpret this entry as the definition of the serial connection parameters, both requiring quoted strings.

The BOOTBLOCKS keyword initializes the de_BootBlocks element; it is not interpreted by the file system itself, but by the boot code of the auto-booting device. As such, this makes only sense if the DosEnvec structure is loaded from the RDB of an auto-booting disk. If non-zero, the device reads the indicated number of boot blocks from the booting partition and runs the code within the boot-block, thus implements boot-block booting similar to booting from the *trackdisk.device*, and by that replaces boot-point booting performed otherwise on auto-booting devices. For details, see [**?**]. The implied value for the floppies is thus 2, and 0 for partitions that follow the boot-point protocol. This value has, in particular, no impact

on the disk layout, though the number of bootblocks shall be smaller or equal than the number of reserved blocks indicated by de_Reserved as otherwise the file system could overwrite the boot code.

The SUPERFLOPPY keyword takes a Boolean 0 or 1 argument and by that either sets or clears the ENVF_SUPERFLOPPY flag, defined in dos/filehandler.h. This flag has been cut off (or reserved) from the otherwise deprecated and thus unused de_Interleave element in version 45 of AmigaDOS. It is by default cleared.

If this flag in de_Interleave is set, then the file system is informed that the partition extends over the entire medium and no partition table or RDB is present. Instead, to find the size of the medium, the file system is authorized to issue a TD_GETGEOMETRY command to the exec level device driver which will report the layout of the disk. This driver information is then used to adjust de_LowCyl, de_HighCyl, de_SizeBlock and de_Surfaces within the DosEnvec structure. Thus, a file system mounted with this flag enabled adjusts the disk geometry directly from the hardware driver. Details on this procedure are found in section **??**. This is important for drives that allow variably sized media, such as floppy disks (supporting both DD and HD disks) as well as ZIP drives (supporting 100MB to 250MB drives).

AmigaDOS up to release 40 hardwired this special case to ROM-based devices that supported variably sized media, namely to the trackdisk.device and carddisk.device, i.e. floppies and memory cards in the PCMCIA slot. Newer releases allow to enable this mechanism for other device drivers as well. As of version 45, the FAT and the FFS both support this mechanism.

The FAT system is a special case and only honors SUPERFLOPPY if the dos type is set to 'FAT\0'. If SUPERFLOPPY is set to 1, a FAT file system mounted with the dos type 'FAT\0' receives the partition size from the physical layer and ignores the MBR, whereas if this this keyword is set to 0, the MBR is used as source of the geometry information. All other dos types of the FAT system ignore this keyword.

The SCSIDIRECT keyword is also a Boolean indicator and controls the ENVF_SCSIDIRECT flag which is also part of the otherwise deprecated de_Interleave element. Similar to the above flags, it is also defined in dos/filehandler.h. This flag is cleared by default, indicating that the file system should use the trackdisk command set to access data. This keyword was also introduced in AmigaDOS version 45.

If this flag is set, the file system is instructed to communicate with the underlying device through the HD_SCSICMD interface, i.e. by SCSI commands instead of trackdisk commands. This helps some legacy device drivers that do not speak the 64-bit dialect of the trackdisk commands to access data beyond the 4GB barrier. Very ancient device drivers may not even support this command set, and it is therefore not enabled by default.

The ENABLENSD keyword is again a Boolean indicator for the ENVF_DISABLENSD flag cut out off from the de_Interleave element; it is, however, set in inverse logic, i.e. the ENVF_DISABLENSD is set if the mount parameter is 0, and reverse. This flag is also defined in dos/filehandler.h.

If this flag is set in de_Interleave, and thus ENABLENSD is set to 0 in the mountlist, the file system is instructed not to attempt to use NSD-style commands to access data beyond the 4GB barrier. This could be necessary on some device drivers that ignore the most significant bits of the io_Command element or react otherwise allergic to commands beyond their supported command range. This flag was also introduced in AmigaDOS version 45.

Unfortunately, multiple command sets exist to access (moderately) large disks. If DIRECTSCSI is enabled, SCSI commands will always be used, even for probing the medium size if SUPERFLOPPY is set. If SCSI commands are not enabled, the FFS first attempts to use regular trackdisk commands if the partition does not cross the 4GB barrier. If it does, it probes the TD64 command set as it is historically the most popular extension. As last resort, it tries NSD commands. This last step can be disabled by setting the ENABLENSD in the mountlist to 0 should NSD create problems.

The ACTIVATE keyword in the mountlist is synonym to the MOUNT keyword, and it also takes a Boolean indicator. If it is set, then the Mount command will already load and initiate the handler corresponding to the mount entry. Otherwise, the handler or file system will be loaded the first time it is accessed through a

path. This option is most useful for testing device drivers, or load them from a disk that is not necessarily accessible later on. It does not affect the DosEnvec structure.

The FORCELOAD keyword of the mountlist is also a Boolean flag, that, if set to 1, indicates to the Mount command not to scan the *FileSystem.resource* for a fitting dos type but rather forcibly load the file system from the path indicated by the FILESYSTEM keyword. This option is also useful for testing, namely to prevent that an already resident file system version is reused. This keyword does neither impact the DosEnvec structure.

8.2 Finding Handler or File System Ports

The following functions find the MsgPort of a handler or file system that is responsible for a given path. The path can be either a relative path, in which case they deliver the port of the file system handling the current directory, or an absolute path. In the latter case, they 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 path and thus reiterate the call if a requested file or directory object cannot be found immediately.

8.2.1 Iterate through Devices Matching a Path

The GetDeviceProc() finds a handler, or a subsequent 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) /* since V36 */
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 a DevProc structure identifying a suitable handler or file system. It returns NULL if no matching handler could be found or all possible directories of a multi-assign have been visited. It is not necessary that the path given by name is an absolute path containing a colon (":"), this function will also operate properly for relative paths and then identifies the file system and lock responsible for the current directory.

> *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 second argument for a subsequent GetDeviceProc() call. In case of failure, this function will return NULL and then also releases all resources, including the DevProc structure provided as second argument.

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;
};
```

dvp_Port is a pointer to a candidate MsgPort of a handler or file system. The substring of name behind the colon (":"), or the entire string if name does not contain a colon, forms a path relative to dvp_Lock on the file system listening on the found port. If this path does not resolve to an existing file system object, iteration over possible candidate file systems should continue and GetDeviceProc() called again with the same name as first, and the return code of the current iteration as second argument. This iteration is necessary to find files or directories in multi-assigns.

dvp_Lock is a directory represented as a lock relative to which name could be resolved. If this lock is ZERO, the path is relative to the root directory of the file system that can be contacted through dvp_Port. Beware! In this case, a ZERO lock does not identify the root directory of the boot volume, but the root directory of the file system to be contacted through dvp_Port.

The lock, if non-ZERO, is owned by the *dos.library* and shall neither be released, nor used for a directory scan by Examine() or ExAll(). If this is desired, it shall be copied first through DupLock(), see also section **??**.

dvp_Flags identifies properties of the found port and lock, though there is typically no need to test them. The flags are defined in the include file dos/dosextens.h. The currently defined flags are listed in table **??** for completeness:

Table 8.4: GetDeviceProc Flags

Flags	**Description**
DVPF_ASSIGN	Path is part of a multi-assign
DVPF_UNLOCK	Internal, do not interpret

If the bit DVPB_ASSIGN is set, i.e dvp_Flags & DVPF_ASSIGN is non-zero, then the found port and lock are part of a multi-assign. The port/lock pair returned does not necessarily contain the (or an) object identified by name; if it does not, another iteration through GetDeviceProc() is necessary. From this one can conclude that an object in a directory scanned earlier will hide an object of the same name in a directory of a multi-assign scanned later.

The bit DVPB_UNLOCK in the flags is an internal bit and indicates whether the lock in dvp_Lock will be released by the next iteration or FreeDeviceProc(). This flag shall not be interpreted or altered by the caller.

The element dvp_DevNode shall not be touched or used. It is actually a pointer to the DosList the function identified as being responsible for the path — if there is one.

A typical use case for this function is to identify a handler or file system for direct-packet I/O, see also chapter **??** and in particular section **??**. The packets listed in chapter **??** typically take a lock/path combination to identify a file system object, and for them, the lock will be taken from dvp_Lock and the path from name. The target MsgPort of the packet is taken from dvp_Port. The interface functions of the *dos.library*, such as Open() and Lock(), call through GetDeviceProc() and therefore perform all these steps within the library. The following code example demonstrates this on an (albeit limited, because it does not handle soft links) re-implementation of the Lock() function of the *dos.library*:

```
BPTR myLock(const UBYTE *path)
{
  struct DevProc *dp = 0;
  BPTR lock          = 0;
  LONG error         = 0;
  size_t len         = strlen(path);
  struct Buffer {
    UBYTE buf[256];
  };
```

```
  D_S(struct Buffer,buf);
  /*
  ** Convert to BSTR on the stack.
  ** Size check not performed by dos.library.
  */
  if (len > 255) {
   error = ERROR_INVALID_COMPONENT_NAME;
  } else {
    buf->buf[0] = len;
    memcpy(&buf->buf[1],path,len);
    while(dp = GetDeviceProc(path,dp)) {
      lock = DoPkt(dp->dvp_Port,ACTION_LOCATE_OBJECT,dp->dvp_Lock,
                   MKBADDR(buf->buf),SHARED_LOCK,0,0);
      if (lock)
        break;
      error = IoErr();
      if (error != ERROR_OBJECT_NOT_FOUND)
        break;
    }
  }

  /* Free devproc. NULL is ok */
  FreeDeviceProc(dp);

  SetIoErr(error);

  return lock;
}
```

It uses the `D_S` macro from section **??** to create a long word aligned buffer on the stack, here for the converted name of the object to be locked. Unlike what official documentation has to say, it is not necessary to check the `DVPF_ASSIGN` flag. `GetDeviceProc()` may be called again even with this flag cleared, though it then always returns `NULL`.

The `GetDeviceProc()` function has a couple of side effects: First, if the corresponding handler is not yet loaded, i.e. `dol_SegList` in the `DosList` is `ZERO`, it will be loaded from the path in `dol_Handler` in the context of the calling process. The `Mount` command uses this side effect to implement the `ACTIVATE` keyword in mountlists.

If `dol_Task` is `NULL`, then an instance of the file system or handler is also started, with the `name` converted to a BSTR in the startup packet, see section **??**. This mostly affects handlers such as the CON-Handler which leave `dol_Task` at `NULL` and use the path in the startup packet to configure itself. The CON-Handler versions 37 and before already opened the window from this path without ever (or before) receiving a packet requesting a file (see sections **??** and following), causing the side effect of creating a stray window or even an error claiming an invalid window specification. This was fixed in version 39. Still, without sending any further packet to the handler, its process remains active.

Thus, as a fairly general recommendation, `GetDeviceProc()` should only be called to find the target `MsgPort` for a subsequent packet transmission, see also chapter **??**. Handlers such as the CON-Handler will process this packet, and if after processing this packet, no file handle is open, will terminate, avoiding a useless idle process. Section **??** provides in source code form an example.

If `GetDeviceProc()` 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 can 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 with an existing DevProc structure as second argument with the DVPB_ASSIGN bit cleared. IoErr() remains in this case 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 error code is not fully correct, and callers should be aware of this and filter the NIL: path out beforehand.

If the path starts with CONSOLE:, then this function reports the port of the current console, see **??**, and a ZERO lock, but sets IoErr() (incorrectly) to an error code even if process is equipped with a console. IOErr() is also set incorrectly for paths relative to PROGDIR:, the home directory of the calling process. Thus, IoErr() shall not be interpreted in case of success. If the calling process does not have a home directory, probably because it is executing a resident command (see **??**), then GetDeviceProc() erroneously shows a requester asking the user to insert a volume of the name PROGDIR instead of indicating failure — unless requesters are disabled of course. This defect persists in even the latest version of AmigaDOS.

Also, GetDeviceProc() does not handle the path "*" at all, even though it corresponds to the current console and the CON-Handler is responsible for it. This case also needs to be filtered out by the caller upfront, and the port of the console should be obtained through GetConsoleTask() specified in section **??**.

> *Does not like all paths* The GetDeviceProc() function unfortunately does not handle all paths 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.

Unfortunately, the GetDeviceProc() handling of assigns includes a race condition, namely that assigns can be added or removed under its feed, i.e. while one process received dvp_Lock from an assign, a second process is able to remove the directory being visited by RemAssignList(), see section **??**. A use-count identifying how often a lock has been passed out to a client is still missing.

Another defect of this function is that it cannot handle device, volume or assign names longer than 30 characters, even though no limit on the path name in total is imposed, at least not by this function. This is because this function extracts the device name into a temporary buffer of this size, and truncates it if it is longer.

GetDeviceProc() and the algorithm by which it loads and starts handlers are also discussed in section **??**, see there for further details.

8.2.2 Releasing DevProc Information

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

```
FreeDeviceProc(devproc) /* since V36 */
                  D1

void FreeDeviceProc(struct DevProc *)
```

This function releases the DevProc structure and all its resources from an iteration through one or multiple GetDeviceProc() calls. It shall be called only to abort an iteration over devices, not within each iteration, see section **??** for an example. In particular, if calling GetDeviceProc() had returned NULL it released such resources itself already and FreeDeviceProc() shall not be called.

The dvp_Port or dvp_Lock within the DevProc structure shall not be used after releasing it with FreeDeviceProc(). If the lock is required after releasing a DevProc structure, 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, see sections **??** and **??**.

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.

8.2.3 Legacy Handler Port Access

The DeviceProc() function is a legacy variant of GetDeviceProc() that should not be used anymore. It is not able to resolve non-binding 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 the handler (or *a* handler) responsible for the path name. It returns NULL on error in which case it sets IoErr().

If the passed in name is a relative path or indicates an assign, the handler port responsible for the current directory or the directory the assign binds to is returned, and IoErr() is set to the lock of this directory. Similar to GetDeviceProc(), this lock is owned by the *dos.library* and shall be duplicated before it is used to iterate over a directory with Examine() or ExAll().

> *Obsolete and not fully functional* The 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. Similar to GetDeviceProc(), it does not handle the paths NIL: and "*" properly.

As this function is based on GetDeviceProc(), it suffers from the same race conditions and limitations. It cannot handle device, volume or assign names longer than 30 characters and does not handle situations gracefully within which an assign is removed after a lock on its target directory has been passed out.

8.2.4 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() /* since V36 */
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 calling process. The latter holds for example for programs started from the Workbench. It does not alter IoErr().

8.2.5 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.

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, see **??**.

```
port = GetFileSysTask() /* since V36 */
 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, they can be different.

8.3 Iterating and Accessing the Device List

While GetDeviceProc() uses the device list to locate a particular MsgPort and lock, all other elements 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.

8.3.1 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 flags that define access to which parts of the list are requested:

```
dlist = LockDosList(flags) /* since V36 */
 D0                  D1

struct DosList *LockDosList(ULONG)
```

This function grants read (shared) or write (exclusive) 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. Unlike what the function prototype implies, the returned value *is not* a pointer to a DosList structure. For locating a particular entry on the device list, see FindDosEntry() in section **??**.

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

Table 8.5: LockDosList Flags

Flags	Description
LDF_DEVICES	Access handlers and file system entries, see **??**
LDF_VOLUMES	Access volumes, see **??**
LDF_ASSIGNS	Access assigns, see **??**
LDF_ALL	Combination of all of the above
LDF_ENTRY	Access to the device list during handler startup
LDF_DELETE	Lock device list for deletion of entries
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. For example, GetDeviceProc() performs a read access on the list while looking for a particular device, while adding assigns through AssignLock() (see section **??**) requires write access with LDF_ASSIGNS. The three first flags may also be combined to access multiple types, e.g. as LDF_ALL to request all entries.

LDF_ENTRY and LDF_DELETE are additional flags that moderate access to entries of the device list. LDF_ENTRY locks the list while loading and starting new handlers and file systems. If dol_GlobVec requires exclusive access, see table **??** in chapter **??**, then an exclusive lock of this type is requested during handler startup. If LDF_DELETE is set, then access is granted for removing entries. If shared access is requested for handler startup, then only a shared lock of this type is requested to prevent that the handler being started is removed during its startup process. Both flags are only used internally by the *dos.library* and *shall not be set* by applications, handlers or file systems.

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.

Application programs *should not* keep the device list locked while sending a packet to a handler or file system. This is because handlers or file systems need access to the device list themselves under certain conditions, for example to add or remove a DosList entry representing a volume inserted into or ejected from the drive maintained by the file system. Unfortunately, the *dos.library* within GetDeviceProc() itself also locks the list when attempting to load a handler or file system, potentially from the same handler that requires to modify the list at the same moment. If the handler requests access using LockDosList(), this can result in a deadlock, as for example the *dos.library* waits for the handler to open the file, and the handler waits for the lock to become available to add or remove a volume. To avoid this problem, handler and file system implementations should rather call AttemptLockDosList() specified in section **??** and defer modifying the list until after the semaphore becomes available; in the meantime, they should continue to serve incoming requests *without waiting for them*, and defer the modification of the device list until after the list could be successfully locked.

> *Not for Handlers and File Systems* Handlers and File Systems should not request blocking access to the device list as this can deadlock the system. Instead, AttemptLockDosList() should be used to request access to the list, while continuing to serve incoming packets until the lock can be granted.

For backwards compatibility to AmigaDOS versions 34 and earlier, this function also calls Forbid(), implementing the access protocol of such earlier versions.

This function does not alter IoErr().

8.3.2 Attempting 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, but fails if the list is not accessible.

```
dlist = AttemptLockDosList(flags) /* since V36 */
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 **??**, and the semantics of the flags are exactly as specified for LockDosList(), see table **??** there for details.

This function should be used within handlers to check whether access to a (subset) of the device list is possible as blocking access through `LockDosList()` may lead to a deadlock, see also section **??**.

On success, the result code is a non-`NULL` handle that may be passed into the `FindDosEntry()` function for finding a handler, file system, volume or assign matching a name, or into the `NextDosEntry()` function for iterating the device list manually. On error, the result code is `NULL`, either because the list is currently locked and access cannot be granted without blocking, or flags are invalid. These two cases of failure cannot be distinguished unfortunately.

Up to AmigaDOS versions 39, this function could return either `NULL` or `(struct DosList *)1` on failure[5], and thus — if compatibility to such versions is intended — both return codes would need to be checked. This was fixed in version 40.

This function does not alter `IoErr()`.

8.3.3 Release Access to the Device List

The `UnLockDosList()` function releases access to the device list obtained by `LockDosList()` or `AttemptLockDosList()`.

```
UnLockDosList(flags) /* since V36 */
               D1

void UnLockDosList(ULONG)
```

This function releases access to the device list. The `flags` argument shall be identical to the `flags` argument provided to `LockDosList()` or `AttemptLockDosList()`.

As a compatibility kludge to AmigaDOS versions 34 and before, this function also calls `Permit()`, implementing the legacy access protocol to the device list.

8.3.4 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()` or `AttemptLockDosList()`.

```
newdlist = NextDosEntry(dlist,flags) /* since V36 */
 D0                      D1    D2

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

This function returns the `DosList` structure following the `dlist` handle on the device list. The device list shall be locked either by `LockDosList()` or `AttemptLockDosList()`. The `dlist` argument shall be either the return value of a previous `NextDosEntry()` or `FindDosEntry()` call, or the handle returned by `LockDosEntry()` or `AttemptLockDosList()`.

`flags` shall be a subset of the flags requested from `LockDosList()` or `AttemptLockDosList()` and specifies the type of `DosList` structures that shall be iterated over. Only the first 3 rows of table **??** are relevant here, all other flags are ignored but may be included.

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

The following example code demonstrates how to iterate over all device list entries:

[5] That is, the numerical value 1 casted to a `DosList` pointer.

```
void IterateDosList(void)
{
  struct DosList *dl;

  dl = LockDosList(LDF_ALL|LDF_READ);
  while((dl = NextDosEntry(dl,LDF_ALL))) {
    const UBYTE *name = ((UBYTE *)BADDR(dl->dol_Name)) + 1;
    ...
  }
  UnLockDosList(LDF_ALL|LDF_READ);
}
```

Note that the `name` is stored as a BSTR that is, conveniently, always `NUL` terminated. The `name` *cannot* be safely printed in the loop as printing implies the transfer of a packet to the console while the device list is locked, see also the notes in section **??**. If the names are required, they shall be copied within the loop and then printed later.

8.3.5 Find a Device List Entry by Name

The `FindDosEntry()` function finds a `DosList` structure of a particular type and name, searching the list starting at a given entry, or the handle returned by `LockDosList()` or `AttemptLockDosList()`.

```
newdlist = FindDosEntry(dlist,name,flags) /* since V36 */
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()` or `AttemptLockDosList()`, and returns the next `DosList` structure, potentially *including* the one pointed to by `dlist`, that is of the type indicated by `flags` and has the name `name`.

`flags` shall be a subset of the flags requested from `LockDosList()` or `AttemptLockDosList()`. Only the first 3 elements of table **??** 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 to be found. The name *shall not* include the colon (':') that separates the name from the remaining components of a path, see chapter **??**. It may be `NULL` in which case every entry of the requested type matches.

Note that it is possible that more than one entry on the device list matches a given name. Duplicate entries of the same name on the device list *are* possible as AmigaDOS does support identically named volumes; they are distinguished by their creation date. This is not possible for handlers or assigns, their names are unique.

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, `NextDosEntry()` may be called upfront to continue the scan from the previous entry found.

Passing `NULL` as `dlist` is safe and returns `NULL`, i.e. the end of the list. Note that the (pseudo-) devices from table **??** in section **??** and assigns from table **??** in section **??** are not part of the device list, i.e. `NIL`, `CONSOLE`, `*` and `PROGDIR` cannot be found and are special cases handled within `GetDeviceProc()` and other functions of the *dos.library*.

`FindDosEntry()` does not alter `IoErr()`.

8.3.6 Accessing Mount Parameters

Once a `DosList` structure has been identified, e.g. by `FindDosEntry()`, it is tempting to understand whether the entry belongs to an assign, a volume, a handler or file system, and in the latter two cases, to find their mount parameters to regenerate a mountlist entry from the mounted partition or volume.

The type of the entry is easily found in the `dol_Type` element of the `DosList` structure; table **??** in section **??** lists the possible entry types. The type `DLT_DEVICE` indicates both file systems and handlers.

The mount information, or to be more precise, the configuration of the handler or file system is found in the `dol_Startup` element; it is, however, up to the handler to interpret it, and AmigaDOS does not define its syntax. The `Mount` command can place four different types of objects here: An integer value, a BPTR to a BSTR, or a BPTR to a `FileSysStartupMsg`, though even other types could be deposited here by handler-specific tools.

Unfortunately, there is no completely safe way how to distinguish these types, and AmigaDOS does not provide any further source of information to learn what a handler expects here — thus a heuristics is needed to tell them apart and access the mount information.

The following algorithm attempts to detect the nature of the information provided in `dol_Startup` given a pointer to a `DosList`, that has been, for example, returned by `FindDosEntry()`:

```
void AnalyzeDosList(struct DosList *dl)
{
 LONG *s;
 UBYTE *text;

 s = (LONG *)(dl->dol_misc.dol_handler.dol_Startup);
 if (((LONG)s & 0xc0000000)==0 && TypeOfMem(BADDR(s))) {
   LONG *startupmsg = BADDR(s);
   /* Looks like a plausible BPTR */
   /* This checks whether fssm_Device
   ** is meaningful in order to derive
   */
   if ((startupmsg[1] & 0xc0000000)==0) {
     /* Hopefully, a BPTR to a BSTR,
     ** namely the device name.
     */
     text = ((char *)BADDR(startupmsg[1]))+1;
     if (TypeOfMem(text)) {
       /* Now check the want-to-be fssm_Envion */
       if ((startupmsg[2] & 0xc0000000) == 0 &&
           TypeOfMem(BADDR(startupmsg[2]))) {
         struct FileSysStartupMsg *fssm;
         struct DosEnvec *env;
         fssm = (struct FileSysStartupMsg *)startupmsg;
         env  = (struct DosEnvec *)BADDR(fssm->fssm_Environ);
         /*
         ** Access fssm->fssm_Device,fssm_Unit,fssm_Flags
         */
         /* Hopefully an environment */
         if (env->de_TableSize >= 11) {
           UBYTE *device = (UBYTE *)(BADDR(fssm->fssm_Device)) + 1;
           ULONG  unit   = fssm->fssm_Unit;
           /*
```

```
            ** By convention, fssm_Device is a NUL-terminated
            ** BSTR, thus no need to convert it to a CSTR.
            **
            ** Thus, likely a file system startup message.
            ** This is good enough, now use the elements
            ** of the environment up to the one indicated
            ** by env->de_TableSize.
            */
            ...
            return;
          }
        }
      }
    }
    /* This is probably a string.
    ** "mount" puts NUL-terminated strings in
    ** here.
    */
    text = ((UBYTE *)startupmsg)+1;
    ...
    return;
  } else {
    /* Likely some sort of handler, and s is
    ** probably some integer
    */
    ...
    return;
  }
}
```

In the first step, the algorithm above attempts to find out whether `dol_Startup` is an integer or a BPTR. It uses `TypeOfMem()` to test whether a pointer goes into valid memory, and also checks the topmost two bits of the BPTR. As BPTRs are created by right-shifting a pointer by 2 bits, the two MSBs should be zero. Even for ROM-mounted devices like the floppies, the `DosList` structure and all structures and strings associated to it are located in RAM and thus the above test also works for them.

In the second step, the heuristics attempts to understand whether the BPTR in `dol_Startup` actually points to a `FileSysStartupMsg` or to a BSTR. For that, it attempts to learn whether the `fssm_Device` element is a valid BSTR and whether the `fssm_Environ` element is also a valid BPTR.

If these conditions are not true, but `dol_Startup` is a valid BPTR, the algorithm assumes that it points to a BSTR. In all other cases, the value is assumed to be an integer.

Even if this function is able to identify an `DosEnvec` structure with high probability, the environment vector found in this way does not need to be complete and does not need to contain all the elements listed in section **??**; instead, only the first `de_TableSize` elements are present, and everything beyond this point shall not be interpreted or modified. AmigaDOS versions 32 and above allocate at least 11 entries here[6], for every additional entry a test of `de_TableSize` shall be performed.

While the above is just a heuristic and is therefore not guaranteed to work, practical experience of the author has shown that it has so far been able to extract environment information from all handlers or file systems that came into his hand.

[6] ...and so does Tripos.

Nevertheless, getting hands on the `FileSysStartupMsg` from an unknown handler is not completely waterproof at this moment, and to this end the author of [**?**] proposed to introduce a `DosPacket` (see table **??** in section **??**) by which a handler could be requested to reveal its startup message. Unfortunately, to date this packet has not found wide adoption, and the above heuristic can be used as an interim solution.

Authors of handlers and file systems have less to worry about. When they document their requirements properly, e.g. by including an example mountlist with their product, "only" user errors can generate startup messages the handler cannot interpret properly. Thus, in general, handlers should be written in an "optimistic" way (unlike the above heuristic) assuming that `dol_Startup` is what they do expect. All AmigaDOS handlers are designed this way, e.g. the FFS expects without verification that the value in `dol_Startup` is, indeed, a `FileSysStartupMsg`, even though it can be fooled by an incorrect mountlist and by that crash the system.

8.4 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. They cannot prevent, however, that an entry is removed whose resources, such as locks of an assign, are currently in use by a another process. Locks recorded in a `DosEntry` therefore cannot be safely released without causing a race condition.

A second potential race condition exists when launching handlers. At this time, `GetDeviceProc()` requires access to the device list and secures it through `LockDosList()`. The list will be unlocked only after the handler process has been started and replied the startup packet. Thus, handlers need not (and shall not) call `LockDosList()` to secure access to the device list while processing the startup packet, see also section **??** for details on handler startup. Attempting to lock the device list with `LockDosList()` 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 get access to the device list.

A similar race condition exists if a file system is used to *load* another handler. At this point, the device list is also already locked by the *dos.library*, and an attempt to lock it within the file system would deadlock as well as the library can only unlock the list after loading completed.

> *Locking the device list deadlocks the system* Handlers shall not attempt to lock the device list through `LockDosList()` as this function may block and cause a deadlock if the *dos.library* or an application program is accessing the list simultaneously while reaching out for the handler. Instead, `AttemptLockDosList()` shall be used, and if getting access to the list fails, the operation requiring the lock, e.g. adding a volume to the device list, shall be delayed until after the list becomes accessible.

8.4.1 Adding an Entry to the Device List

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

```
success = AddDosEntry(dlist) /* since V36 */
D0                     D1

LONG AddDosEntry(struct DosList *)
```

This function takes an initialized `DosList` 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 safely used here.

Assigns *shall not* be added to the device list through this function, but rather through the functions in section **??**. 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 shall 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 attempt is made to add an entry of name to the list that is already present, regardless whether the types are identical or not. The only exception is that the list may contain two volumes of the same name, provided provided their creation date dol_VolumeDate differs, see chapter **??**.

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

8.4.2 Removing an Entry from the Device List

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

```
success = RemDosEntry(dlist) /* since V36 */
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. The device list shall be locked upfront depending on the type of the entry that is to be removed, e.g. if the entry is a volume, an LDF_VOLUMES|LDF_WRITE lock on the device list is required, either through LockDosList() or through AttemptLockDosList(), see also the example below.

The function does *not* attempt to release the memory allocated for the DosList passed in, or any of its resources such as locks, it just removes the DosList from the device list. While file systems may know how they allocated the DosList structures representing 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 FreeDosEntry() in **??**. In particular, as entries representing handlers and file systems may have been created in multiple ways, their memory cannot be safely recycled. Assigns *shall not* be manipulated through this function, but rather through the functions in section **??**.

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 the GetDeviceProc() function, a deadlock can result where file system and *dos.library* mutually block access. To prevent this from happening handlers shall check upfront whether the device list is available for modifications by AttemptLockDosList(), e.g.

```
if (AttemptLockDosList(LDF_VOLUMES|LDF_WRITE)) {
  rc = RemDosEntry(volumenode);
  UnLockDosList(LDF_VOLUMES|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 removing 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 element of the device list. This function does not touch `IoErr()`.

> *Locking protocol differs* While `AddDosEntry()` locks the device list itself, this is *not* the case for `RemDosEntry()` which requires the caller to obtain a `LDF_WRITE` lock combined with the flags corresponding to the type of the entry to be removed, see table **??** for the flags corresponding to volumes, devices or assigns.

8.5 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 initialization of the structure. For assigns, the functions in section **??** 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, regardless of their type, `AssignLock()` from section **??** with a `ZERO` lock is the best solution.

`DosList` structures representing volumes are build and released by file systems; it depends on them which resources need to be released along with the `DosList` structure. 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 to only unlink such nodes from the device list if absolutely necessary, but tolerate the memory leak.

8.5.1 Creating a Device List Entry

The `MakeDosEntry()` creates an empty `DosList` structure of the given type, and performs elementary initialization. It does not acquire any additional resources despite the `DosList` structure and the name, and neither inserts the created structure into the device list.

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

```
newdlist = MakeDosEntry(name, type) /* since V36 */
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 **??** in chapter **??**. The function also initializes the `dol_Name` element to a `NUL`-terminated BSTR, copied from the (regular C string) argument `name`.

Note that this function performs only minimal initialization of the `DosList` structure. All elements except `dol_Type` and `dol_Name` are initialized to 0. This makes additional initialization necessary by the caller.

> *Limit to 30 Characters* While this function does not impose a limit on the size on the `name`, multiple other components of AmigaDOS are not able to handle device, volume or assign names longer than 30 characters. Thus, the length of `name` shall be limited to this size.

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.

8.5.2 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()` specified in section **??**. While `FreeDosEntry()` releases the memory holding the name of the entry and the `DosList` structure itself, it does not release any other resources. They shall be released by the caller of this function upfront. Furthermore, this function shall not be called if the `DosList` structure was allocated by any other means than `MakeDosEntry()`.

```
FreeDosEntry(dlist) /* since V36 */
                D1

void FreeDosEntry(struct DosList *)
```

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

Unfortunately, the *expansion.library* function `MakeDosNode()` uses a memory allocation policy that is different from `MakeDosEntry()` and thus `DosList` structures created by the *expansion.library* cannot be safely released by `FreeDosEntry()`. 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`, `DLT_NONBINDING` or `DLT_LATE`, this function should not be used. Instead, the functions from section **??** are more appropriate; in specific, `AssignLock()` with the name of the assign to remove and the `lock` argument set to `ZERO` will remove the assign from the device list and release all resources associated to it, including the name and all locks.

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

> *Only for Internal Use* The `FreeDosEntry()` function has limited uses. It should not be called on handler or file system entries as (at least) two incompatible functions exist that create such `DosList` structures. For assigns, the functions from section **??** are more appropriate, and for volumes, only the file system managing the volume may call it. That leaves no practical use for this function by application programs. It is, however, called internally from other functions of the *dos.library*.

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

8.6 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, enqueue and remove `DosList` structures representing assigns in a single call and are thus easier to use. They also lock and unlock the device list appropriately, and thus `LockDosList()` does not need to be called.

Unfortunately, all the functions provided in this section have a race condition in handling locks representing the target directory (or directories) of an assign. As locks representing such directories are passed out by `GetDeviceProc()` (see section **??**) or `DeviceProc()` without any internal reference counting, it can happen that they are still in use when they are released by canceling, updating or converting an assign.

Multi-Assigns are created by starting from a regular assign to one target directory with `AssignLock()`, see section **??**, and then iteratively adding directory after directory to the assign with the `AssignAdd()` function specified in section **??**.

8.6.1 Create, Update or Remove an Assign

The `AssignLock()` function scans the device list for an assign of a given name. If this assign is not yet present, it adds a regular assign to the directory as represented by a lock. If the assign is already present, the assign is converted to a regular assign to the target directory. If the target lock is `ZERO`, the assign is canceled.

```
success = AssignLock(name,lock) /* since V36 */
D0                    D1   D2

BOOL AssignLock(STRPTR,BPTR)
```

This function creates, updates or cancels the assign identified by `name` which shall not include a trailing colon (“`:`”). The `lock` argument shall be either `ZERO` or a shared lock to a directory.

If no `DosList` of the given `name` exists, and `lock` is not `ZERO`, the function creates a new regular assign under the given name that points to the target directory given by `lock`.

If an assign `name` already exist and `lock` is not `ZERO`, then the assign, regardless of its type, is converted to a regular assign pointing to the target directory given by `lock`. Any resources associated to multi-assigns or late- or non-binding assigns are released.

If the assign `name` already exists and `lock` is `ZERO`, then any type of assign is canceled, that is, all its resources — including locks — are released and the assign is removed from the device list.

Unfortunately, even in the most present release of AmigaDOS, this function has a race condition if an attempt is made to cancel an assign whose lock is currently in use by an application program, for example because it has been passed out through `GetDeviceProc()`, see section **??**.

If the function is successful, it returns a non-zero result code. The `lock`, if non-`ZERO`, is then absorbed into the assign and should no longer be used by the calling program. On success, `IoErr()` is not consistently set and its value cannot be relied upon.

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, and `ERROR_INVALID_COMPONENT_NAME` if the assign name is longer than 30 characters. If a volume, handler or file system of the same name already exists, the error code is `ERROR_OBJECT_EXISTS`.

8.6.2 Create or Update a Non-Binding Assign

The `AssignPath()` function creates or converts an assign to a non-binding assign and adds it to the device list if it is not yet present. 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 = AssignPath(name,path) /* since V36 */
D0                    D1   D2

BOOL AssignPath(STRPTR,STRPTR)
```

This function scans the device list for an assign given by name. The name shall not contain a trailing colon (":"). If such an assign does not yet exit, it creates a non-binding assign to path. 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 directories of the processes using the assign.

If an assign name already exists, it is canceled, all of its resources, including one or multiple locks are released, and it is converted into a non-binding assign to path.

Unfortunately, even in the most present release of AmigaDOS, this function has a race condition if locks of a previous assign are released and these locks are still in use by an application program.

If the function is successful, it returns a non-zero result code. On success, IoErr() is not set consistently and cannot be relied upon. The string path then has been copied into a buffer allocated by the function, and path remains available to the caller.

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 the assign name is longer than 30 characters, the error code is set to ERROR_INVALID_COMPONENT_NAME. If a DosList entry representing a handler, file system or volume of the same name already exists, the error code is ERROR_OBJECT_EXISTS.

8.6.3 Create a Late Binding Assign

The AssignLate() function creates or converts an assign into late binding assign whose target is initially given by a path. After its first access, the assign reverts to a regular assign to the directory path refers to at this time. From that point on, the assign binds to the same target directory on the same volume, even if the volume is ejected or re-inserted into another drive. 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) /* since V36 */
D0                      D1   D2

BOOL AssignLate(STRPTR,STRPTR)
```

This function scans the device list for an assign given by name. The name shall not contain a trailing colon (":"). If such an assign does not yet exit, it creates a late-binding assign to path. While not required by this function, the path should better be an absolute path as otherwise resolution of the created assign can be very confusing — it is then relative to the current directory of the first process using the assign.

If an assign name already exists, it is canceled, all of its resources, including one or multiple locks are released, and it is converted into a late-binding assign to path.

Unfortunately, even in the most present release of AmigaDOS, this function has a race condition if locks of a previous assign are released and these locks are still in use by an application program.

If the function is successful, it returns a non-zero result code. On success, IoErr() is not set consistently and cannot be relied upon. The string path then has been copied into a buffer allocated by the function, and path remains available to the caller.

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 the assign name is longer than 30 characters, the error code is set to ERROR_INVALID_COMPONENT_NAME. If a DosList entry representing a handler, file system or volume of the same name already exists, the error code is ERROR_OBJECT_EXISTS.

8.6.4 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 converted into a multi-assign.

```
success = AssignAdd(name,lock) /* since V36 */
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` shall not contain a trailing colon (“`:`”), and the lock shall be a shared lock to a directory.

A `DosList` of the given `name` shall already when entering this function, and this `DosList` shall be a regular or a multi-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 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 an unsuitable type of assign.

8.6.5 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) /* since V36 */
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 along with all remaining resources and removed from the device list.

The `lock` remains available to the caller, regardless of the return code, its only purpose is to identify the directory to be removed; it does not need to be identical to the lock contained in the assign, but it shall be a lock on the same directory. This function uses the `SameLock()` function to compare `lock` and the candidate locks within the assign.

Identified locks to be removed from the assign are released with `UnLock()`, which causes, even in the most recent version of AmigaDOS, a race condition if these locks are still in use by an application program, e.g. because it received them through `GetDeviceProc()`.

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.

8.7 File System Support Functions

Functions in this section act on a file system as a whole; thus, they do not need a file or a lock to operate on, but modify the file system globally given a volume or device name.

8.7.1 Adjusting File System Buffers

The `AddBuffers()` function increases or reduces the number of buffers of a file system.

```
buffers = AddBuffers(filesystem, number) /* since V36 */
D0                   D1          D2

LONG AddBuffers(STRPTR, LONG)
```

This function adds `number` buffers to the file system responsible for the path given by `filesystem`. This should be an absolute path, e.g. the name of the device followed by a colon (“`:`”), such as “`DF0:`”. In case a relative path is provided, the buffer count of the file system responsible for the current directory of the calling process is modified. The file system object identified by this argument does not actually matter and does not even need to exist, and providing a complete path is not an error either. See also `GetDeviceProc()` in section **??** how a file system is determined from a path.

The `number` argument may be both positive — for adding buffers to the file system — or negative, to reduce the number of buffers. The purpose of these buffers is file-system dependent. The Fast File System in ROM uses it to buffer administrative information such as directory contents, and also blocks that describe the location of file content on the disk; thus adding more buffers can help to improve the performance of random-access into the file with `Seek()`.

A third purpose of the buffers is to store input and output data of the `Read()` or `Write()` functions if the source or target memory block is not aligned to block boundaries or if the source or target buffer is considered unsuitable for direct transfer to the underlying hardware exec device.

Even though this function only exists from AmigaDOS version 36 onward, the underlying packet, see section **??**, is also supported in all earlier versions of AmigaDOS, it is just not exposed as a function of the *dos.library*.

This function returns a non-zero result on success and 0 on failure. In first case, the *dos.library* autodocs state that the number of buffers is returned in `IoErr()`, though the FFS returns it as (primary) result code of this function instead, and the `AddBuffers` command of the Workbench also expects it there. On error, the return code is 0 and `IoErr()` delivers an error code. As Workbench programs depend on the primary result code, it is therefore suggested to accept this defect as a specification change[7].

8.7.2 Change the Name of a Volume

The `Relabel()` function changes the name of a volume a file system operates on.

```
success = Relabel(volumename,name) /* since V36 */
D0                D1       D2

BOOL Relabel(STRPTR,STRPTR)
```

This function relabels the volume that resides on the file system corresponding to the `volumename` path. This path is resolved through `GetDeviceProc()` and thus may be a relative or absolute path based on the device or volume name. As `volumename` is interpreted as a path, a device or volume name passed

[7] [**?**] recommends to check the return code for this function for `DOSTRUE` or `DOSFALSE`, and if it is unequal to these two values, accept it as buffer count. Otherwise, the buffer count is expected in `IoErr()` as described in the autodocs.

in shall include a colon (“:”) as it would be otherwise interpreted as a path relative to the current directory, and thus relabel the volume on which the current directory is located. The file system object identified by the path does not matter and does not even need to exist.

Beware, multiple volumes of the same name, but differing creation dates, can be known to the system. In case a volume name is provided as first argument, this function (and in general, `GetDeviceProc()` on which this function is based) affects the *first* volume of the given name on the device list. Thus, in case of doubt, a path based on a file system or handler name (e.g. “`DF0:`”) should be preferred.

The volume name of the medium or partition is then changed to `name`. Unlike the first argument, `name` is *not* a path and therefore shall *not* contain a colon (“:”) nor a slash (“/”) (or the component separator, in general). Note that not all file systems support volume names; this function fails if a file system does not.

Even though this function only exists from AmigaDOS version 36 onward, the underlying packet, see section **??**, is also supported in all earlier versions of AmigaDOS, it is just not exposed through a function of the *dos.library*.

This function returns a non-zero result code for success or 0 for an error. In case of failure, it sets `IoErr()` to an error code, otherwise to an undefined value.

8.7.3 Initializing a File System

The `Format()` function initializes a complete file system. The initialized medium or partition appears afterwards as completely empty, even though not all blocks are overwritten and only the previous administration information (if any) is lost.

```
success = Format(filesystem, volumename, dostype) /* since V36 */
D0                   D1          D2          D3

BOOL Format(STRPTR, STRPTR, ULONG)
```

This function erases all information on the medium or partition identified by `filesystem`, which is interpreted as a path. Thus, it may be a device or volume name, which shall then be terminated by a colon (“:”). However, all other paths also work; however, if they do not include a colon, the argument is interpreted as path relative to the current directory, and thus will initialize the file system responsible for the current directory.

Beware, multiple volumes of the same name, but differing creation dates, can be known to the system. In case a volume name is provided as first argument, this function (and in general, `GetDeviceProc()` on which this function is based) affects the *first* volume of the given name on the device list. Thus, in case of doubt, a path based on a file system or handler name (e.g. “`DF0:`”) should be preferred.

To block processes from accessing information on the file system while it is initializing, it shall be inhibited upfront, e.g by `Inhibit(filesystem,DOSTRUE)` or by lower level packet communication to the handler, see section **??**.

The `Format()` function *does not* attempt a low-level initialization of the corresponding medium; that is, it does not attempt to low-level format it at the physical layer as for example required when a floppy disk is prepared for initial use. This step needs to be performed manually by first blocking access of the file system to the floppy through `Inhibit()`, then initializing the physical layer through the exec device driver, e.g. by a `TD_FORMAT` IORequest, and then finally by calling this function.

The volume name of the medium or partition is initialized to `volumename`, which *shall not* contain a colon (“:”) nor a slash (“/”) (or the component separator, in general). Note that not all file systems support volume names. In such cases, this argument is ignored.

The `dostype` defines the flavor of file system created on the device if the file system allows multiple variations. The flavors the Fast File System supports along with other dos types are listed in table **??** in

section **??**. This argument corresponds to the DOSTYPE in the mountlist. File systems may also ignore it if they only support a single flavor.

Unfortunately, AmigaDOS does not provide an easy way to access the flavors supported by a file system. The Format command of the Workbench offers the types listed in the first half of table **??** if the mount entry of the file system indicates that it is the FFS, and otherwise does not offer any choices and just copies the dostype from the de_DosType of the DosEnvec structure, see also section **??**.

After initializing the file system, use Inhibit(filesystem,DOSFALSE) or the corresponding packet ACTION_INHIBIT to grant the file system access to the partition or medium again. See sections **??** and **??**.

After making the volume accessible again, it is possible that write access is not permitted immediately afterwards as the file system has to go through an initial validation phase. This usually takes only seconds as the FFS, for example, creates the bitmap only after the volume becomes available.

The Format() function and its underlying packet did not exist in AmigaDOS versions below 36. In such versions, the layout of the FFS and OFS was hard-coded into the Format command which wrote the root block manually, depending on file-system internals.

This function returns a Boolean success indicator that is non-zero on success or 0 on error. In either case, IoErr() is to an error code on failure, and to an undefined value on success.

8.7.4 Inhibiting a File System

The Inhibit() function disables or enables access of the file system to the underlying exec device driver. Typical application for this function are disk editors or file system salvage tools that require exclusive access to the file system structure; initializing a file system also requires this function, see section **??**.

```
success = Inhibit(filesystem, flag) /* since V36 */
D0                    D1       D2

BOOL Inhibit(STRPTR,LONG)
```

This call controls whether the file system identified by the path name given as filesystem is allowed to access the medium or partition it operates on. The filesystem argument is interpreted as a path through GetDeviceProc(). That is, the function resolves relative and absolute paths, device and volume names, and even assigns. As filesystem is interpreted as a path, a device or volume name passed in shall include a colon (“:”) as it would be otherwise interpreted as a path relative to the current directory and thus inhibit the file system responsible for the current directory of the caller.

Beware, multiple volumes of the same name, but differing creation dates, can be known to the system. In case a volume name is provided, this function (and in general, GetDeviceProc() on which this function is based) affects the *first* volume of the given name on the device list. Thus, in case of doubt, a path based on a file system or handler name (e.g. “DF0:”) should be preferred.

The flag argument controls whether access to the medium is allowed or disallowed. If flag is set to DOSTRUE, access is inhibited and the file system stops accessing the partition or volume. It also sets id_DiskType to the four-character code 'BUSY', see also section **??**. Once a file system is inhibited, application programs may access the exec device driver directly to access or modify blocks within the partition managed by the inhibited file system.

If flag is set to DOSFALSE, the file system is allowed to access to the medium again. The file system then performs a consistency check of the file system structure of the disk, i.e. validates it. This will require a couple of seconds within which write access to the medium is not possible.

Even though this function only exists from AmigaDOS version 36 onward, the underlying packet, see section **??**, is also supported in all earlier versions of AmigaDOS, it is just not exposed as a function of the *dos.library*.

This function returns a non-zero result code for success and then sets `IoErr()` to an undefined value. On error, it returns 0 and provides an error code in `IoErr()`.

Chapter 9

Pattern Matching

Unlike other operating systems, it is neither the file system nor the shell that expands wildcards — or patterns as they are also called. 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 can also be used to check whether a given string matches a pattern and thus can also be used to search for patterns within a text file.

The pattern matcher syntax is build on special characters or *wild cards* that define the rules by which strings match. A sequence of regular (non-wild card) characters and wild cards forms a pattern. AmigaDOS recognizes the following wild cards, defined in `dos/dosasl.h`:

`?` The question mark matches a single, arbitrary character within a string. 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 in AmigaDOS.

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

`()` The brackets bind a pattern together forming a single wild card. This is particularly useful for the hash mark "`#`" as it allows to formulate repeats of character sequences or patterns. For example, `#(ab)` indicates zero or more repeats of the character sequence `ab`, such as `ab`, `abab` or `ababab`.

`~` The ASCII tilde ("`~`") matches names that do not match the next wild card. 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 apostrophe ("`'`"), i.e. `[['-']]` matches the backslash ("\"), the opening and the closing bracket. 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 apostrophe (`'`) is the escape character of the pattern matcher and indicates that the next character is not a wild card of the matcher, but rather stands for itself. Thus, `'?` matches the question mark, and only the question mark, and no other character. The apostrophe only escapes the wild cards in this list, if a non-wild card character follows the apostrophe, it stands for itself. That is, the pattern `'a` stands for itself, i.e. a two-character sequence starting with the apostrophe.

`%` The percent sign ("`%`") matches the empty string. This is most useful with the vertical bar (see below) formulating alternatives. For example, `Tool(%|.info)` matches the file `Tool` and its icon.

`|` The vertical bar ("`|`") defines alternatives and matches the pattern to its left or the pattern 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 of the Shell; those are properties AmigaDOS inherited 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 the asterisk would be interpreted differently than potentially 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 match a string against a pattern. 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 to pre-process the pattern.

> *Only ISO-Latin Code points* 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. The code points in this range do not represent printable characters. The Fast File System also disallows such characters in file names. Compare also to section **??** defining usable characters in paths.

9.1 Scanning Directories

The prime purpose of the pattern matcher is to scan a directory, 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.

Scanning a directory requires the following steps:

First, the user shall initialize an `AnchorPath` structure. Only the elements `ap_Flags`, `ap_Strlen`, `ap_BreakBits` and `ap_FoundBreak` require initialization by the caller; the element `ap_Reserved` shall also be set to zero for forwards compatibility. All remaining elements are initialized by the *dos.library* itself. In the simplest case, the structure is allocated from exec with the `MEMF_CLEAR` flag.

This structure contains the state of the directory scanner, including a `FileInfoBlock` structure (see section **??**) describing the matched object. Optionally, the `AnchorPath` structure can also be configured to contain its complete (relative) path. The following paragraphs describe this structure in more detail.

Shall be Long-Word Aligned As the AnchorPath structure embeds 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 **??**.

Then MatchFirst() shall be called, retrieving an AnchorPath structure as second argument, see section **??**. It returns the first match of the pattern if there is any. Upon return, the AnchorPath structure contains information on the found object.

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, entering this directory if the flag is set, see section **??**. 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 shall be released by calling MatchEnd() described in section **??**. The AnchorPath structure is *not* released by this function, but must be disposed manually, e.g. by FreeMem().

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 semantics of the elements 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. These structures are 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. The ap_Base element 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 corresponding to the AChain structure, i.e. 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 described by the AChain structure, see section **??** for its definition.

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

ap_BreakBits of the AnchorPath structure shall be initialized to a signal mask that defines which signal bits abort a directory scan. This is typically a combination of signal masks defined in the dos/dos.h include file, e.g. SIGBREAKF_CTRL_C to abort on the Ctrl-C key combination 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, the following flags are defined in dos/dosasl.h:

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 can match. Otherwise, no directory scan is performed and the pattern is delivered as only match. The user may also set this flag to enforce a scan. This resolves 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 ap_Info indicates that a directory was found.

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 itself, it shall be cleared by the caller.

APF_NOMEMERR is an internal flag that shall not be interpreted; it is set if an error is encountered while scanning a directory. Errors indicated by this flag are not restricted to memory allocation errors.

APF_DODOT is, even though documented, not actually used. Its intended purpose was probably to emulate the Unix-style directory entries "." and ".." indicating the current and the parent directory.

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

APF_FollowHLinks may be set by the caller to indicate that hard links to directories shall be recognized as regular directories and may be recursively entered by setting APF_DODIR. Otherwise, hard links to directories are never entered. Note that not all file systems are able to distinguish between hard links and regular entries, so this functionality does not work reliable for all file systems.

Soft links to directories are never entered, this this cannot be enforced by any flag. A potential danger of links is that they can 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. It is safer to keep the APF_FollowHLinks flag cleared.

ap_Strlen is the size of the buffer ap_Buf that is optionally filled with the complete path of the matched entry, see below for a more detailed description. 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 complete 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 reported. If, however, the complete path is not required, this element shall be set to 0.

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

ap_Buf is filled with the complete 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 s bytes, in total sizeof(AnchorPath)+s-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 present, ap_Strlen shall be set to 0.

9.1.1 Starting a Directory Scan

The MatchFirst() function starts a directory scan, locating the first object in a directory matching a pattern.

```
error = MatchFirst(pat, AnchorPath) /* since V36 */
D0                  D1        D2

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

This function starts a directory scan, locating the first object matching the pattern pat. This pattern does *not* require pre-parsing (e.g. the functions in section **??**), i.e. MatchFirst() performs the initial step of pre-parsing the pattern and converting it to something suitable for the lower level pattern matcher.

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. Furthermore, ap_FoundBreak shall be set to 0, and ap_Strlen to the size of the buffer ap_Buf which is filled by the full path of the matching objects. If the path is not required, ap_Strlen shall be set to 0. ap_Flags shall be set to the flags you need, see the parent section. For forward compatibility, ap_Reserved shall also be set to zero.

Unlike many other functions, MatchFirst() returns an error code directly, and not a success/failure indicator. That is, 0 indicates success, and everything else an error code IoErr() would provide for many other functions of the *dos.library*. In particular, if ERROR_BREAK is returned in case any of the signal bits in ap->ap_BreakBits have been received during the scan. If pat is a wild card that could match multiple objects and not a single match is found, the error code will be ERROR_NO_MORE_ENTRIES. If pat does not contain a wild card, the function will attempt to Lock() the provided object name pat directly, and ERROR_OBJECT_NOT_FOUND will be returned instead in this situation. Unfortunately, this has the side effect that a dangling soft link, i.e. a link whose target is not available, will also trigger a return code of ERROR_OBJECT_NOT_FOUND. In some cases, this is undesirable, e.g. if a scan is made to delete such a link. To work around this issue, the input should be forcefully turned into a pattern and the access should be retried:

```
#define ENVMAX 112
LONG MyMatchFirst(const UBYTE *name,struct AnchorPath *ac)
{
    LONG error,trc;
    UBYTE path[ENVMAX];

    error = MatchFirst(name,ac);
    if (error == ERROR_OBJECT_NOT_FOUND &&
        (trc = strlen(name)) + 4 < ENVMAX) {
      strcpy(path,name);
```

```
      strcpy(path + trc,"(|)");
      MatchEnd(ac);
      error = MatchFirst(path,ac);
    }

    return error;
}
```

The above code checks the return code for the first attempt to match, and if an error code is generated that indicates that a single object could not be found, "`(|)`" is attached to the pattern. This matches the same object as the original name, but is a wildcard the pattern matcher will resolve by scanning a directory instead of locking the object itself.

On success, `ap->ap_Info.fib_FileName` contains the name of the first matched object; the directory containing the found object is available in `ap->ap_Current->an_Lock`, represented as a lock. You would typically set the current directory to this lock, then access the found object, and then revert the directory. This lock *shall not* be released; it is implicitly released by the pattern matching functions when changing the directory or terminating the scan.

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`. Note that `ap_Strlen` is the byte size of the buffer, i.e. an additional `NUL` for string termination shall be accounted for.

If the matching object is a directory, i.e. `ap->ap_Info.fib_DirEntryType` is positive and not equal to `ST_SOFTLINK` identifying it as a soft link, the caller may request to enter it by setting `APF_DODIR` in `ap_Flags`. If the matching object can be identified as a hard link to a directory, that its type is `ST_LINKDIR`, and `APF_FollowHLinks` is set, then `APF_DODIR` will also enter such linked directories.

Beware, however, that first not all file systems are able to distinguish between regular directories and links to directories, i.e. the `ext2` and related systems from the Unix world will not; furthermore, entering hard links can cause an endless recursion if the hard link goes to a parent directory of the current directory.

9.1.2 Continuing a Directory Scan

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

```
error = MatchNext(AnchorPath) /* since V36 */
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 indicator. In particular, if `ERROR_BREAK` is returned in case any of the signal bits in `ap_BreakBits` have been received.

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

The following code provides a simple example for a directory scan. It uses the `D_S` macro from section **??** to place the `AnchorPath` structure on the stack and align it properly.

```
/*
** Scan a directory tree recursively for entries matching
** the supplied pattern, and print the objects found.
*/
LONG ScanDirectories(const UBYTE *pat)
{
  LONG error;
  D_S(struct AnchorPath,ac);

  /* Minimal initialization */
  ac->ap_BreakBits  = SIGBREAKF_CTRL_C;
  ac->ap_FoundBreak = 0;
  ac->ap_Flags      = 0;
  ac->ap_Reserved   = 0;
  ac->ap_Strlen     = 0;

  for(error = MatchFirst(pat,ac);error == 0;error = MatchNext(ac)) {
    if (ac->ap_Flags & APF_DIDDIR) {
      Printf("leaving %s\n",ac->ap_Info.fib_FileName);
      ac->ap_Flags &= ~APF_DIDDIR;
    } else if (ac->ap_Info.fib_DirEntryType > 0 &&
       ac->ap_Info.fib_DirEntryType != ST_SOFTLINK &&
       ac->ap_Info.fib_DirEntryType != ST_LINKDIR) {
      Printf("entering %s\n",ac->ap_Info.fib_FileName);
      ac->ap_Flags |= APF_DODIR;
    } else {
      BPTR lock = CurrentDir(ac->ap_Current->an_Lock);
      Printf("%s\n",ac->ap_Info.fib_FileName);
      /* Do something on ac->ap_Info.fib_FileName */
      CurrentDir(lock);
    }
  }
  MatchEnd(ac);

  return error;
}
```

This function returns the error code that lead to abortion of the scan, it does not attempt to detect a dangling soft link if `pat` does not contain a wildcard, see section **??** for a workaround. It neither attempts to detect endless recursions on file systems that cannot identify hard links to directories. Note that each directory will be visited twice: Once when entering it, and once when leaving it.

The `MatchFirst()` and `MatchNext()` functions compare file names with the pattern in a case-insensitive way, thus upper and lower case are considered identical. For that, characters of the file name and the pattern are converted to upper case by means of the `ToUpper()` function of the *utility.library* which is potentially localized by the *locale.library* if loaded. Note that this is potentially *different* from how the Fast File System converts characters to upper case, namely by the algorithm that is used in computing the hash key, see section **??**. Thus, the pattern matcher can potentially identify file system objects as matching the pattern the file system itself would consider non-matching, and vice versa. This is most critical for the non-international versions of the FFS and OFS, i.e. the first two (legacy) entries of table **??** in section **??** which consider upper and lower characters from the ISO-Latin-1 supplement block as non-identical. For the remaining flavors, the FFS interpretation of case-insensitive comparison is, most likely, the correct one assuming the locale is also based on ISO-Latin-1. Compare also with section **??** for similar quirks.

9.1.3 Terminating a Directory Scan

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

```
MatchEnd(AnchorPath) /* since V36 */
             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 (e.g. the desired object has been detected and no further matches are required, or the scan has been aborted by `Ctrl-C`).

This function does not release the `AnchorPath` structure, and it may be reused after re-initializing it as described in section **??**.

9.2 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 pre-parsed, 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 against the pattern. The wild cards possible in a pattern are those from the start of chapter **??**. While the token encoding is documented in `dos/dosasl.h`, it should be considered internal, see also the warning at the end of the introduction in chapter **??** and and also section **??**. As the tokens use the C1 control set of ISO-Latin 1, patterns should be restricted to printable ISO-Latin characters.

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

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.

9.2.1 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. The buffer shall be at least `2 + (n << 1)` bytes large, where `n` is the length of the input pattern. While this buffer size is currently sufficient, future implementations can require larger buffers; the result code shall therefore be checked for -1 to determine possible buffer overruns (see below) and the buffer should then be enlarged and the call retried. The result code `IsWild` is one of the following:

`1` is returned if the source contained wildcards and was tokenized successfully.

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. Then, IoErr() provides an error code. Possible errors include ERROR_LINE_TOO_LONG if the target buffer is not large enough to keep the parsed pattern, and ERROR_BAD_TEMPLATE if the template is ill-formed, for example if the "|" token is not enclosed in brackets.

9.2.2 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. The buffer shall be at least 2 + (n << 1) bytes large, where n is the length of the input pattern. While this buffer size is currently sufficient, future implementations can require larger buffers; the result code shall therefore be checked for −1 to determine possible buffer overruns (see below) and the buffer should then be enlarged and the call retried. 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. Then, IoErr() provides an error code. Possible errors include ERROR_LINE_TOO_LONG if the target buffer is not large enough to keep the parsed pattern, and ERROR_BAD_TEMPLATE if the template is ill-formed, for example if the "|" token is not enclosed in brackets.

9.2.3 Match a String against a Pattern

MatchPattern() matches a string against a tokenized pattern prepared by ParsePattern(), taking the case of the string and the pattern into consideration.

```
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.

9.2.4 Match a String against a Pattern ignoring Case

The `MatchPatternNoCase()` function matches an input string against a tokenized pattern prepared by `ParsePatternNoCase()`, ignoring the case of the string and the pattern.

```
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()`.

Case-insensitive comparison is defined by converting characters of the string and the pattern to upper case by the `ToUpper()` function of the *utility.library* which is potentially localized by the *locale.library* if loaded.

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 10

Processes

Processes are extensions of exec *tasks*. They include a message port in the form of a `MsgPort` structure for inter-process communication to handlers, a current directory to resolve relative paths, the standard input, output and error streams 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;

    /* the following definitions are new in V36 */
    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 */
```

Many functions of the *dos.library* can only be called from processes as they depend on and update the elements of the process structure shown above. The `AddPart()`, `FilePart()` and `PathPart()` functions from sections **??** and following are noteworthy exceptions as they do not interface to handlers but only operate on strings. The `DoPkt()` function from section **??** is also prepared to accept ordinary exec tasks as callers, but then cannot set the secondary result code, i.e. the return value of `IoErr()`, as it is represented as an element in the process structure. To be able to initiate processes, `CreateNewProc()` and `CreateProc()` specified in sections **??** and **??** are also callable from tasks; the latter function also existed in AmigaDOS versions 34 and below, but was *not* task-callable there.

The elements of this structure are as follows:

`pr_Task` is the exec task structure defined in `exec/tasks.h` and discussed in more detail in [**?**]. To distinguish between an exec *Task* and a *Process*, the `pr_Task.tc_Node.ln_Type` element of the latter is set to `NT_PROCESS` instead to `NT_TASK`. Prior starting the process, the *dos.library* also pushes the stack size onto the stack, i.e. `(ULONG *)(pr_Task.tc_Upper)[-1]` contains the size of the stack in bytes. Some binaries, in particular those compiled with the Aztec (Manx) compiler depend on this value deposited there.

The *dos.library* also installs a custom exception handler into the `tc_TrapCode` element of the `Task` structure which is called upon CPU exceptions caught by the exec kernel. It shows the "Software Failure" requester, allowing users to suspend the process or reboot the system. When selecting suspension, the code runs into a `Wait(0)` which waits forever. Otherwise, the exception handler runs into the `Alert()` function of exec which will show the usual red software failure, with the option to enter the ROM debugger. Unfortunately, AmigaDOS versions 45 and below had a defect here that did not interpret the 68K exception stack frames correctly and thus left unusable data on the stack before entering the ROM debugger.

`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 chapters **??** and **??**.

`pr_Pad` is unused and only included in the structure to ensure that all following elements are aligned to long word boundaries.

`pr_SegList` contains an array of segments which are used to populate the BCPL Global Vector (see below) by the BCPL runtime binder. The usage of the segment array and the Global Vector is questionable as version 47 removed the last BCPL compiled handler and by that also the last BPCL compiled program from the system. Its contents will be briefly described in the following for historical purposes:

The first entry in the segment array is a 32-bit integer indicating the number of elements the BCPL runtime binder scans for populating the Global Vector, the remaining entries are BPTRs to segment lists. A segment list is a BPTR linked list of program sections, its structure is explained in more detail in chapter **??**. Some entries in the segment array can also be `ZERO` indicating that the corresponding entry is currently not used. Entries 1 and 2 are system segments containing AmigaDOS functions, namely the BCPL "kernel library" *klib* and the BCPL support library *blib*. Functions in these two segments are either trivial (e.g. long multiplication) or available as functions the *dos.library* as well. They are always included in the count at offset 0 of the segment array.

Entry 3 is populated by the segment list of the program, see chapter **??**, the process is executing. It is only included in the count if the started process requested BCPL binding, thus for handlers with a `GLOBVEC` entry in their mountlist of 0 or -3, see table **??** in chapter **??**. For BCPL code, the program segments are also used to populate the Global Vector, see section **??** for details, and thus the count value is then 3. Segments of C and assembler programs lack BCPL runtime binding information, they are excluded from the count and thus ignored by the binder. The count for them is therefore 2.

Entry 4 contains the segment list of the shell if the process represents a shell, or is currently executing a command line program overlaying the shell process. The shell startup code moves its own segment from entry 3 to 4, see also section **??**, leaving room for the segment of command line programs in entry 3.

The above only reflects the current (version 47) usage of the pr_SegList, and later versions of AmigaDOS can populate this vector differently or abandon it completely as it does not contain relevant information for non-BCPL programs, and the BCPL runtime binding mechanism will likely be phased out at some point, too.

pr_StackSize is the size of the process stack in bytes which is always a multiple of 4. In case the process is executing a command line program, the stack size recorded here is the stack size of the shell, and not the stack size of the command line tool executing in the context of the shell. Thus, this element is unsuitable to retrieve the stack size of the currently running process in general. Instead, the elements tc_SPLower and tc_SPUpper of the task structure shall be used, which are updated properly when providing a new stack by StackSwap(), or RunCommand() is used to execute commands in the context of the shell and provides a custom stack for them. The latter, namely the stack size for command execution, is adjustable by means of the Stack command.

pr_GlobVec is another BCPL legacy. It contains the *Global Vector* of the process. For binaries using the BCPL linkage, this is a custom-build array of global data and function entry points populated from the pr_SegList array through the BCPL runtime binder introduced in section **??**. For C and assembler binaries, the Global Vector is the system shared vector; it contains *dos.library* global (BCPL) functions and data required by the *dos.library*, such as base pointers to system libraries. As no particular advantage can be taken from this vector (anymore), it should be left alone and its entries are not documented here.

pr_TaskNum is an integer allocated by the system for processes that execute a shell, or command line programs executed in the context of a shell. The number here corresponds to the integer printed by the Status command. It is the closest AmigaDOS analog of a process ID. The FindCliProc() function can be used to retrieve the process corresponding to a task number, it is specified in section **??**. Note that AmigaDOS does not use task numbers consistently, i.e. processes that are started from the Workbench, handlers, file systems or processes that have been created by CreateNewProc() or CreateProc() are not identified by a task number. In such a case, this element 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 opposite direction, it is the start of the BCPL stack. While it is initialized, it is not used by the *dos.library* at all. It neither reflects the lower end of the stack of a command line program executing in the context of a shell process. In such a case, pr_StackBase is the lower end of the stack of the shell. The lower end of the stack of the process may instead be found in the tc_SPLower element of pr_Task, which is always correct, regardless of whether the process is run from the Workbench, or executing a command overlaying the shell process. pr_StackBase should thus be left alone as it serves no practical purpose.

pr_Result2 is the secondary result code set by many functions of the *dos.library*. The value stored here is delivered by IoErr() specified in section **??**, and it can be altered by SetIoErr(), see section **??**. The above two accessor functions should be preferred to accessing this element directly.

pr_CurrentDir is the lock representing the current directory of the process. All relative paths are resolved through this lock, i.e. they are relative to pr_CurrentDir. If this element is ZERO, the current directory is the root directory of the file system whose MsgPort pointer is 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. As for the above, this element should not be read or modified directly, instead the accessor functions GetCurrentDir() from section **??** and CurrentDir() from section **??** should be preferred.

pr_CIS is file handle of the standard input stream of the process. It should preferably accessed through the Input() function specified in section **??**. The input file handle can be ZERO in case the process does not have a standard input stream. This is *not* equivalent 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(), see section **??**.

pr_COS is the file handle of the standard output stream of the process. It should preferably be obtained through the Output() function of the *dos.library* specified in section **??**. The output file handle 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(), see section **??**.

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() from section **??**. The pointer stored in this element should be preferably obtained by GetConsoleTask(), see section **??**.

pr_FileSystemTask is the MsgPort of the file system that is contacted in case a path relative to the ZERO lock is resolved, i.e. if pr_CurrentDir is ZERO. This element is initialized to the MsgPort of the file system the system was booted from, but can be changed by SetFileSysTask(), see section **??**. This element is also returned by GetFileSysTask() specified in section **??**, and access through this function should be preferred.

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 element is ZERO. This is for example the case for programs started from the Workbench, or handlers and file systems. This structure is specified in in section **??**, and it should preferably be obtained through the Cli() function of the same section. There is no corresponding setter function, indicating that this element shall not be modified by user code at all.

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 overlaying the process, and used there to restore the previous stack frame by the Exit() function from section **??**. 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 code does not, however, release any resources obtained by user code, and for C programs the C standard library function exit() (with a small-case "e") is in almost all cases the better (and correct) choice. BCPL code or custom startup code could deposit here a pointer to a BCPL stack frame for a custom shutdown code, but as the entire BCPL activation and shutdown mechanism is a legacy construction, this element should be better left alone.

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

```
struct BCPLStackFrame {
       ULONG bpsf_StackSize; /* in bytes */
       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, the latter value is placed in the CPU register A7.

pr_PktWait is a function that is called by the *dos.library* when waiting for a packet to return from a handler or file system. Packets are used by the *dos.library* to communicate with handlers, see also chapters **??** and **??**. If this is NULL, the system default function is used, which waits on the arrival of a message on pr_MsgPort. The signature of this function is

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

struct Message *(*pr_PktWait)(void)
```

that is, no particular arguments are delivered, the process structure of the calling process must be obtained from the *exec.library*, and the message received shall be delivered back into register D0. From this, the *dos.library* retrieves the packet. The returned pointer shall not be NULL, rather, this function shall block until a message has been retrieved. For details on packets and inter-process communication through packets, see the DoPkt() function in section **??** and chapter **??** in general.

pr_WindowPtr is a pointer to an *intuition* Window structure, see intuition/intuition.h, which will borrow its title and its screen to error requesters. If this is NULL, error requesters appear on the default public screen, usually the Workbench screen; if this is set to (APTR)(-1L), error requesters will be suppressed at all, and the implied response to them is to cancel the operation. All other values will redirect the requester to the same screen the window pointed to by this element is located on. The workings of error requesters is specified in more detail in the description of the ErrorReport() function in section **??**.

Particular care shall be taken when replacing pr_WindowPtr within command line executables as commands only overlay the shell process, and thus share the same process with the shell and all other commands executing within the same shell. If, upon termination of a command, the window pointed to by this element is closed, the next command could find an invalid pr_WindowPtr in its process. The generation of an error requester would then crash the machine. Shell commands shall therefore store the initial value of this pointer prior to modification, and restore it back when exiting.

Also, redirection of error requesters only works for requesters that are generated in the context of the calling process. Requesters the file system itself displays, for example if a particular block on a medium is corrupt and unreadable, cannot be redirected by means of pr_WindowPtr.

At present, there are no accessor functions for this pointer, it can only be read and modified by accessing the process structure itself.

All subsequent elements were added in AmigaDOS version 36 and are not available for earlier releases:

pr_HomeDir is the lock to the directory containing the command or binary that is currently executing as this process, if such a directory exists. It is ZERO if the command was taken from the list of resident commands, see section **??**, or is a shell built-in command. This lock is filled in by the shell or the Workbench when loading an executable program. It is used to resolve paths relative to the PROGDIR pseudo-assign, see chapter **??**. If this lock is ZERO, any attempt to resolve a path within PROGDIR: will create a request to insert a volume named PROGDIR[1]. This lock shall not be released nor altered by the process; instead, the *dos.library* unlocks it when the process dies.

pr_Flags are system-use only flags that shall not be accessed, modified 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 element or extend the flags stored here. Table **??** lists the currently used flags defined in dos/dosextens.h:

Table 10.1: Process Flags

Flag	Meaning
PRF_FREESEGLIST	Unload the segment pr_SegList[3]
PRF_FREECURRDIR	Unlock the current directory pr_CurrentDir
PRF_FREECLI	Unlink the CLI structure pr_CLI
PRF_CLOSEINPUT	Close the standard input stream pr_CIS
PRF_CLOSEOUTPUT	Close the standard output stream pr_COS
PRF_FREEARGS	Release CLI arguments pr_Arguments
PRF_CLOSEERROR	Close the standard error output pr_CES

The PRF_FREESEGLIST flag requests that the program segment is released with UnLoadSeg() when the process terminates. This segment is taken from pr_SegList[3], see above. This flag corresponds to the NP_FreeSeglist tag of the CreateNewProc() function from section **??**.

The PRF_FREECURRDIR flag releases pr_CurrentDir via UnLock() upon termination of the process. This flag is set when creating processes by CreateNewProc() and a non-ZERO lock is installed as current directory. The only way how to clear this flag is to install a ZERO lock by the NP_CurrentDir tag, see also section **??** and the description of this tag.

[1] This is probably unintended and a defect of GetDeviceProc().

The PRF_FREECLI flag requests to release the CommandLineInterface structure and all structures referenced by it upon termination of the process. This flag is set if building a CLI structure is requested by the NP_Cli tag.

The PRF_CLOSEINPUT flag closes the pr_CIS standard input stream upon termination of the process. This flag corresponds to the NP_CloseInput tag of CreateNewProc().

The PRF_CLOSEOUTPUT flag closes the pr_COS standard output stream upon termination of the process. It corresponds to the NP_CloseOutput tag of CreateNewProc().

The PRF_FREEARGS flag requests to release the command line arguments stored in pr_Arguments upon process termination. This flag is set implicitly by providing arguments through the NP_Arguments tag of CreateNewProc().

The PRF_CLOSEERROR flag closes the pr_CES standard error stream upon termination of the process. This flag was introduced in AmigaDOS version 47. It corresponds to the NP_CloseError tag of CreateNewProc().

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

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

LONG ExitFunc(LONG,LONG)
```

The value of rc is the process return code, i.e. the value left in register D0 when the code drops off the final RTS. exitdata is taken from pr_ExitData. The returncode allows the exit function to modify the process return value; however, currently this value has no use as the *dos.library* shutdown code directly runs into RemTask() of the *exec.library* which discards all register values of the terminating task. The process exit function is defined by the NP_ExitCode tag of the CreateNewProc() function, see section **??**.

pr_ExitData is used as argument for the pr_ExitCode() function, see above. It is defined by the NP_ExitData tag of CreateNewProc().

pr_Arguments is a pointer to the command line arguments of the process if it executes a command started from the shell. It is a NUL terminated string *not* including the command name itself. The latter can be retrieved from cli_CommandName, see section **??**. The argument string can also be found in register A0, or in the buffer of the pr_CIS file handle. The ReadArgs() function takes it from the latter source, and not from pr_Arguments. If the process does not execute a command from the shell, this element remains NULL. The command line arguments can also be retrieved through GetArgStr() specified in section **??**. Even though this is possibly a bit pointless, the argument string can also be set by SetArgStr() from section **??**, though arguments installed by this function will not reach ReadArgs().

pr_LocalVars is a MinList structure, as defined in exec/lists.h, that contains all local variables and alias definitions specific to the shell and commands executed by it in the context of this process, see section **??** and following. The structure of a node in this list is specified in section **??** and defined in dos/var.h. Access to this list is not protected by a semaphore, and thus, it shall not be accessed from any other process but the one represented by the containing Process structure. Even more so, the list and its contents should not be accessed directly at all, but through the accessor functions listed 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 goes usually to the console the process runs in, if such a console exists. This handle can be changed by SelectError() specified in section **??**. If pr_CES is ZERO, processes should print errors through pr_COS instead; preferably, processes should use the ErrorOutput() function defined in section **??** to get access to the standard error stream. This

function will also implement the fall-back to the standard output stream in case no standard error stream is available. Even though this element already existed in AmigaDOS version 36, it only came in use from AmigaDOS version 47 onward.

> *Prefer Accessor Functions* The elements of the `Process` structure should not be accessed directly if an accessor function is available. Thus, prefer `Input()` to `pr_CIS` and `Cli()` to `pr_CLI`, etc. There are only very few elements in this structure which should be accessed directly: `pr_WindowPtr` controls the generation of error requesters, and `pr_TaskNum` provides the task ID of a shell process. All remaining elements are either accessible through functions of the *dos.library* or are BCPL legacies that serve little practical purpose today.

10.1 Creating and Terminating Processes

AmigaDOS provides several functions to create processes: `CreateNewProc()` is the newer and most flexible function for launching a process, taking parameters in the form of a tag list. The legacy function `CreateProc()` supports less options, but is available under all AmigaDOS versions. Creating shells and running shell scripts by the `System()` function implicitly also creates processes, but it is not discussed here, but in chapter **??**. However, `System()` shares a couple of tags with `CreateNewProc()` to create the process running the shell.

There is not a single function to delete a process. Processes die whenever their execution drops off at the end of the `main()` function, or whenever execution reaches the final `RTS` instruction of their program code. The `Exit()` function only terminates the calling program, but cannot be used to shut down any other process. As it neither releases any resources beyond those allocated implicitly by AmigaDOS when starting the process or command, its practical use is limited. Processes shall not be terminated by the *exec.library* function `RemTask()` either as this function will miss to release even the resources allocated by the *dos.library* when creating processes, leave alone resources allocated within the executed program code.

`CreateProc()` and `CreateNewProc()` alone are not sufficient to launch applications intended to be run from the Workbench, nor to start command line programs expecting a shell environment. While the Workbench also goes through `CreateNewProc()`, the delivery of a `WBStartup` message as documented in `workbench/startup.h` is necessary in addition. Such a message can be created manually and send to the process once started. The `WBLoad` program added to AmigaOs version 45 emulates the Workbench startup mechanism by such a technique.

For shell commands, the `RunCommand()` function of section **??** or the `System()` function described in section **??** are more appropriate. The former overlays the calling process with a command represented by a segment list, e.g. obtained from `LoadSeg()`, but requires already the presence of a shell environment on the caller side. The latter creates a shell, loads one or multiple commands from within this new shell and executes them there. More details on these functions are found in sections **??** and **??**.

10.1.1 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 taking parameters from this list.

```
process = CreateNewProc(tags) /* since V36 */
D0                        D1

struct Process *CreateNewProc(struct TagItem *)

process = CreateNewProcTagList(tags) /* since V36 */
```

```
D0                                D1

struct Process *CreateNewProcTagList(struct TagItem *)

process = CreateNewProcTags(Tag1, ...) /* since V36 */

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

The above functions go all through the same entry point of the *dos.library*, only 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 first two functions are identical and differ only by name.

The following tags are recognized by the function and are defined in `dos/dostags.h`:

`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 provided segment list is released via `UnLoadSeg()` when the process terminates. Unlike what the official documentation claims, the default value of this tag is `DOSFALSE`, i.e. the segment list is *not* released by default. The segment list is never released if creating the process failed, regardless of this tag.

`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. Either `NP_Entry` or `NP_Seglist` shall be included in the tags. If `NP_Entry` tag is present, then `NP_FreeSeglist` shall *not* be included, or shall be set to `DOSFALSE`.

`NP_Input` sets the input file handle, i.e. `pr_CIS`, of the process to be created. This tag takes a BPTR to a `FileHandle` structure. The default is to open a file handle to `NIL:`.

`NP_CloseInput` selects whether the input file handle 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, no matter which handle the process left in `pr_CIS` when terminating. If creating the process failed, a handle provided by `NP_Input` remains available to the caller and will not be closed. The handle to `NIL:` that is used as default will, however, be cleaned up on error.

`NP_Output` sets the output file handle, i.e. `pr_COS`, of the process to be created. This tag takes a BPTR to a `FileHandle` structure. The default is to open a file handle to `NIL:`.

`NP_CloseOutput` selects whether the output file handle 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 in `pr_COS` when the created process terminates, even if it is not the file handle passed into `NP_Output`. If creating the process failed, a file handle provided through `NP_Output` remains open and available to the caller. The file handle to `NIL:` that is used as default will, however, be cleaned up on error.

`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 `FileHandle` structure. The default is to set the error output handle to `ZERO`. This tag was not implemented prior to AmigaDOS version 47.

`NP_CloseError` selects whether the error file handle will be closed when the process terminates, no matter whether it was set by a tag or by the process itself during its life-time. 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. This tag was also introduced in AmigaDOS version 47.

`NP_CurrentDir` sets the current directory of the process to be created. The argument is a lock which is absorbed into the created process, and will also be released by it. The default is to duplicate the current directory of the caller with `DupLock()` if the caller is a process, or set it to `ZERO` if the caller is only a task. The current directory of the process, i.e. `pr_CurrentDir`, is released when the process terminates, unless the initial current directory is `ZERO`. The provided lock is also released if creating the new process failed.

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 determining the name of the new process. This string is copied before the process is launched, and the copy is released automatically when the process terminates or creating the process fails; thus, the original string remains available to the caller. 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 19. 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 "$\star$" or a path relative to CONSOLE:, it will use the specified handler. The default is to use the console handler of the caller if it is process, or NULL if the caller is only a task.

While not explicitly available as a tag, the pr_FileSystemTask of the created process is set to the default file system of the calling process, or to the default file system from the *dos.library* if the caller is a task. This file system is contacted to resolve paths relative to the ZERO lock, and is typically also the file system responsible for the SYS assign.

NP_WindowPtr specifies a pointer to a Window structure that determines the title of and the screen on which error requesters will be displayed, see also section **??**. If this pointer is NULL, requesters will be shown on the default public screen — that is, typically the Workbench — and if it is -1, error requesters are suppressed and the *dos.library* reacts as if the user canceled the requester. The argument of the tag will be installed in the pr_WindowPtr of the new process. If no tag is provided, the default is to copy the value from the calling process if its window pointer is NULL or -1, or set to NULL otherwise. To set the pr_WindowPtr of the created process to that of the caller, the tag and the window pointer shall 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 otherwise the calling process need to ensure that its own window remains at least as long open as the created process keeps running.

NP_HomeDir sets the pr_HomeDir lock which is used to resolve paths relative to the PROGDIR pseudo-assign. The default is to duplicate pr_HomeDir of the calling process, or ZERO in case the caller is a task. This lock is released when the process terminates, even if it is the lock of the caller. Should the caller still require the lock, it needs to be copied by DupLock() upfront. The provided lock is released even if creating the new process fails.

NP_CopyVars determines if the local shell variables and alias definitions in pr_LocalVars of the calling process are copied into the new process. If set to non-zero, a deep copy of the variables of the calling process are made, otherwise the new process does not receive any shell variables and alias definitions. The latter also happens if the caller is a task and not a process. The variables are automatically released when the new process terminates, and once the new process is running, two independent sets of variables and alias definitions exist.

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 element. 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 possible. If the tag value is 0, no CLI structure will be created. The latter is also the default.

NP_Path provides a chained list of locks within which commands for shell execution 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 request a shell environment. This chained list is *not* copied, only referenced, 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 is equipped with a CLI structure. This (implicit) copy is, of course, released on error.

NP_CommandName provides the name of the command being executed within the shell environment if NP_Cli indicates that one is to be created. This tag does not actually *load* any command, it only provides an identification within the shell which command is running, see also section **??**. 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 the caller is not part of a shell or is a task. The string is copied, and the original 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 containing all arguments to be received by the created process, *excluding* the command name itself which is set by the above tag. If provided, the arguments are copied into pr_Arguments of the new process, and the string and its lengths will be loaded into registers A0 and D0 before execution of the process begins. If NP_Arguments are non-NULL, the tag value of NP_Input, if it is included, shall not be ZERO. This is because the arguments are additionally copied into the buffer of the input file handle to make them available to ReadArgs(), or any other function that reads from pr_CIS using buffered I/O functions, see section **??** for details. The arguments are always copied by this function and the string thus remains available to the caller, the copy will released when the created process terminates or when creation of the process failed.

NP_ExitCode determines a pointer to function that is called when the created process terminates. It is installed into pr_ExitCode. See chapter **??** 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. Details are again at the beginning of chapter **??**.

While dos/dostags.h also defines 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, thus one cannot easily determine the cause of the error.

> *What's Released on Error?* Unfortunately, CreateNewProc() is not very consistent on which resources it releases in case it cannot create a process. The locks representing the current directory and the home directory are released in such a case, all other resources remain available to the caller.

10.1.2 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 function that is not as flexible as CreateNewProc().

```
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 of stackSize bytes will be allocated for the process.

The Process structure is initialized as follows: The task name is set to a copy of name, and this copy is released when the process terminates. pr_ConsoleTask and pr_WindowPtr are copied from the calling process, or are set to NULL respective 0 if called from a task. The element pr_FileSystemTask

is also copied from the calling process, or is initialized from the default file system from the *dos.library* if called from a task. AmigaDOS versions 34 and below *do not* support calling `CreateTask()` from a task at all and would simply crash.

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 not initialized and left at `ZERO`. No command line arguments are provided to the process, and no shell variables are copied from the caller either.

The passed in `seglist` is *not* released upon termination. This should be done manually, though `CreateProc()` does not offer a protocol to learn when the created process terminates and the segment list containing the running code can be safely released again. The Workbench startup protocol described in [**?**] may be used as a blue-print for a similar mechanism, though `CreateNewProc()` offers a more practical solution by the `NP_FreeSeglist` tag and should be preferred, unless compatibility to AmigaDOS 34 and below is required.

If `CreateProc()` 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.

10.1.3 Terminating a Process

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

However, this function does not release any resources except those implicitly allocated when creating the process through `CreateNewProc()` or `CreateProc()`, or the resources created when starting the program from the shell through `RunCommand()`. As it misses to release resources allocated by the program itself or the compiler startup code, this function *should not be used* and rather a compiler or language specific shutdown function should be preferred. Ideally, programs should simply return from their `main()` function to the operating system. The C standard library provides `exit()` (with smaller case "e") which releases resources obtained through the ANSI C environment.

```
Exit( returnCode )
      D1

void Exit(LONG)
```

This call either terminates the calling process, in which case the argument is ignored, or returns to the shell, then delivering `returnCode` as result code. It locates the BCPL stack frame at `pr_ReturnAddr`, removes it, initializes the previous stack linked there and then returns to whatever function created the previous stack frame; see the beginning of chapter **??** how the stack frame looks like. The function returned to is typically either the process shutdown code of AmigaDOS, or the shell command shutdown code installed by `RunCommand()`. In the former case, `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 and historically exported from there to the *dos.library* interface; BCPL programs would typically overload its entry to customize their shutdown activities. ANSI-C offers with the `atexit()` function a similar mechanism. `Exit()` is therefore less relevant (if relevant at all) to C or assembler programs.

10.2 Process Properties Accessor Functions

Many elements of the `Process` structure described in section **??** are accessible through getter and setter functions. They implicitly relate to the calling process, and are the preferred way of getting access to its properties and state. The functions listed in this section do not touch `IoErr()` unless explicitly stated.

10.2.1 Retrieve the Process Input File Handle

The `Input()` function returns the input file handle of the calling process 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. It returns `pr_CIS` of the calling process, which is approximately the `stdin` of ANSI-C. Depending on process creation, this file handle will be closed by the process shutdown code or the calling shell and thus should not be closed explicitly by the caller. The standard input stream may be changed through `SelectInput()`.

For command line programs started from the shell, this file handle is connected to the console, unless it has been redirected by the "<", "<<" or "<>" operators, see section **??**. Programs started from the Workbench do not receive an input file handle at all, `pr_CIS` remains `ZERO` unless explicitly set[2].

10.2.2 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) /* since V36 */
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. It sets `pr_CIS` of the calling process. It is advisable to replace the input file handle to its original value before terminating a running program.

10.2.3 Retrieve the Output File Handle

The `Output()` function returns the output file handle of the calling process 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. It returns `pr_COS` of the calling process, which is approximately the `stdout` of ANSI-C. Depending on process creation, this file handle will be closed by the process shutdown code or the calling shell and thus should not be closed explicitly by the caller. The standard output stream may be changed through `SelectOutput()`.

For command line programs started from the shell, this file handle is connected to the console, unless it has been redirected by the ">", ">>" or "<>" operators, see section **??**. Programs started from the Workbench do not receive an output file handle at all, `pr_COS` remains `ZERO` unless explicitly set[3].

[2]The information in [**?**] that programs run from the Workbench receive a `NIL:` handle is incorrect.

[3]Again, while [**?**] claims that programs run from the Workbench receive a `NIL:` handle, this is not the case.

10.2.4 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) /* since V36 */
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. It sets `pr_COS` of the calling process. It is advisable to replace the output file handle to its original value before terminating a running program.

10.2.5 Retrieve the Error File Handle

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

```
file = ErrorOutput() /* since V47 */
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 corresponds therefore 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 not be closed explicitly. The standard error stream may be changed through `SelectError()`.

For command line programs started from the shell, this file handle is connected to the console. It is by the workings of `ErrorOutput()` redirected along with the output file handle as typically a standard error stream is not supplied — note that this is different from how shells under Unix type operating systems work. The standard error stream of command line executables can be explicitly redirected to a different file by means of the "`*>`", "`*<>`" or "`*>>`" operators, see section **??**. Programs started from the Workbench do not receive an error file, `pr_CES` remains `ZERO` unless explicitly set.

10.2.6 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) /* since V47 */
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. It sets `pr_CES` of the calling process. It is advisable to replace the error file handle to its original value before terminating a running program.

10.2.7 Retrieve the Current Directory

The `GetCurrentDir()` function returns the current directory of the calling process, represented by a lock. This lock, and the file system that manages the lock are used to resolve relative paths, see also chapter **??**.

```
lock = GetCurrentDir(void) /* since V47 */
D0

BPTR GetCurrentDir()
```

This function returns the lock to the current directory, unlike the `CurrentDir()` function which also changes it. It returns the `pr_CurrentDir` element of the calling process. The original current directory supplied to a program should not be released in general, see also the warnings in section **??**.

10.2.8 Replace the Current Directory

The `CurrentDir()` function updates the current directory of the calling process and returns the previously installed current directory. The directory is represented by a lock. This lock, and the file system that created the lock are used to resolve relative paths, see also chapter **??**.

```
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. It updates the `pr_CurrentDir` element of the `Process` structure. 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. For programs started from the shell, the original lock (and not just a lock to the same directory) shall be restored back before the program terminates as otherwise the shell would be confused, i.e. the current directory shown by the prompt would then be no longer corresponding to the current directory the shell operates in. The only exception is the `CD` command and related commands such as `PushCD`, `PopCD` and `SwapCD` which modify the current directory on purpose, but also update the prompt, see sections **??** and following for further details.

If the current directory is `ZERO`, paths are relative to the root directory of the file system accessible through the `pr_FileSystemTask` element of the calling process. AmigaDOS installs there the file system of the boot volume, unless changed by a user.

10.2.9 Return the Error Code of the Previous Operation, List of Error Codes

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

```
error = IoErr()
D0

LONG IoErr(void)
```

This function returns the secondary result code from the last call to the *dos.library*. Unfortunately, not all functions set `IoErr()`, and some only set it in particular cases and leave it untouched otherwise. All unbuffered operations in section **??** provide an error code in case of failure, though still modify `IoErr()` in case of success to an undefined value. Reading and writing bytes, and the `Seek()` function can be expected to set `IoErr()` to 0 on success.

The buffered functions in section **??** generally only set a secondary result code in case an (unbuffered) I/O operation is required, but do not touch `IoErr()` if the call can be satisfied from the buffer of the file handle. Whether a function of the *dos.library* touches `IoErr()` is stated in the description of the corresponding function.

Some rare functions of the *dos.library* also provide a secondary result code differing from an error code in case of success. Such unconventional use of `IoErr()` is explicitly mentioned in the description of the corresponding function. A particular example is `DeviceProc()`, which returns the (first) lock of a regular assign through `IoErr()`.

Most error codes are defined in `dos/dos.h`, with some additional error codes only used by the pattern matcher (see chapter **??**) in `dos/dosasl.h`. While those include files define the error codes within AmigaDOS, some file systems forward errors generated by an underlying device driver directly to the caller, and thus, for example, the errors from `devices/trackdisk.h` can also appear as `IoErr()` return codes. This design is probably not ideal as a caller has in general no idea which underlying device is used by a handler or file system, and thus lacks the proper interpretation of such codes. Instead, handlers and file systems should rather attempt to restrict the delivered error codes to those defined in the above two files and potentially translate errors from a lower level to those defined within AmigaDOS, i.e. should only report codes from the list below.

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_QPKT_FAILURE`: This non-documented error code of value `101` is generated if an attempt is made to send a DosPacket whose `dp_Link` element is `NULL` and thus is not carried by a message. See also section **??** for the definition of the `DosPacket` structure and its usage.

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

`ERROR_TASK_TABLE_FULL`: This error code was generated by AmigaDOS versions 34 and before if more than 20 shell processes were about to be created. It used a fixed-size table to store CLI processes that could not hold more than 20 processes. See also section **??** and following for more details. As this limitation was removed, the error code is unused in AmigaDOS version 36 and above.

`ERROR_GLOBALS`: This undocumented error code of the value `111` is generated when the initialization of the Global Vector fails because an entry beyond the vector bounds is supposed to be populated. This error is generated by the runtime binder described in section **??**.

`ERROR_BAD_TEMPLATE`: This error code indicates that the argument template for `ReadArgs()` is syntactically incorrect, see section **??**. It is also set by the pattern matcher in case the pattern is syntactically incorrect, see **??**.

`ERROR_BAD_NUMBER`: This error code indicates that a string could not be converted to a number. It is for example set by `ReadArgs()` if a command line argument of type `/N` is requested, but the user failed to provide a valid number. It is also set by `ExAll()` if its `type` argument is beyond the supported range, see section **??**. Surprisingly, the `StrToLong()` function of section **??** does not touch `IoErr()` and thus does not generate this error.

`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 on the command line than requested by the template.

`ERROR_UNMATCHED_QUOTES`: This error code indicates that a closing quote is missing for at least one opening quote. It is created by the argument parser and `ReadItem()`, see section **??**.

ERROR_LINE_TOO_LONG: This error code is a general indicator that a user provided buffer is too small to contain a string. It is for example used by the argument parser, the path manipulation functions in section **??** and the soft link resolver ReadLink() of section **??**. Surprisingly, the *dos.library* internal function that uses ReadLink() to resolve soft links for functions such as Lock() or Open() raises the error code ERROR_INVALID_COMPONENT_NAME if its internal buffer is too small to hold the target of a link.

ERROR_FILE_NOT_OBJECT: This error code is generated by the shell if an attempt is made to load a file that is neither a script, nor an executable nor a file that can be opened by a viewer. Surprisingly, the Workbench uses the code ERROR_NOT_EXECUTABLE if an attempt is made to load a tool whose FIBB_EXECUTE bit (see section **??**) is set to indicate a non-executable file.

ERROR_INVALID_RESIDENT_LIBRARY: AmigaDOS versions up to release 34 generated this error code if an attempt was made to open a non-existent or invalid system library as a segment of a binary file (see chapter **??**). The structure of such files allowed runtime linking to libraries, further information on this obsolete mechanism is found in [**?**]. This mechanism was phased out in AmigaDOS version 36 as it is incompatible with the AmigaOs library calling conventions expecting the library base in register A6. Today, several handlers and other AmigaOs components use this error code to indicate that a required library or device is not available.

ERROR_NO_DEFAULT_DIR: This is error code is not in use by the current version(s) of AmigaDOS. Its intended purpose is unclear.

ERROR_OBJECT_IN_USE: This error code is used by multiple AmigaDOS components to indicate that a particular operation cannot be performed because the object to be modified is in use. It is for example raised to indicate that an attempt was made to lock an object that is exclusively locked by another process, or an request for an exclusive lock on an already locked object was made.

ERROR_OBJECT_EXISTS: This error code is a generic error indicator that an operation could not be performed because another object already exists in a place where an object is to be created. 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 — or rather the CD, PopCD, PushCD and SwapCD commands of the AmigaDOS Shell — use this error code on an attempt to change the working directory to a non-existing 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. It is also generated on an attempt to resolve a soft link whose target has been renamed, moved or deleted.

ERROR_BAD_STREAM_NAME: This error code is currently not in use by AmigaDOS ROM components. AmigaDOS versions 45 and before generated it if an attempt was made to provide a non-interactive input file to the NewShell command. The CON-Handler version 34 and below also used this error code if it could not parse the path.

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 DosPacket, see section **??**) 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 once in a path as separator between the device name and the path components, a

colon within a component is therefore a syntactical error; also, the forwards slash ("/") shall not appear in a volume name. All Amiga ROM file systems do not accept code points below `0x20` in file names, i.e. ASCII control characters. It is also generated if the temporary buffer used by the *dos.library* to resolve a soft link is too small to keep the link target.

`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 can result in such an error. 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 file or an attempt create a hard link to the root directory of a FFS volume 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, and the consistency check either failed or is still ongoing. This error is for example generated if a write operation is attempted on an FFS volume whose validation has not yet completed. In such a case, retrying the operation later may solve the problem.

`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. It is also generated if the medium is write protected by software, e.g. through the `Lock` command, see section **??**.

`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. As two different file systems with potentially different architectures are involved, this cannot succeed. The Workbench copies objects (and their sub-objects) in such a case instead.

`ERROR_DIRECTORY_NOT_EMPTY`: Indicates that an attempt was made to delete a directory that is not empty. First, all objects 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 soft links refer iteratively to other soft links. In order to avoid an endless indirection of links referring to each other, the *dos.library* aborts following links after 15 passes; application programs attempting to resolve soft links themselves through `ReadLink()` should implement a similar mechanism, see also section **??**. This error is also generated by the `MatchNext()` function of the directory scanner specified in section **??** if directories are nested too deeply and the function runs out of stack space.

`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 available in any mounted drive. If, however, a known drive is accessed through an absolute path name but no medium is inserted, then the error `ERROR_NO_DISK` is generated instead.

`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 signaled if the mode of `Seek()` or `SetFileSize()` is none of the modes indicated in table **??** of section **??**. The FFS also sets this error if it cannot read one of its administration blocks and thus is unable to find the data blocks of a file.

`ERROR_COMMENT_TOO_BIG`: This error is raised if the size of the comment is too large to be stored in 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. The FFS allows comments of at most 79 characters.

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

`ERROR_DELETE_PROTECTED`: This error is generated by file systems if an attempt is made to delete an object that is delete protected, i.e. whose `FIBB_DELETE` protection bit is set. This bit is also defined in table **??** of section **??**.

ERROR_WRITE_PROTECTED: This error is raised by file systems if a write is attempted to a file that is write protected, i.e. whose FIBB_WRITE bit is set. Physical or software write protection through Lock is, however, indicated by the error code ERROR_DISK_WRITE_PROTECTED.

ERROR_READ_PROTECTED: This error is generated on an attempt to read from a file that is protected from reading, i.e. by having its FIBB_READ bit is set.

ERROR_NOT_A_DOS_DISK: This error is raised by a file system on an attempt to access a disk that is not structured according to the requirements of the file system, e.g. because it is initialized by another incompatible file system different from the mounted one. Unfortunately, AmigaDOS does not have a control instance that selects file systems according to the disk structure.

ERROR_NO_DISK: This error indicates that an attempt was made to access an object on an empty physical drive through an absolute path based on its device name. If the access to a file system object on an empty drive is made through a lock or a file handle, then the error code ERROR_DEVICE_NOT_MOUNTED is reported instead.

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 by ReadLink() within the library, not requiring intervention of the caller. However, not all functions of the *dos.library* are aware of soft links, see table **??** of section **??** for the list.

ERROR_OBJECT_LINKED: This error code is currently not used by AmigaDOS and its purpose 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 for 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. This error could also be generated by file systems that are able to interpret a packet, but do not implement it. The borderline between ERROR_ACTION_NOT_KNOWN and this error is not well defined, and handlers should report the former and not this error if they receive a packet they do not understand.

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. See section **??** for more information on record locking.

ERROR_LOCK_COLLISION: This error is also created by the record-locking subsystem of file systems if an attempt is made to lock the same region within a file by two conflicting locks, that is, request an exclusive lock on a region that is overlapping with an already locked region, or request a shared lock on a region overlapping with an exclusively locked region. See section **??** for more details.

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

ERROR_UNLOCK_ERROR: This error is currently not generated by any file system, though could be used to indicate that an attempt to unlock a record failed for an unknown reason. It was used by earlier versions of the RAM-Handler.

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 chapter **??** and the functions in section **??**.

ERROR_BREAK: This error is also raised by the pattern matcher if it received an external signal for aborting a directory scan, see section **??**. Such signals are raised, for example, by pressing Ctrl-C to Ctrl-F on the console.

ERROR_NOT_EXECUTABLE: This error is generated by the Workbench on an attempt to start an application icon from a file whose FIBB_EXECUTE protection bit is set, indicating that the file is not executable. Why the Workbench does not use the same error code as the shell, namely ERROR_FILE_NOT_OBJECT is unclear.

10.2.10 Setting IoErr

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

```
oldcode = SetIoErr(code) /* since V36 */
D0                 D1

LONG SetIoErr(LONG);
```

This function sets the next value returned by IoErr() by updating the pr_Result2 element of the Process structure of the calling process; 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 practice 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. See section **??** for the specified error codes. This function may also be used by command line tools to set the error code the Why command will report on.

10.2.11 Select the Console Handler

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

```
oldport = SetConsoleTask(port) /* since V36 */
D0                       D1

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

This function selects the MsgPort of the console handler. AmigaDOS contacts this handler for opening the "*" as file name, or a path 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. The function returns the previously used console handler in oldport. It sets the pr_ConsoleTask element of the calling process.

This function is the setter function corresponding to the GetConsoleTask() getter function introduced in section **??**.

A particular use case for this function to clone a file handle, i.e. to create a duplicate of an already open handle, see the code in section **??**.

In command line executables, any modification to the console task shall be reverted before the program returns to the shell.

10.2.12 Select the Default File System

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

```
oldport = SetFileSysTask(port) /* since V36 */
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 replaced 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 in oldport. This function is the setter equivalent of GetFileSysTask() introduced in section **??**. It sets the pr_FileSystemTask element of the Process structure of the calling process.

10.2.13 Retrieve the Lock to the Program Directory

The GetProgramDir() returns a lock to the directory that contains the binary the calling process executes, if such a directory exists. If the executable was taken from the list of resident segments (see section **??**), this function returns ZERO.

```
lock = GetProgramDir() /* since V36 */
D0

BPTR GetProgramDir(void)
```

The lock returned by this function corresponds to the PROGDIR (pseudo)-assign and the pr_HomeDir element of the Process structure, with the only exception that ZERO does not correspond to the root directory of the boot volume, but rather indicates that no home directory exists. This lock may be used to access data that are stored along with the executing binary. It is allocated — and will be released — by the AmigaDOS component that loaded the executable, e.g. the Workbench or the shell; it shall therefore *not* be unlocked by the caller.

10.2.14 Set the Program Directory

The SetProgramDir sets the directory within which the executing program is made to believe of getting started from, and the directory that corresponds to the PROGDIR pseudo-assign.

```
oldlock = SetProgramDir(lock) /* since V36 */
D0                        D1

BPTR SetProgramDir(BPTR)
```

This function installs lock into pr_HomeDir of the Process structure and returns the lock that was previously installed there. The passed lock indicates the directory the currently executing program was loaded from and is used to resolve the PROGDIR pseudo-assign. If ZERO is installed, the current process will be unable to resolve this pseudo-assign.

Application programs should rarely have a reason to call this function. However, if the home directory of the caller is modified, the original lock shall be restored back before the program terminates.

10.2.15 Retrieve Command Line Arguments

The GetArgStr() function returns the command line arguments, if any, of the calling process. If the calling process was launched from from the Workbench, this function returns NULL.

```
ptr = GetArgStr() /* since V36 */
D0

STRPTR GetArgStr(void)
```

This function returns the command line arguments of the calling process as NUL-terminated string. The same string is found in register a0 on startup, and is also located in the buffer of the Input() file handle. While this string contains all arguments provided on the command line, it does *not* include the program name itself.

This function returns NULL if the program was run from the Workbench. This function is the accessor function of the pr_Arguments element in the Process structure.

10.2.16 Set the Command Line Arguments

The SetArgStr() function sets the string returned by GetArgStr(). It cannot set the command line arguments as seen by ReadArgs().

```
oldptr = SetArgStr(ptr) /* since V36 */
D0                 D1

STRPTR SetArgStr(STRPTR)
```

This function requires a pointer to a NUL terminated string as ptr and installs it in the pr_Arguments element of the Process structure of the calling process. This function has limited use as ReadArgs() takes the command line arguments from a different source, namely the input buffer of the Input() file handle, and it rarely makes sense to adjust the command line arguments in first place as the functions RunCommand() and CreateNewProc() can set them in a more consistent way before a loaded program starts up.

Chapter 11

Binary File Structure

The AmigaDOS *Hunk* format is the structure of executable programs, linkable object files and link libraries on disk. The first type includes application programs that are loaded from the Workbench or the shell. Their structure is specified in section **??**. An object file is an intermediate file that is generated by a compiler or assembler from a translation unit, i.e. typically one source file and the headers it includes. It still contains references to symbols defined in other translation units or in link libraries that are resolved when linking them to an executable. Object files are introduced in section **??**. Link libraries are collections of small utility functions; those referenced in the object files are pulled into the executable by the linker. Unlike AmigaDOS libraries stored in the `LIBS` assign, they are not loaded at runtime and not shared between applications. Section **??** specifies their format.

A file in either format, i.e. regardless whether it is an executable, an object file or a link library, consists of multiple *hunks*, each of which defines either payload data as an indivisible segment of code or data that is zero-initialized or loaded from disk, or additional meta-information. This format is interpreted by the AmigaDOS loader — i.e. the `LoadSeg()` function — or by a linker. Meta information within executables is used to relocate the payload to their final position in memory, to define the size of the segments, to select the memory type that is allocated for the segment, or to control the loading process. Meta information only present in object files or link libraries is required for resolving references across files when linking them together to form a final executables.

Once an executable file has been loaded to memory by the `LoadSeg()` function — see section **??** for details on this process — it is represented there as a singly linked list of *segments*. Each node in this list looks as follows:

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

The `LoadSeg()` function returns a BPTR to the first node of this list. While the entire list is called the *segment list* in the following, each of its nodes will be called a *segment*. The long word immediately preceding the `NextSegment` element contains the byte size of the allocated memory block including overhead, i.e. compatible to `AllocVec()` and `FreeVec()` — which is sometimes helpful to derive the size of the segment.

The above structure is *not* documented and is not identical to the `Segment` in `dos/dosextens.h`, neither to the `pr_SegList` stored in the `Process` structure. The former describes a resident executable, see section **??**, but also contains a BPTR to a segment list in the above sense. The `pr_SegList` element is an array where each entry except the first consists of a BPTR to a segment list in the above sense. The segment array is defined in chapter **??**.

The Hunk format represents each segment by a hunk, but distinguishes three different types: *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, but only their size is.

> *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 the above 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 each hunk is corrected according to the addresses the hunks are loaded to. The relocation process takes an offset into one hunk, and adds to the long word at this offset the absolute address of another hunk, see section **??** and following. That is, hunks on disk are represented as if their first byte is placed at address 0, and relocation adjusts the pointers within hunks such that they point to their target positions into other loaded hunks.

An extension of the executable file format is the *overlay format* also supported by the AmigaDOS loader, i.e. `LoadSeg()`. Here initially only a part of the file is brought into memory, while the remaining parts are only loaded on demand, potentially releasing other already loaded parts from memory. Overlaid executables thus take less memory, though require that the volume containing the executable remains available while the loaded program is running. The format is specified in more detail in section **??**.

AmigaDOS also contains a simple run-time binder that is only used by executables written in BCPL, or by code that operates under BCPL conventions. 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 depend on it. These are handlers or file systems that use the `dol_GlobVec` value of 0 or -3, or the corresponding `GLOBVEC` entry in the mountlist. While new handlers should not use this BCPL legacy protocol, the ROM file system (the FFS) and the Port-Handler currently still depend on it, despite not being written in BCPL. This mechanism is described in section **??**.

11.1 Conventions and Pseudo-Code

In the following, the syntax of files and its hunks is represented as pseudo-code in three-column tables that reflect approximately the inner workings of the AmigaDOS loader, or the implementation of an Amiga linker. In such 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 larger objects are read from most significant to least significant byte. 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 variable or element of a structure to which the value removed from the stream is assigned to; if it is empty, the read data is ignored. If it contains a constant, then the read value shall be equal to this constant, otherwise the file is ill-formed. An expression such as $t \in [a, b]$ first removes the number of bits as indicated in the left-hand column from the stream, assigns it to the variable t, and then checks whether the resulting value is greater or equal than a and smaller or equal than b. If this is not the case, the file is ill-formed. Unfortunately, the AmigaDOS loader does not always perform these checks and can crash the system if the formulated constrains are violated. The second column can also contain pseudo-code that describes the flow of operations during interpretation of files. These elements follow closely the conventions of the C language [**?**]. In their absence, the contents of the tables shall be interpreted top to bottom, where the topmost elements are removed from the file first, or executed first.

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. The statements executed in each case are enclosed in curly brackets, i.e. `{` and `}`.

The `do ... while(`*cond*`)` control structure indicates a loop that continues as long as `cond` is true, and that may alternatively be terminated by a `break` within the body of the loop. The body of the loop is again enclosed in curly brackets. The syntax element `for(init;cond;incr)` also represents a loop where `init` are statements executed initially, `cond` are conditions that are checked before each loop is entered, and `incr` are statements that are executed at the end of each iteration. If `cond` is false, the loop is aborted; as this test is also made before the first loop, the body of the loop is possibly not executed at all. Similar to the first type of loop, `break` aborts the loop, and the loop body is enclosed in curly brackets.

Statements starting with `parse_` continue the parsing process in other sections that are referenced in the rightmost column and return to the execution path when done there. Thus, they are the analog of function calls in the C programming language. A statement such as `ERROR_BAD_HUNK` aborts the loading process completely and signals the indicated error code. The symbols starting with `HUNK_` are defined in the include file `dos/doshunks.h`, but their values are conveniently also presented at the top of each table in square brackets.

The conditional operator == checks for equality, and `!=` checks for inequality. Similarly $<$, $>$, $\leq$ and $\geq$ check whether the left hand expression is smaller, greater, smaller or equal or greater or equal than the right hand side, and return either true or false. The condition `EOF` is true if the end of the file has been reached, it is false otherwise. Unlike its C counterpart, the condition becomes true *at* the end of the file, and not when an attempt is made to read beyond its end. The operator `!` inverts the result of the condition behind it.

Arithmetic operators follow the conventions of the C language, $+$ is addition, $-$ is subtraction and $\times$ indicates multiplication of the left hand side with the right hand side, where multiplication binds stronger than addition or subtraction, i.e. is executed first. The $\&$ operator is the binary "and", i.e. $p\&1$ is true if p is odd. 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 expressions `+=` and `-=` add or subtract the value on its right to the variable on its left.

11.2 Executable File Format

The Hunk format of executable files consists of 4-byte (long word) hunk identifiers and subsequent data that is interpreted by the AmigaDOS loader according to the introducing hunk identifier. The following pseudo-code describes the top-level syntax of a binary executable file AmigaDOS is able to bring to memory by means of `LoadSeg()`. The table defines the file format using the pseudo-code introduced in section **??**:

Table 11.1: Regular Executable File

Size	Code	Syntax
?	`parse_HEADER`	Defines all segments, see section **??** 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 **??**
1	a_f	Advisory hunk flag.
29	h	This is the hunk type

	`if (`a_f`) {`	Check for bit 29, these are advisory hunks
32	l	Read length of the advisory hunk
$32 \times l$		l long words of hunk contents ignored
	`}`	
	`else if (`h `== HUNK_END)` i`++`	Advance to next segment, see **??**
	`else if (`h `== HUNK_BREAK)` `break`	Terminate an overlay, see **??**
?	`else if (`h `== HUNK_NAME)` `parse_NAME`	See section **??**
?	`else if (`h `== HUNK_CODE)` `parse_CODE`	See section **??**
?	`else if (`h `== HUNK_DATA)` `parse_DATA`	See section **??**
?	`else if (`h `== HUNK_BSS)` `parse_BSS`	See section **??**
?	`else if (`h `== HUNK_RELOC32)` `parse_RELOC32`	See section **??**
?	`else if (`h `== HUNK_SYMBOL)` `parse_SYMBOL`	See section **??**
?	`else if (`h `== HUNK_DEBUG)` `parse_DEBUG`	See section **??**
?	`else if (`h `== HUNK_OVERLAY)` `{ parse_OVERLAY; break }`	See section **??**
?	`else if (`h `== HUNK_DREL32)` `parse_RELOC32SHORT`	Compatibility kludge Use `HUNK_RELOC32SHORT` instead See section **??**
?	`else if (`h `== HUNK_RELOC32SHORT)` `parse_RELOC32SHORT`	See section **??**
?	`else if (`h `== HUNK_RELRELOC32)` `parse_RELRELOC32`	See section **??**
	`else ERROR_BAD_HUNK`	Everything else is invalid
	`} while(!EOF)`	repeat until end of file

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 body of the `HUNK_HEADER`. They have there, however, no effect as the segments are allocated within the `HUNK_HEADER` and not at the time 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, and the contents of this hunk is skipped over. Currently, there is no publicly known use of such hunks, and AmigaDOS versions 34 and below do not support them.

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 *overlaid* files. In this case, 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 **??**. Finally, an error condition or an attempt to read past the end of file also aborts loading.

11.2.1 HUNK_HEADER

The HUNK_HEADER shall be 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 11.2: 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	$t_{\texttt{size}} \in [1, 2^{31}-1]$	Number of segments in binary
32	$t_{\texttt{num}} \in [0, t_{\texttt{size}}-1]$	First segment to load
32	$t_{\texttt{max}} \in [t_{\texttt{num}}, t_{\texttt{size}}-1]$	Last segment to load (inclusive)
	`for(`j=$t_{\texttt{num}}$`;`$j \le t_{\texttt{max}}$`;`j`++) {`	Iterate over all hunks
2	$m_t[j]$	Read memory type of the segment
30	$m_s[j]$	Read memory size in long words
	`if (`$m_t[j]$ `== 3) {`	if the memory type is 3
32	μ_t	Memory type is explicitly provided
	`} else {`	
	$\mu_t = m_t[j] \ll 1$	Regular memory requirement
	`}`	
	$m_a[j]$ `= AllocVec(` `sizeof(BPTR)+`$m_s[j]$`×sizeof(LONG),` μ_t `| 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, it was not particularly useful as the calling conventions for such libraries did not follow the usual conventions of AmigaDOS, i.e. requiring the library base in register a6. Later versions of AmigaDOS, in particular its re-implementation as of Kickstart v36, removed support for such libraries. As this mechanism is no longer supported, it is not documented here. More information is found in [**?**].

The second entry t_{size} contains the number of segments the executable consists of. In case of overlays, it is the maximal number of segments that can be resident in memory at once. See section **??** for more information. This value shall be consistent for all HUNK_HEADERs within an overlaid 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-overlaid) file and for the root node of the overlay tree, t_{num} shall be 0, that is, the first segment to load is 0.

For regular files, t_{max} shall be identical to $t_{\text{size}} - 1$, that is, the last segment to load is the last entry in the segment table described by this HUNK_HEADER. For overlaid 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.

The remaining data in the hunk define the sizes of the segments along with their memory type. The memory type is indicated by two bits that map directly to MEMF_ANY, MEMF_CHIP and MEMF_FAST for the bit combinations 00, 01 and 10 respectively, see also exec/memory.h. If the two memory type bits are both 1, an additional long word contains the memory type explicitly. This combination was introduced in AmigaDOS version 33.

11.2.2 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 segment of an executable should be represented by a `HUNK_CODE`, and it should start with a valid opcode.

Compilers use typically this hunk to represent `text` segments, i.e. compiled code and constant data, such as strings.

The structure of this hunk is as follows:

Table 11.3: Hunk Code Syntax

Size	Code	Syntax
	`HUNK_CODE [0x3e9]`	A hunk describing a segment of code and constant data
32	$l \in [0, m_s[i]]$	Size of the payload
$l \times 32$	`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 36 skipped this initialization. Due to a bug in the loader in later versions, the initialization is also skipped if the hunk length l is 0.

11.2.3 HUNK_DATA

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

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

Table 11.4: Hunk Data Syntax

Size	Code	Syntax
	`HUNK_CODE [0x3ea]`	A hunk describing a segment of data
32	$l \in [0, m_s[i]]$	Size of the payload
$l \times 32$	`Data`	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 36 onward. Due to a bug in the loader in later versions, the initialization is also skipped if the hunk length l is 0.

11.2.4 HUNK_BSS

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

The structure of this hunk is as follows:

Table 11.5: Hunk BSS Syntax

Size	Code	Syntax
	`HUNK_CODE [0x3eb]`	A hunk describing zero-initialized data
32	$l \in [0, m_s[i]]$	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 36, 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 $l < m_s[i]$.

11.2.5 HUNK_RELOC32

This hunk contains relocation information for the currently loaded segment; that is, it corrects addresses within segment i by adding the absolute address of this or other segments to long words at indicated offsets of the previous segments. As it needs to fix up offsets within an already loaded segment, it shall appear *behind* the hunk carrying the payload data it is supposed to relocate, i.e. behind `HUNK_CODE` or `HUNK_DATA`. While it is also possible to relocate data within BSS hunks, this is at least unusual.

The structure of this hunk is as follows:

Table 11.6: 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 offset
	`(UBYTE **)(`$m_a[i] + r_o$`) +=` $m_a[j]$	Fix-up 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. As this hunk relocates pointers into other hunks, the relocation offsets r_o shall all be even; the AmigaDOS loader does not check this condition, and would crash on an odd offset if executed on an 68000 or 68010 processor.

For AmigaDOS versions 36 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 been fixed. If more relocation entries are needed, they need to be split into multiple hunks.

11.2.6 HUNK_RELOC32SHORT

This hunk contains relocation information for the currently loaded segment, 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. This hunk type was introduced[1] in AmigaDOS version 36. Similar to all hunks carrying relocation information, this hunk shall appear after the hunk carrying the payload data it relocates.

The structure of this hunk is as follows:

Table 11.7: Hunk Reloc32Short Syntax

Size	Code	Syntax
	`HUNK_RELOC32SHORT [0x3fc]`	A hunk containing relocation information
	p=1	Padding count

[1] But see also the note at the end of the table for defects.

	`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 offset
	`(UBYTE **)(`$m_a[i] + r_o$`) +=` $m_a[j]$	Fix-up 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
	`}`	

Similar to its long counterpart, the relocation offsets r_o shall all be even as otherwise the AmigaDOS loader crashes on the smaller processors.

Due to an oversight, versions 36 to 38 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.

11.2.7 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. This hunk was introduced in AmigaDOS 39. Similar to all hunks carrying relocation information, this hunk shall appear after the hunk carrying the payload data it relocates.

The structure of this hunk is as follows:

Table 11.8: 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 **)(`$m_a[i] + r_o$`) +=` $m_a[j] - m_a[i] - r_o$	Fix-up 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.

Even if this hunk only makes sense for processes from the 68020 onward, odd relocation offsets make little sense as the instructions it relocates are still word-aligned.

For AmigaDOS versions 36 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 in AmigaDOS 39 onward, all elements of this hunk, namely c, j and r_o are only read as 16 bit wide elements, even though they should be (as documented) 32 bits wide, 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. Luckily, the support and demand for this hunk type is very limited.

11.2.8 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. This hunk shall always appear upfront the hunk carrying the payload data, i.e. before `HUNK_CODE`, `HUNK_DATA` or `HUNK_BSS`. This hunk *is used and relevant* in object files: Linkers that bind object files together use the name to decide which segments to merge to a single segment, more details on this in section **??**.

The structure of this hunk is as follows:

Table 11.9: 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 zero-padded to the next 32-bit boundary if necessary 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.

11.2.9 HUNK_SYMBOL

This hunk defines symbol names and corresponding symbol offsets 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. This hunk should appear behind the payload data it annotates.

The syntax of this hunk reads as follows:

Table 11.10: Hunk Symbol Syntax

Size	Code	Syntax
	`HUNK_SYMBOL [0x3f0]`	A hunk assigning symbols to offsets within a segment
	`do {`	Repeat . . .
8	s_t	Symbol type
24	s_l	Symbol name 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. $s_l = 31$.

The symbol type s_t defines the nature of the symbol. The symbol types are defined in the include file `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; it is defined in section **??**.

Only a single type is permitted here, namely `EXT_SYMB`, which corresponds to the value $s_t = 0$. The address of the symbol is given by $s_v + m_a[i]$, i.e. the symbol value is the offset into the segment represented by the hunk to which the `HUNK_SYMBOL` hunk belongs. All other types only appear in `HUNK_EXT`.

11.2.10 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. Similar to `HUNK_SYMBOL`, this hunk should appear behind the payload data it annotates.

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 segment containing the code that was compiled from this line.

The syntax of this hunk is as follows:

Table 11.11: Hunk Debug Syntax

Size	Code	Syntax
	`HUNK_DEBUG [0x3f1]`	Hunk including debug information
32	$l \in [3, 2^{31} - 1]$	Size of the hunk in long words
32	h_o	Offset of symbols into the segment
Compiler- and configuration specific data for line-debug data:		
32	`'LINE'`	Identifies the type of 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 $m_a[i] + l_v + h_o$ 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.

The hunk offset h_o is added to all offsets l_v to determine the position of a symbol in the segment. It is

generated by the linker when merging multiple segments into one.

While [**?**] and [**?**] document the entire hunk contents except the hunk length l to be compiler dependent, it is is recommended for custom debug hunks to always include the hunk offset h_o and the ID field — `'LINE'` in this case — to simplify linker designs.

11.2.11 HUNK_END

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

Table 11.12: Hunk End Syntax

Size	Code	Syntax
	`HUNK_END [0x3f2]`	Terminate a segment

11.3 The AmigaDOS Loader

The *dos.library* provides service functions for loading and releasing binary executables in the *Hunk* format introduced in section **??**. 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 segment lists. Overlay files are discussed separately in section **??** due to their additional complexity.

A segment list is a linearly linked list of segments as defined in the beginning of chapter **??**, 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.

Segment lists are the representation of loaded executables. The loaded code can be, for example, either launched as a new process through `CreateNewProc()` explained in chapter **??** or run as command overloading the current process via `RunCommand()` introduced in section **??**. Processes and commands using the C or assembler binding are started from offset 0 of the first segment loaded. Segments loaded as BCPL handlers as indicated by their `GLOBVEC` mountlist entry, see table **??** in chapter **??**, use the mechanism of section **??** instead.

11.3.1 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 from the path `name` and returns a BPTR to the singly linked list of segments in case of success, or `ZERO` in case of failure. The `name` is passed into the `Open()` function, accessing the file in the `MODE_OLDFILE` (shared) mode, and follows the conventions of this function for locating the file.

When done, the segment list shall be released from memory via `UnLoadSeg()`, see section **??**.

Unfortunately, this function has a series of defects: For AmigaDOS 34 and below, the function only zero-initialized the first 256K of BSS segments, and it failed to zero-initialize the regions behind the payload data of code and data segments. AmigaDOS 36 and up fixed the initialization problem, but the implicit BSS space at the end of code and data segments is only initialized if they *do* contain payload. Due to another bug in AmigaDOS 36 and up, at most 2^{16} relocation entries are supported. While this version also introduced a new

and more efficient relocation hunk with 16 bit offsets for short executables, see section **??**, the wrong hunk identifier was picked for it. AmigaDOS 39 fixed this problem (but none of the others), and also introduced another hunk type for 32-bit relative relocation as specified in section **??**. Unfortunately, its implementation is defect and limited to 16 bit relocation offsets.

To work around these issues, one should first clear the (implicit) BSS regions of code and data hunks manually, or avoid them. BSS hunks larger than 256K should also be avoided, memory of this size should probably be allocated through the *exec.library* anyhow. While more than 2^{16} relocation entries are rare, authors of linker tools should break such relocation lists up into multiple arrays, all going to the same hunk. While `HUNK_RELOC32SHORT` is more efficient, it shall not be used for executables that are supposed to execute under AmigaDOS 34 and below, and for better compatibility, the (albeit incorrect) hunk type `HUNK_DRELOC32 [0x3f7]` should be preferred as such binaries also load under AmigaDOS 36, including the fixed version 39. The new 32-bit relative relocation by `HUNK_RELRELOC32` should be avoided altogether.

The `LoadSeg()` function sets `IoErr()` to an error code in case of failure, or to an undefined value in case of success.

11.3.2 Loading an Executable with Additional Parameters

The `NewLoadSeg()` function loads an executable providing additional data for loading, and receiving additional data from the binary if available.

```
seglist = NewLoadSeg(file, tags) /* since V36 */
D0                   D1    D2

BPTR NewLoadSeg(STRPTR, struct TagItem *)

seglist = NewLoadSegTagList(file, tags) /* since V36 */
D0                          D1    D2

BPTR NewLoadSegTagList(STRPTR, struct TagItem *)

seglist = NewLoadSegTags(file, ...) /* since V36 */

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

This function loads a binary executable from `file` and returns a BPTR to the first segment of the singly linked list of segments, 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 name; 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 tag list, i.e. the last argument shall be `TAG_DONE`. This tag list 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 exceeding that of `LoadSeg()` is made available. Version and stack information is instead extracted from by the loaded segments by the mechanisms specified in sections **??** and **??**.

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

As this function shares the implementation with `LoadSeg()`, it unfortunately also shares its defects, see section **??**. This function returns `ZERO` on error, and then sets `IoErr()` to an error code. Otherwise, it returns the BPTR to the loaded segment list and sets `IoErr()` to an undefined value.

11.3.3 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 file access and allocating memory, this function instead calls through user-provided functions.

```
seglist = InternalLoadSeg(fh, table, funcs) /* since V36 */
D0                        D0   A0     A1

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

This function loads a binary executable in the Hunk format from an opaque handle `fh` through functions in the `funcs` argument. 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 **??** for more details.

The `LoadSegFuncs` structure contains function pointers through which this function reads data or retrieves memory. It is unfortunately not officially documented, though is defined as follows:

```
struct LoadSegFuncs {
    LONG __asm ReadFunc (register __d1 BPTR fh,
                         register __d2 APTR buffer,
                         register __d3 ULONG size,
                         register __a6 struct DosLibrary *DOSBase);
    APTR __asm AllocFunc(register __d0 ULONG size,
                         register __d1 ULONG flags,
                         register __a6 struct ExecBase *SysBase);
    void __asm FreeFunc (register __a1 APTR mem,
                         register __d0 ULONG size,
                         register __a6 struct ExecBase *SysBase);
}
```

The `ReadFunc()` function retrieves `d3` bytes from an opaque handle passed into register `d1` and places the read bytes into the buffer pointed to by register `d2`, 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*. Clearly, the `Read()` function of the *dos.library* (see section **??**) satisfies this interface,

The `AllocFunc()` function allocates `d0` bytes of memory, requiring the memory type in `d1`. The requirements are encoded as flags from `exec/memory.h` such as `MEMF_CHIP` to request 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 `AllocMem()` function is suitable for this interface.

The `FreeFunc()` function releases a block of `d0` bytes pointed to by `a1`. Register `a6` is loaded with a pointer to the *exec.library*. The `FreeMem()` function works according to this interface.

The purpose of this function is to load a segment or a binary without having access to a file or a file system, or for loading binaries into dedicated memory areas; for example, this function could load binaries from ROM-space, or from the Rigid 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. This argument is not interpreted, it is forwarded to `funcs->ReadFunc()` in register `d1`.

The `InternalLoadSeg()` function follows the conventions of the `AllocVec()` function and stores the number of allocated bytes in the first long word of every memory block by itself. In specific, the memory

allocator and memory releaser functions provided in the LoadSegFuncs structure *do not need* to keep track of the memory sizes, and the AllocMem() and FreeMem() functions of the *exec.library* satisfy the interfaces for InternalLoadSeg() function already. All necessary size and pointer adjustments are made within the *dos.library*.

This function does not set IoErr() consistently, unless the functions within LoadSegFuncs do. Callers should potentially use SetIoErr(0) upfront this function to have the possibility to identify errors. This function also shares its implementation, and thus its defects, with LoadSeg().

11.3.4 Unloading a Binary

The UnLoadSeg() function releases a linked list of segments as returned the AmigaDOS segment loaders.

```
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 overlaid segments, see section **??**, and releases additional resources acquired for them. Clearly, the segments shall be no longer in use, i.e. executed by the CPU, at the time they are unloaded. AmigaDOS does not provide a protocol to learn when a process started from a segment terminates, but also see the notes on the CreateProc() function in section **??** and the NP_FreeSeglist tag of the CreateNewProc() function in section **??**.

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

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, and IoErr() will not be touched in this case; 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 associated to overlaid files.

11.3.5 UnLoading a Binary through Call-Back Functions

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

```
success = InternalUnLoadSeg(seglist,FreeFunc) /* since V36 */
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 pointer to the *exec.library*.

The pointer in a1 should be identical to the FreeFunc() function pointer in the LoadSegFuncs structure provided to InternalLoadSeg(), or at least shall be able to release memory allocated by the AllocFunc() function pointer in the same structure. Note that the InternalLoadSeg() stores the sizes of the allocated memory blocks itself and the memory release function does not need to retrieve them.

This function is also able to release overlaid binaries, but then closes the file stored in the root node of the overlay tree (see section **??**) through the `Close()` function of the *dos.library*. It therefore can only release overlaid files that were loaded from regular file handles obtained through `Open()`[2].

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 overlaid files. It therefore neither sets `IoErr()` consistently in case of failure.

11.4 Overlays

While regular binary executables are first loaded to memory in entity and then brought to execution, overlaid binaries only keep a fraction of the executable code in memory and then load additional code parts as required, potentially releasing code fragments no longer needed; thus, more memory remains available for the program. Overlaid binaries therefore allow execution of large and complex programs under constrained memory conditions.

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. If the program calls a function that is located in a segment residing in an overlay mode, this call is routed through the *overlay manager* which determines whether the function is in memory. If not so, it locates the overlay node containing the requested function, potentially releases unused nodes, loads the required node into memory and then calls into the target function.

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

As it would be bothersome to redesign an overlay manager for each application, compiler vendors equip their development kits with suitable implementations. The Amiga linker `ALink`, the Software Distillery linker `BLink` and the SAS/C linker `SLink` include a hierarchical overlay manager; except for minor details in the calling convention, their designs and data structures are all alike and discussed in detail in section **??**.

An alternative to the hierarchical ALink overlay manager was provided by the MANX (Aztec) C compiler. Its design is roughly based on the Resource Manager of the 68K MacOs, which organizes applications in multiple resources that can be loaded and unloaded independently. Unfortunately, as the overlay manager uses self-modifying code without being aware of CPU caches, it is no longer functional from the 68040 (or even 68030) onward, but will nevertheless be described here as a historical artifact. Section **??** provides details on this flat overlay manager.

11.4.1 The Overlay File Format

A binary file making use of overlays consists of several nodes, one root node and several overlaid nodes. Nodes contain one or multiple segments, represented by hunks such as `HUNK_CODE`, `HUNK_DATA` or `HUNK_BSS`, similar to regular (non-overlaid) binary files. Additional hunk types provide data to the overlay manager and steer the loading process.

Each node, the root node and all overlaid 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 additional data for the purpose of the overlay manager, and the organization of this hunk therefore depends on the overlay manager contained in the first hunk of the root node.

Every other overlay node terminates with a `HUNK_BREAK` where `LoadSeg()` interrupts loading as well. Unlike `HUNK_OVERLAY`, this hunk does not contain any data.

[2]Loading overlays through `InternalLoadSeg()` is probably a bad idea anyhow.

The overall structure of an overlaid binary looks as follows:

Table 11.13: 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 **??**
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 previous segment
HUNK_BREAK	Terminates the overlay node, see **??**
} while(!EOF)	This pattern repeats until end of file

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 used to locate symbols in overlay nodes, load the segments, but is also required to release resources associated to loaded overlays and is therefore expected there by `UnLoadSeg()` and `InternalUnLoadSeg()`.

The first bytes of the first `HUNK_CODE` in the root node shall 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             */
};
```

The elements `oh_FileHandle` to `oh_GV` are filled in by the AmigaDOS loader, i.e. `LoadSeg()` and related functions; `oh_Jump` and `oh_Magic` shall be part of the segment itself and shall be included as first two long words in the hunk. Note that the first segment(s) contains the overlay manager, and thus are typically contributed by the linker and not by the application programmer.

`oh_Jump` forms a valid `68K` opcode, 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, and the CPU would run into the data of the structure otherwise, likely crashing on illegal opcodes. This first long word is not interpreted by the AmigaDOS loader, it just expects it to be present.

`oh_Magic` shall contain the "magic" long-word `0xabcd`. This value is neither filled or interpreted by the loader, but shall nevertheless 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[3].

`oh_FileHandle` will be filled by the AmigaDOS loader with a BPTR to the file handle from which the root node has been loaded, or with the first argument of `InternalLoadSeg()`. This handle is used

[3]From this follows that *flat* (non-overlaid) AmigaDOS binary shall not contain this magic long word at this position as otherwise attempting to unload such a binary will cause mischief.

by the overload manager to load all subsequent overlay nodes. Also, `UnLoadSeg()` and related functions call `Close()` on the handle stored here. This handle needs to stay open for the life time of the program as overlay nodes are loaded as they are required by the executing code.

`oh_OVTab` is filled by the AmigaDOS loader to a pointer to the payload data of `HUNK_OVERLAY`, i.e. the data stored in the hunk following the length indicator l in table **??**.

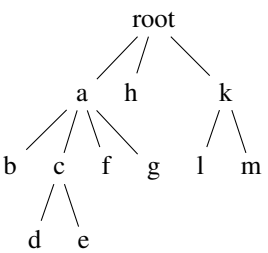
`oh_Segments` in the `OverlayHeader` structure is filled by the AmigaDOS loader with a BPTR to the segment table of the loaded binary, represented as an array of BPTRs. The size of this array is taken from t_{size} in the `HUNK_HEADER` of the root node, see table **??** in section **??**. Each element in the segment table 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, i.e. the overlay manager itself. Thus, the segments of the loaded executable form a singly linked list whose nodes are in addition directly accessibly through the table pointed to by `oh_Segments`.

When parsing a `HUNK_HEADER`, the table entries t_{num} to t_{max} will be populated by the BPTRs to the segments allocated within this hunk, and when unloading an overlay node, the corresponding segments will be unlinked, released and their corresponding entries in this table set to `ZERO` again. This table is therefore essentially the m_a array of table **??**, except that its entries are BPTRs, not regular pointers — of course another Tripos legacy. Note that the segment table contains segments and is indexed by the segment number counting from 0, whereas the entries of the `OVTab` array are ordinate numbers of loaded overlay nodes, where each node typically consists of *multiple* segments. The first entry in the latter array is the first *overlaid* node, the root node is not represented there.

`oh_GV` is, finally, filled with the public 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 base address loaded in register `a6`.

11.4.2 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 will 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 at 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 hierarchical 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.

11.4.3 HUNK_OVERLAY

This hunk terminates loading and indicates the end of the root node. The HUNK_OVERLAY contains metadata, namely the overlay table, for the overlay manager. This table contains information where symbols within the overlaid segments are located and is as such dependent on the overlay manager included in the root node.

This section describes the layout of HUNK_OVERLAY for the hierarchical overlay manager that is provided by ALink, BLink and SLink. While their implementations differs slightly, the contents of the hunk is identical. The format of their HUNK_OVERLAY is as follows:

Table 11.14: 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 hierarchical overlay manager, $l + 1$ long-words in total		
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	Will become the currently loaded ordinate once loaded, shall be zero in the file.
	}	...that is, $o_d - 1$ zeros
	$l-=o_d$	Remove read data from the count.
	for($s = 0; l \geq 0; s$++) {	Repeat over the overlay table
32	$o_p[s]$	Absolute offset of the HUNK_HEADER of the overlay node containing the symbol.
64	0	Two reserved long-words.
32	$o_l[s]$	Level of the overlay node containing the symbol, the root level containing the overlay manager is at level 0.
32	$o_n[s]$	Ordinate of the overlay node, enumerating overlay nodes of the same level.
32	$o_h[s]$	Segment index of the first segment within the overlay node.
32	$o_s[s]$	Segment index of the segment containing the symbol described by this entry in the overlay node.
32	$o_o[s]$	Symbol offset within segment $o_s[s]$.
	l -=8	Remove 8 long words.
	}	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 within a tree of two levels would be indicated by a value of $l = 9$ and would have $10 \times 4 = 40$ bytes of payload, excluding the length field. The length is always indicated in this (unorthogonal) way, regardless of the overlay manager used.

As for all overlay managers, the oh_OVTab pointer in the OverlayHeader structure introduced in **??** points to the memory representation of the overlay hunk and consists initially of the contents of table **??** starting from element the element o_d. They can also be described by the following structures which store for each tree level the ordinate of the currently loaded overlay node, and for every externally referenced symbol in an overlaid executable the position of the symbol within the overlay tree, along with the segments to be loaded:

```
struct OVTab {
  ULONG ot_TreeDepth;     /* Depth of the tree, including the root    */
  ULONG ot_Ordinate[od-1];/* 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_FirstSegment;  /* First segment of the overlay node        */
  ULONG ot_SymbolSegment; /* Segment containing the referenced symbol */
  ULONG ot_SymbolOffset;  /* Offset of the segment within the segment */
} []                      /* Repeats for each symbol */
```

That is, the overlay table of a tree depth o_d starts with an array of $o_d - 1$ elements within which each element stores the ordinate of the currently loaded overlay node at this level[4], excluding the level of the root 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 in the overlay tree.

The ordinate table is followed by the symbol table. The purpose of this structure is to allow the overlay manager to find and load an overlay node given a reference to an external symbol. How exactly it does so is explained in more detail in section **??**.

11.4.4 HUNK_BREAK

This hunk terminates the loading process and indicates the end of an overlay node. The hunk itself does not contain any data. It is not required at the end of the root node as there the `HUNK_OVERLAY` provides already sufficient information for the loader that the end of a node has been reached.

Table 11.15: Hunk Break Syntax

Size	Code	Syntax
	`HUNK_BREAK [0x3f6]`	Terminate an overlay node

11.4.5 Loading an Overlaid Node

The `LoadSeg()` function is not only able to load the root node of an overlaid binary, it can also be used for loading an overlay node and all segments within it. For that, the file pointer shall first be placed by `Seek()` to the file offset of the `HUNK_HEADER` of the node to load. For the hierarchical overlay manager, this file offset should 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 overlaid 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 should be taken from `oh_Segments` in the first segment of the root node. It contains BPTRs to all allocated segments, see section **??**.

`fh` is a BPTR to the file handle from which the overlay node is to be loaded. This handle should be taken from `oh_FileHandle`, also stored in the first segment of the root node.

[4]Such variably sized arrays cannot be expressed properly in the C syntax.

While this function allocates and loads the segments in the overlaid 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 release entries in the segment table upfront, see **??**. The overlay manager therefore needs to know which segments the node to load will populate.

For the hierarchical overlay manager, this information is available from the the `ot_FirstSegment` element of the overlay table. Due to the tree structure imposed by the hierarchical overlay manager, it has to release all segments from `ot_FirstHunk` onward up to the end of the table, unlink the segments contained therein, and then proceed loading the new overlay node through `LoadSeg()`.

Note that this function populates the same entry in the *dos.library* as the regular `LoadSeg()` function; the function distinguishes loading regular binaries through a file name from loading overlay nodes by its first argument. The function prototype available in the official AmigaDOS include files do not expose the additional arguments.

> *Seek to the Segment!* The file pointer of the file handle `fh` from which the executable is loaded is *undefined* after returning from `LoadSeg()`. You therefore shall `Seek()` to the start of the overlaid node before calling `LoadSeg()`, i.e. the byte offset of a `HUNK_HEADER` defining the node to be loaded. How to obtain the file position of an overlaid node depends on the overlay manager in use, but they are typically included in the overlay hunk, see section **??** for the hierarchical overlay manager, and section **??** for the MANX overlay manager.

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

11.4.6 Loading an Overlay Node through Call-Back Functions

While the `InternalLoadSeg()` function can also load an overlay node, its usage for overlay managers is discouraged. Generally, overlays should load code through standard file handles and not through an abstract object as `UnLoadSeg()` will call `Close()` on the stream used for overlay loading, regardless whether it is a file handle or not.

```
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 overlaid node, e.g. by a functionality similar to `Seek()` for regular file handles. This file offset should, for the standard hierarchical overlay manager, be taken from `ot_FilePosition`.

The `table` shall be the BPTR to the segment table; this should be taken from `oh_Segments` in the root node, see section **??**. 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 such segments. For the hierarchical overlay manager, 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 **??** for an algorithm.

The `funcs` argument points to a `LoadSegFuncs` structure as defined in section **??** 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 a BPTR to the first segment of the overlay node on success, or `ZERO` on error.

11.4.7 Unloading Overlay Nodes

Unloading overlay nodes — and *not* the root node — of an overlaid 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 and thus depends on its structure; the implementation within the hierarchical overlay manager is documented here for completeness. Other overlay managers would need to perform different algorithms, as for example the flat overlay manager presented in section **??**.

The hierarchical overlay manager first finds the previous segment upfront the first segment of the overlay node to be unloaded, and cleans its `NextSegment` pointer to unlink all following segments. Then all subsequent segments are released through `FreeVec()` or, in case a custom allocator was provided for `InternalLoadSeg()`, whatever memory release function is appropriate. An implementation in C of this algorithm is given below, section **??** provides a complete implementation of the hierarchical overlay manager and thus an alternative implementation in assembler.

The following sample code releases the overlay node starting at segment $i > 0$ from a segment table of an overlay header, i.e. the structure from section **??** found in the first segment of the root node:

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

  /* Unlink the first segment to release */
  *segment = NULL;

  while (segment = (BPTR *)BADDR(segtbl[i++])) {
    FreeVec(segment);
  }
}
```

Note that a previous segment always exists because the root node populates at least entry 0 of the segment table, i.e. $i > 0$ upon entry. In addition, the AmigaDOS loader also zero-terminates the segment table, thus the above code cannot overrun its end.

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; however, see the note in section **??**, usage of `InternalLoadSeg()` for overlays is discouraged.

11.4.8 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. The overlay manager used does not matter. `UnLoadSeg()` will detect overlaid executables from the magic value in `oh_Magic` in the first segment passed, 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* calls `Close()` on `oh_FileHandle`, even if `oh_FileHandle` does not correspond to a BPTR to a `FileHandle` structure 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 set it to zero and then finally call into `InternalUnLoadSeg()` to perform all remaining cleanup steps. This strategy works because `Close()` on a `ZERO` file handle performs no operation and is legit. Due to these peculiarities,

`InternalLoadSeg()` should *not* be used for overlaid code, or at least, its `fh` argument should be a BPTR to a `FileHandle` such that loading is attempted through a regular file handle.

11.4.9 An Implementation of the Hierarchical 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 by which a subset of function arguments are passed in CPU registers. Earlier versions of the overlay manager required stack-based parameter passing. The version provided here is, however, also suitable for assembler programs and preserves all CPU registers.

When binding objects together to an overlaid executable, the linker checks whether a referenced symbol is within or outside the node it is referenced from. References that go to a parent node or to 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 segment containing the symbol is already present when the referencing hunk is loaded. Calls to such symbols thus become regular function calls whose jump destination is updated through the relocation mechanism of the Hunk format.

References to symbols within children nodes require another treatment as their segment is not necessarily in memory yet. Instead, the linker assigns to each such symbols a unique integer identifier and creates an entry in the symbol table. Each instance of the `SymTab` structure that is part of the `HUNK_OVERLAY` (section **??**) represents such a (forward) reference to a child node. The actual call to the referenced symbol is replaced by the linker to a call of a trampoline function

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

where `@ovlyMgr` is the entry point of the overlay manager and `symbX` is the integer identifier of the referenced symbol. The overlay manager then reads from its stack frame the return address, which instead points to the integer identifier. From that, it finds the `SymTab` structure representing the target function.

The symbol table contains both the ordinate and the level of the symbol which allows the overlay manager to check whether the node containing the symbol is already loaded. If this is not the case, it will unload the currently loaded node at the same level along with all its children, and then loads the required node from the file offset recorded in the symbol table.

Once the overlay node containing the symbol has been loaded or was found to be in memory already, the symbol address is computed from the offset in the symbol table and the address of the segment it is contained within. The symbol address replaces then the return address of the overlay manager, thus forwards the code to the desired function.

As symbols in children nodes can only be reached through a call of the overlay manager potentially loading the symbol, data in a child cannot be directly referenced from the parent node. A possible workaround for this limitation is to make such data available through accessor functions in children nodes that return pointers to the requested data.

The following code provides a general overlay manager for register-based or stack-based function 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_FirstSegment:        rs.l 1          ;Start of the overlay node
ot_SymbolSegment:       rs.l 1          ;Segment containing symbol
ot_SymbolOffset:        rs.l 1          ;Offset of symbol
ot_len:                 rs.b 0

;************************************************
;** Other stuff                                 **
;************************************************
MajikLibWord    =       23456           ;ignored by by the Os

       section NTRYHUNK,CODE            ;ensure this is first
;************************************************
;** Manager starts here                         **
;************************************************
Start:
        bra.w NextModule                ;Jump to init code

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

        dc.l       $ABCD                ;Magic longword for UnLoadSeg

;* The next four LWs are filled by the loader (LoadSeg())
oh_FileHandle:  dc.l 0                  ;Overlay file handle
oh_OVTab:       dc.l 0                  ;Overlay table
oh_Segments:    dc.l 0                  ;BPTR to Overlay hunk table
oh_GlobVec:     dc.l 0                  ;BPTR to global vector (unused)

                dc.l MajikLibWord       ;identifies hierarchical mngr
                                        ;used by SegTracker
                dc.b 7,"Overlay"        ;Identifier

;* the following data is specific to this manager

oh_SysBase:     dc.l 0                  ;additional pointer
oh_DOSBase:     dc.l 0                  ;to libraries

                dc.b "THOR Overlay Manager 1.2",0      ;another ID

@ovlyMgr:                               ;Entry-points
        movem.l d0-d3/a0-a4/a6,-(a7)    ;Saveback register

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

        move.l oh_OVTab(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 symbol
        move.l ot_OverlayLevel(a3),d0   ;get symbol level
        lsl.l #2,d0
        adda.l d0,a4
        move.l ot_Ordinate(a3),d0       ;get required ordinate
        cmp.l (a4),d0                   ;check resident ordinate
        beq.s .segmentresident          ;if equal, is already resident
                                        ;incorrect ordinate
                                        ;clear all other entries
        move.l oh_OVTab(pc),a1          ;get pointer to overlay table
        move.l d0,(a4)+                 ;fill with new ordinate
        move.l (a1),d1                  ;get size
        lsl.l  #2,d1
        adda.l d1,a1                    ;end of table

        moveq #0,d1
.do1:    cmp.l a1,a4                    ;terminate on end of table
         bhs.s .br1
         move.l d1,(a4)+                ;clear this
        bra.s .do1
.br1:
        move.l ot_FirstSegment(a3),d0   ;first segment to load
        add.l oh_Segments(pc),d0        ;plus BPTR of hunk table
        lsl.l #2,d0                     ;address of entry in segtable
        move.l d0,a4
        move.l -4(a4),d0                ;get previous segments
        beq.s .noprevious
        lsl.l #2,d0
        move.l d0,a2
        move.l d1,(a2)                  ;unlink fields before loading

                                        ;now free all children
        move.l oh_SysBase(pc),a6
.do2:    move.l (a4)+,d0                ;next hunk ?
         beq.s .br2
         lsl.l #2,d0
         move.l d0,a1                   ;->a1
         move.l -(a1),d0                ;get length
         jsr FreeMem(a6)                ;free this hunk
        bra.s .do2                      ;and now the next
.br2:
.retry:
        move.l oh_DOSBase(pc),a6
        move.l oh_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 oh_Segments(pc),d2       ;segment table
        moveq #0,d1                     ;no name (is overlay)
        move.l oh_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
.segmentresident:
        move.l ot_SymbolSegment(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

        movem.l (a7)+,d0-d3/a0-a4/a6
        rts

;***********************************************
;** Go here if we find an error                **
;***********************************************
.loaderror:
        movem.l d7,-(a7)
        move.l oh_SysBase(pc),a6
        move.l #$0700000C,d7
        jsr Alert(a6)                   ;Post alert
        movem.l (a7)+,d7
        bra.s .retry                    ;Retry or die

.noprevious:
        move.l oh_SysBase(pc),a6
        move.l #$8700000C,d7            ;dead end !
        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
        lea oh_SysBase(pc),a3
        move.l AbsExecBase.w,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
        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 oh_SysBase(pc),a6
        move.l oh_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
        jsr Alert(a6)
        moveq #30,d0                    ;Something went really wrong !
        rts

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

11.4.10 The MANX Overlay Manager

The Aztec C compiler from MANX offers an alternative overlay manager that is related to the Resource Manager present in the early 68K versions of MacOs. It does not organize overlay nodes in a hierarchy, but implements a flat organization of nodes — or *resources* as they would be called under MacOs. All nodes of the single level can be loaded independently of each other, either on demand through the function `segload()` in the MANX C library, or whenever a function of an overlaid node is called. The corresponding `freeseg()` function unloads a node again. A node consists of one or more segments that are loaded and unloaded jointly. Unfortunately, the implementation of this overlay manager depends on self-modifying code and ignores the instruction cache present in later members of the Motorola 68K family. It is therefore no longer safe to use.

Similar to the hierarchical overlay manager, the MANX overlay manager keeps meta-data for organizing the overlay nodes in the `HUNK_OVERLAY` hunk, though its format is different from the one in documentation in section **??**. The AmigaDOS `LoadSeg()` function does not care about the contents of this hunk as long as its first `LONG` word indicates its size.

Trampoline functions redirect the code flow for non-resident functions to the overlay manager. The trampolines are not part of `HUNK_OVERLAY` but reside at the beginning of the `HUNK_DATA` hunk, in the second segment of the root node relative to the `__H1_org` symbol. If a call to a function in another overlay node is called, then the jump goes through this trampoline, which initially calls the overlay manager.

The `HUNK_OVERLAY` hunk contains offsets within the file, the offsets to the trampoline functions and to the symbol table. This table provides information in which segment the referenced overlaid functions

reside. The contents of the overlay hunk are, as always, accessible through the `oh_OVTab` element of the `OverlayHeader` structure.

The overlay hunk for the MANX compiler reads on disk as follows:

Table 11.16: MANX Hunk Overlay Syntax

Size	Code	Syntax
	`HUNK_OVERLAY [0x3f5]`	Overlay table definition for MANX
32	l	Size of the overlay table, it is $l + 1$ long-words large.
Format for the MANX overlay manager, $l + 1$ long-words.		
32	o_d	Number of nodes in the overlay, excluding the root node
	`for(`i `= 1;`i `<` o_d`;`i`++) {`	For all nodes, excluding the root node
32	$o_p[i]$	Absolute offset of the `HUNK_HEADER` of the overlay node
16	$o_t[i]$	Offset of the first trampoline relative to the linker symbol `__H1_org`
16	$o_s[i]$	Offset to the symbol table relative to $\&o_p[1]$
	`}`	
	l `-=` $o_d \times 2 - 1$	Remove read data from the count.
	`for(`$s = 0$`;`$l \geq 0$`;`s`++) {`	Repeat over the symbol table
16	$o_h[s]$	Segment within which the symbols resides, or 0 for end of node
16	$o_c[s]$	Symbol count within segment $o_h[s]$
	l `-= 1`	Remove one long words.
	`}`	End of loop over table

In memory, it is approximately described by the following (pseudo-) structure, replacing the `OVTab` structure in section **??**. As in that section, some elements are variably sized and are thus hard to represent in the C syntax:

```
struct MANXOVTab {
    ULONG ot_NodeCount;          /* Number of overlay nodes            */
    struct OvNode {              /* One per overlay node               */
        ULONG ot_FilePosition;   /* Position of HUNK_HEADER            */
        UWORD ot_TrampolineOff;  /* Offset of the first trampoline     */
        UWORD ot_SymbolOffset;   /* Relative to &ot_NodeCount+1        */
    }     ot_Nodes[od-1];
}

struct MANXSymTab {
    UWORD ot_Segment;      /* Segment containing symbols or 0 for end  */
    UWORD ot_Count;        /* Number of trampolines to patch           */
} []                       /* Repeats multiple times */
```

The element o_d representing `ot_NodeCount` defines the number overlay nodes, all of which can be loaded or unloaded individually. The element $o_p[i]$ providing `ot_FilePosition` is the offset relative to the start of the file at which the `HUNK_HEADER` of the overlay node i is found. The offset $o_t[i]$ corresponding to `ot_TrampolineOff` is the offset of the first trampoline to a symbol within overlay node i; the offset is relative to the second segment of the root node, or more precisely, relative to `__H1_org`.

Finally, $o_s[i]$, or in memory `ot_SymbolOffset`, is used by the MANX overlay manager to find the first symbol descriptor within the second part of the `HUNK_OVERLAY`, i.e. a `MANXSymTab` structure. The offset

is relative to the third long word within this hunk, or as l is not part of the internal memory representation, relative to `&ot_NodeCount+1`. The target of this offset is a sequence of $o_h[s]$,$o_c[s]$ pairs. The first member of this pair, $o_h[s]$, or `ot_Segment` in memory, is the segment index containing the overlaid symbols; this clearly cannot be 0 as this would indicate a symbol within root node. The second member of the pair, $o_c[s]$ or `ot_Count` is the number of symbols within this segment. The symbols for the loaded node terminate with an $o_h[s]$ entry being 0.

The `HUNK_OVERLAY` hunk does not contain the symbol offsets itself. Rather, they are part of the trampoline which is included in the second segment of the root node. The offset to the first trampoline of an overlay node relative to the start the second segment is provided by $o_t[i]$, that is, by `ot_TrampolineOff`.

Each trampoline looks as follows on disk:

Table 11.17: MANX Overlay Trampoline

Size	Code	Syntax
16	`0x6100`	68K Opcode of `bsr.w`
16	t_j	Branch offset to overlay manager
8	t_n	Overlay node number
24	t_o	Offset of the overlay symbol within the loaded hunk

In memory, this is equivalent to the following structure:

```
struct MANXTrampoline {
    UWORD mt_BSR;               /* filled with 0x6100          */
    WORD  mt_OvMngrOffset;      /* Offset to overlay manager   */
    UBYTE mt_OverlayNode;       /* Overlay node, counts from 0 */
    UBYTE mt_SymbolOffset[3];   /* Big-endian 24-bit offset    */
};
```

The start of the trampoline is a word-sized relative subroutine jump to the overlay manager. As the trampoline is typically in the second segment of the root node, but the code of the overlay manager is in its first segment, this branch goes to another absolute jump to the overlay manager. The element t_j, or `mt_OvMngrOffset` is the branch distance to this jump. When building an overlaid binary, the MANX linker resolves all references to overlaid symbols to a trampoline as indicated above, and when the code of the loaded binary calls through them, the overlay manager fetches from the return stack of the 68K processor the address of the trampoline.

From t_n, or `mt_OverlayNode` in memory, it finds the entry in the first part of the `HUNK_OVERLAY`, namely a triple $o_p[i]$, $o_t[i]$, $o_s[i]$ of file offset, trampoline offset and symbol offset, that is, an `OvNode` structure. This overlay node index is zero-indexed, i.e. the first overlay node is node 0, corresponding to the first element of the `ot_Nodes` array. The t_o offset, or `mt_SymbolOffset` in memory, is finally the offset of the symbol within its segment.

When an overlay node is loaded, the overlay manager uses the symbol table consisting of `MANXSymTab` structures and relocates from them the trampolines to the symbols; that is, the opcode of the relative branch in the first word of the trampoline is replaced by an absolute jump[5], opcode `0x4ef9`. The subsequent long is filled by the address of the symbol, computed from the start of the hunk $o_h[s]$ in `HUNK_OVERLAY` plus the offset t_o stored in the trampoline. The branch offset t_j is moved to the last 16 bits of the trampoline to enable the overlay manager to restore it back when the overlay node is unloaded.

Such a patched trampoline then looks as follows:

[5]Here the MANX overlay manager fails to clear the instruction cache, causing failure on later members of the 68K family.

```
struct MANXPatchedTrampoline {
    UWORD mt_JMP;               /* filled with 0x4ef9          */
    APTR  mt_OVSymbol;          /* absolute symbol address     */
    WORD  mt_OvMngrOffset;      /* Offset to overlay manager   */
};
```

When unloading an overlay node, the original trampolines have to be restored such that a call to an overlay symbol triggers again loading the segments containing the symbol from disk. For that, `mt_JMP` is replaced by a `bsr.w` instruction, `mt_OvMngrOffset` is moved to the next word, and the overlay node is found by identifying the `OvNode` that contains the offset to the trampoline that is to be unloaded. As trampoline offsets are assigned in increasing overlay node order, it is sufficient to find the largest trampoline offset that is smaller or equal to the trampoline to the symbol to be unloaded. Finally, `mt_SymbolOffset` is re-computed by subtracting the base address of the segment from the absolute symbol address.

11.5 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 them.

11.5.1 The Version Cookie

The `Version` command scans a ROM-resident modules or all segments of an executable for the character sequence `$VER:` followed by at least a single blank space.

The syntax of the version cookie is formally specified as follows:

Table 11.18: Version Cookie

Size	Data	Syntax
40	`'$VER:'`	The version cookie identifier
≥8	`' '` ...	One or more blank spaces
≥8	*name*	Program name, terminated by `'\0'`, `'\n'`, `'\r'` or a digit between `'0'` and `'9'`
≥8	*version*	Major *version* of the program encoded as decimal number in ASCII
8	`'.'`	An ASCII dot
≥8	*revision*	Minor *revision* of the program encoded as decimal number
≥0	`' '` ...	Zero or more blank spaces
8	`'('`	An opening bracket
≥8	*day*	Day of the month, between 1 and 31, encoded as decimal number
8	`'.'`	A dot
≥8	*month*	Month, between 1 and 12, encoded as decimal number
8	`'.'`	A dot
≥8	*year*	Year, either as two or four decimal digits
8	`')'`	Closing bracket
≥0	`' '` ...	Zero or more blank spaces
≥8	*comment*	A comment, terminated by `'\0'`, `'\n'`, or `'\r'`

Everything following the version cookie is optional, even the version number may be omitted. However, the cookie is not particularly useful if the version number is not present. The string is terminated by an ASCII `NUL` character, i.e. `'\0'`, a line feed `'\n'`, or a carriage return `'\r'`.

If the number representing the year is below 1900, the `Version` command assumes a two-digit year and either adds 2000 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 revision is 3. The string behind the date is a comment and usually not printed by the `Version` command, unless instructed to do so with the `FULL` command line argument.

11.5.2 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 minimum stack size in bytes.

Formally, the stack size cookie looks as follows:

Table 11.19: Stack Size Cookie

Size	Data	Syntax
56	`'$STACK:'`	The stack size cookie identifier
≥0	`' ' ...`	Zero or more blank spaces
≥8	*stack size*	Required minimum stack size in bytes as ASCII encoded decimal number
≥0	*comment*	Terminated by `'\0'`, `'\n'`, or `'\r'`

The stack of the program is then 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 mountlist provide alternative information sources. 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.

11.5.3 Extending the Stack Size from the Stack Cookie

The *dos.library* provides with the `ScanStackToken()` an optimized function to quickly scan segments for the stack cookie and potentially adjust a default stack size to at least the value found in there.

```
stack = ScanStackToken(segment, defaultstack) /* since V47 */
D0                     D1       D2

LONG ScanStackToken(BPTR segment, LONG defaultstack)
```

This function scans the singly linked segment list starting at `segment` for a stack cookie, potentially adjusting the `defaultstack` size in bytes passed in. If a stack cookie is found, and the minimal stack size it finds is larger than the default stack argument, the stack size found in the stack cookie is returned in `stack`. If no stack cookie is found, or the value in the stack cookie is smaller than the `defaultstack` size, the default size is returned.

The `segment` is a BPTR to a singly linked list of segments, e.g. as returned by `LoadSeg()` function.

11.5.4 Runtime binding of BCPL programs

BCPL programs depend on a Global Vector that contains function entries and data that is made available to all its program units. AmigaDOS includes a runtime binder functionality that creates the Global Vector of such BCPL code from data found in the segments of the loaded binary and the *dos.library* public Global Vector.

Even though this mechanism is deprecated and AmigaDOS has long been ported to C and assembler, some of its components still depend on this legacy mechanism, namely all handlers and file systems mounted with a `GLOBVEC = 0` or `GLOBVEC = -3` entry in the mount list, see table **??** in chapter **??**. While newer handlers should not depend on this mechanism anymore, the Port-Handler mount entry, see section **??**, is created in the Kickstart ROM as BCPL handler, beyond control of the user.

The same legacy startup mechanism is also used by the (now deprecated) CLI, or to be more precise, by the resident segment `CLI` recorded in the *dos.library*, see section **??**. The AmigaDOS shell, even though it is build from the same source code and thus equivalent to the CLI, is represented by a different resident segment, namely `shell` and `BootShell`, and uses the C/Assembler startup mechanism. More details on this are in sections **??** and **??**.

If the above components — the Port-Handler or the CLI — are attempted to be customized, implementers need to be aware that their 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 or depend upon anymore.

The *Runtime Binder* of AmigaDOS, initiated only for BCPL handlers and BCPL processes, scans the segments of such programs for information on how to populate the Global Vector. The layout of the segments is as follows:

The *first long-word* of the segment, usually the entry point of the process for C/Assembler startup, contains instead the long-word offset from the start of the segment 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 right before the address computed from the offset.

The first long-word of the initialization data is the size of the global vector the program requires, i.e. the number of entries in the vector. This value is only used to check the following initializers for validity. The public 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 relative to the start of the segment to be filled into the Global Vector, the second (lower) address is the index of the Global Vector entry to be populated. An offset of 0 terminates the list.

The following assembler stub should 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. This register, along with registers `a0`, `a2`, `a4` and `a5` need to be preserved and restored before exiting the program through the BCPL "return from function" call.

```
                SECTION text,code

                XREF    _main           ; handler or shell main

G_GLOBMAX       EQU     150             ; size of GV
G_START         EQU     1               ; BCPL START functions

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

```
*               C Startup function, called for GlobVec=-1 or -2
*               see text below why this works

CStart:         sub.l   a0,a0           ; no startup message
                bsr.w   _main           ; need to GetMsg() in main
                rts

*               Align to a long word boundary
                CNOP    0,4

*               BCPL startup function, called for GlobVec=0 or -3
*               and also for the CLI (but not the Shell)

BCPLStart:      movem.l a0-a6,-(a7)     ; save for BCPL use
                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

*               Align to a long word boundary
                CNOP    0,4
BCPLTable:
                DC.L    0               ; End of global list
                DC.L    G_START,BCPLStart-CodeHeader
                DC.L    G_GLOBMAX       ; max global used (default)
CodeEnd:
                   END
```

Note that there are other differences for BCPL handlers the `main()` function of the handler needs to take care of. BCPL handlers do not receive their startup message in the process `pr_MsgPort`, but rather receive a BPTR to the startup `DosPacket` in register `d1` (see also sections **??**,**??** and **??**). Likewise, the CLI also receives the startup packet in this register rather than in its process port. For convenience, the above startup code converts it to a C-style pointer and provides it in register `a0` to the `main()` function of the handler or shell.

The `CStart` label is not called at all if the handler is mounted with `GLOBVEC=0` or `GLOBVEC=-3`, and thus would be not required for pure BCPL-type handlers or the CLI. It is included here to demonstrate another technique, namely *dual use* handlers that can be mounted both as C and as BCPL handlers. The FFS makes use of this technique to be able to drive both OFS-mounted floppy disks requesting the BCPL startup mechanism, as well as harddisk partitions mounting the FFS with parameters signaling C startup.

In such a case, i.e. if `GLOBVEC=-1` or `GLOBVEC=-2` is indicated in the mountlist, the code *is* started from the first byte of the first segment, which is in this case actually the long-word offset to the BCPL initializer list. This still works because the offset represents the 68K opcode of a harmless `ORI` instruction as long as it is small enough, i.e. below 64K. To signal the C startup mechanism, the `main()` function is now invoked with a `NULL` argument in register `a0`, indicating that the handler or CLI implementation shall wait for the startup package to arrive in the `pr_MsgPort` of its process instead. A corresponding handler main program for this startup code is provided as example in section **??**.

The Shell — or rather its BPCL incarnation as `CLI` — is also started through the `G_START` entry of the Global Vector and thus the above code may be used as universal “BCPL kludge” for both the CLI and handlers that depend on the legacy BCPL binding and startup mechanisms.

11.6 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 along with all the files included by it. Such object 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. In assembler, such symbols are defined by `xref`, in the C language they correspond to functions or objects declared by the `extern` keyword. The linker then combines multiple object codes, potentially along with static linker libraries (see section **??**), resolves all unreferenced symbols, and generates an executable binary file as output.

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

Table 11.20: Object File Format

Size	Code	Syntax
?	`HUNK_UNIT`	Defines the start of a translation unit, see **??**
	`do {`	Multiple segments follow
?	`HUNK_NAME`	Name of the segment, defines segments to merge, see **??**
2	m_t	Read the memory type of the next segment
30	h	Read the next hunk type
	`if (`h `== HUNK_CODE)` `parse_CODE`	Code and constant data, see **??**
	`else if (`h `== HUNK_DATA)` `parse_DATA`	Data, see **??**
	`else if (`h `== HUNK_BSS)` `parse_BSS`	Zero-initialized data, see **??**
	`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 `== HUNK_RELOC32)` `parse_RELOC32`	32-bit relocation, see **??**
	`else if (`h `== HUNK_RELOC32SHORT)` `parse_RELOC32SHORT`	32-bit relocation, see **??**
	`else if (`h `== HUNK_RELRELOC32)` `parse_RELRELOC32`	32-bit PC-relative relocation, see **??**
	`else if (`h `== HUNK_RELOC16)` `parse_RELOC16`	16-bit PC-relative relocation, see **??**
	`else if (`h `== HUNK_RELOC8)` `parse_RELOC8`	8-bit PC-relative relocation, see **??**
	`else if (`h `== HUNK_DRELOC32)` `parse_RELOC32`	32-bit base-relative relocation, see **??**
	`else if (`h `== HUNK_DRELOC16)` `parse_RELOC16`	16-bit base-relative relocation, see **??**
	`else if (`h `== HUNK_DRELOC8)` `parse_RELOC8`	8-bit base-relative relocation, see **??**
	`else if (`h `== HUNK_EXT)` `parse_EXT`	External symbol definition, see **??**
	`else if (`h `== HUNK_SYMBOL)` `parse_SYMBOL`	Symbol definition, see **??**

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

Since there is no `HUNK_HEADER` in object files, the memory attributes for the segment are instead stored in the topmost two bits of the hunk type itself. The memory type is then derived from m_t as in `HUNK_HEADER`, see table **??** in section **??**:

$$\mu_t = (m_t << 1) \mid \texttt{MEMF\_PUBLIC}$$

That is, the bit combinations 00, 01 and 10 correspond to `MEMF_ANY`, `MEMF_CHIP` and `MEMF_FAST`. Unlike in `HUNK_HEADER`, there is no documented way how to indicate a memory type explicitly, and the bit combination 11 does not have a documented meaning. As the interpretation of object files is up to the linker, it is suggested to store for $m_t = 3$ the memory type explicitly in the next long word, before the size of the hunk, similar to the layout of `HUNK_HEADER`.

11.6.1 HUNK_UNIT

This hunk identifies a translation unit and assigns a name to it. 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, and it is an indivisible unit of the object file as references between segments of a unit are usually already resolved. That is, upon linking, entire units are pulled into the final executable file. Typically, the name of the unit is ignored by linkers, and unless the name is defined by other means, it is usually set to the file name that was compiled to the object file.

The structure of this hunk is as follows:

Table 11.21: 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.

11.6.2 HUNK_NAME

While the `HUNK_NAME` hunk is already specified in section **??**, it does not have a particular meaning in executable files. In object files and libraries, however, the name of the segment determines which segments of the input files are merged together by the linker: if the names of two segments in two object files are identical, they will be merged to a single segment by the linker.

By convention, some segment names also have a special meaning:

Segments named `__MERGED` contain data or blank space (BSS) that is addressed relative to a base register. Typically, the startup code of the compiler loads one address register, usually `a4`, with the address of

the load address of the merged segment, or with an address 32K into the segment. The code then addresses all data within this segment relative to the base register. Offsets relative to this base register are relocated by HUNK_DRELOC32, HUNK_DRELOC16 and HUNK_DRELOC8, see sections **??** and following. A 32K offset is added to the data register if the total size of the base-register addressed data exceeds 32K; this makes both positive and negative offsets relative to the base register available, and thus allows to address 64K of data in total. Segments of this type are typically created by compilers in the *small data model*. The name stems for the limitation to 64K as the 68000 processor allows only 16-bit for base-relative addressing.

Segments named _NOMERGE are never merged to any other segment, under no circumstances, even in the small code or small data model.

The segment named NTRYHUNK always becomes the first segment of the created executable and thus should be the name of a code segment. This, for example, can be used to ensure that the overlay manager (see section **??**) is always placed at the beginning of the executable file, regardless of the link order.

Executable code and constant data typically ends up in segments named text. This name has, however, no further implications to the linking process.

Non-constant data in the *large data model* typically ends up in segments of the name data. In this model, data is addressed using absolute addressing, without a base register. Similar to the above, the name has no further implications on linking.

Uninitialized data in the *large data model*, that is, BSS segments, end up in segments named udata. The name has no further implications on the linking process.

Data that is placed in MEMF_CHIP memory typically ends up in data segments of the name chip. The name itself does not instruct the linker to request any special memory type, however, only the memory type m_t of the hunk does, see table **??** in section **??**. The name is only a convention.

In assembler, the segment name can be set by the section directive, compilers sometimes offer command line options to set the segment name, or select it according to their configuration.

11.6.3 HUNK_RELOC16

This hunk defines relocation information of one segment into another segment, and its format is identical to HUNK_RELRELOC32, see section **??** and table **??**. 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, including PC-relative 16-bit wide branches.

Table 11.22: Hunk Reloc16 Syntax

Size	Code	Syntax
	HUNK_RELOC16 [0x3ed]	16-bit PC-relative relocation information
	...	See table **??** in **??**

This restricts possible displacements to 16 bits. As the loading address of hunks is not under control of the linker, the only way how to ensure that the branch distance is within bounds is to merge the referencing segment and the target segment of the reference together. In the notation of table **??**, the hunks representing segments i and j must be merged. To ensure that this happens, their names as provided by HUNK_NAME shall be identical.

However, even if target and referencing segments are merged, it may still happen that the joined segment generated by merging two or more segments together is too long to allow 16-bit displacements. In such a case, the relocation cannot be performed. Then 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 PC-relative jump to an intermediate "automatic" vector that 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 bits, *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.

11.6.4 HUNK_RELOC8

This hunk defines relocation information of one segment into another segment, and its format is identical to `HUNK_RELRELOC32`, see section **??** and table **??**. 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.

The same restrictions as for `HUNK_RELOC16` applies, i.e. the segment within which the relocation offset is to be adjusted and the target segment shall be merged to a single segment as the AmigaDOS loader cannot resolve 8-bit relocations. This can be arranged by giving the two segments the same name.

Table 11.23: Hunk Reloc8 Syntax

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

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.

11.6.5 HUNK_DRELOC32

This hunk defines relocation of 32-bit data elements within a segment that is addressed relative to a base register. The name of such a segment shall be `__MERGED`. Such segments contain data and zero-initialized elements in the *small data model*, see also section **??**.

The format of the `HUNK_DRELOC32` hunk is identical to the `HUNK_RELOC32` hunk, see section **??** and table **??**, where each 32-bit wide relocation offset r_o points to a long-word within the preceding code hunk. The long word at this offset is then adjusted by the position of the first byte of this segment relative to the start of the `__MERGED` segment in the final executable into which it is merged. In case the target segment becomes larger than 32K, the base register points 32K into the segment, and then linkers need to subtract the additional 32K displacement from the r_o offsets when performing relocation.

Table 11.24: Hunk DReloc32 Syntax

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

11.6.6 HUNK_DRELOC16

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

The format of the HUNK_DRELOC16 hunk is identical to the HUNK_RELOC32 hunk, see section **??** and table **??**, where each 32-bit wide relocation offset r_o points to a signed 16-bit word within the preceding code hunk. The word at this offset is then adjusted by the position of the first byte of this segment relative to the start of the __MERGED segment in the final executable into which it is merged.

Table 11.25: Hunk DReloc16 Syntax

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

Similar to the comments made in section **??**, this hunk is typically used to resolve symbols that are reached relative through a base register, e.g a4. As base-relative addressing is restricted to 16-bit displacements for the 68000, linkers typically adjust the base register to point 32K into the __MERGED segment if the total amount of base-relative addressed data exceeds 32K in size. In such a case, they need to include an additional (negative) offset of -32K in r_0 when performing relocation.

11.6.7 HUNK_DRELOC8

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

The format of the HUNK_DRELOC8 hunk is identical to the HUNK_RELOC32 hunk, see section **??** and table **??**, where each 32-bit wide relocation offset r_o points to a byte within the preceding code hunk. The byte at this offset is then adjusted by the position of the first byte of this segment relative to the start of the __MERGED segment in the final executable.

Table 11.26: Hunk DReloc8 Syntax

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

11.6.8 HUNK_EXT

This hunk defines symbol names and corresponding symbol offsets or values within the current 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 11.27: 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 name 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 types are defined in `dos/doshunks.h` and shared with the `HUNK_SYMBOL` hunk, see section **??**.

The symbol type can be roughly classified into two classes: If bit 7 of the type is clear, it is a symbol *definition*. Such definitions can be referenced by another object file, or they can be used for debugging purposes. Symbol definitions do not have a bit size, they either represent an address or a value that is substituted into a reference of the symbol. If bit 7 is set, the symbol is *referenced* and requires resolution by a symbol definition (i.e. a corresponding symbol with bit 7 cleared) upon linking. References are sized, and an attempt to fit a symbol value too large for a reference either results in a linker error, or the creation of an automatic link vector.

In the following table $m_a[i]$ is the address of the first data byte of the segment that corresponds to the current segment once loaded, i.e. `&Data[0]` of the `SegmentList` structure (see chapter **??** and table **??** in section **??**):

Table 11.28: Symbol types in HUNK_EXT

`EXT_SYMB [0x00]`	Definition of a symbol, $s_v + m_a[i]$ is the address of the symbol. This type only exists in `HUNK_SYMBOL` hunks.
`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 segment i.
`EXT_ABS [0x02]`	Absolute value, s_v is the value of the symbol which is substituted into the executable by the linker when it is referenced. No relocation information is created, the absolute value is only substituted.
`EXT_RES [0x03]`	Not longer supported as it is part of the obsolete dynamic library runtime binding protocol, see [**?**] for more details.
`EXT_REF32 [0x81]`	Reference to a 32-bit symbol that is resolved by a corresponding `EXT_ABS` definition to an absolute value or by an `EXT_DEF` definition to relocation information.
`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, an object in a BSS hunk of the maximal size of all references to the symbol is created by the linker. Thus, this reference generates a zero-initialized object if no definition is found. Not all linkers support this type.

EXT_REF16 [0x83]	Reference to a 16-bit PC relative offset, requiring that the defining and referencing segments are merged together.
EXT_REF8 [0x84]	Reference to a 8-bit PC relative offset within the same segment.
EXT_DREF32 [0x85]	Reference to a 32-bit offset relative to a base register (typically a4), resolved by a EXT_DEF definition in a __MERGED segment. This and the next two types are not supported by all linkers.
EXT_DREF16 [0x86]	Reference to a 16-bit offset relative to a base register, resolved by a definition in a __MERGED segment.
EXT_DREF8 [0x87]	Reference to an 8-bit offset relative to a base register.
EXT_RELREF32 [0x88]	32-bit PC-relative reference for 32-bit address, this is resolved by an EXT_DEF definition into an entry into a HUNK_RELRELOC32 hunk by the linker. Not supported by all linkers.
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 space for the object 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 absolute value by an EXT_ABS definition.
EXT_ABSREF8 [0x8b]	8-bit absolute reference, resolved by the linker to an 8-bit absolute value through an EXT_ABS definition.

For references, s_n identifies the number of times the symbol is referenced, while the $s_o[]$ array defines the offsets into the current segment where the symbol is used and into which the symbol definition will be resolved during linking.

EXT_DREF32, EXT_DREF16 and EXT_DREF8 are references to symbols in a __MERGED segment defined somewhere outside of the current HUNK_UNIT by an EXT_DEF entry there. Thus, in the C language, they represent objects declared through extern, but defined in another translation unit. Upon linking, they become 32, 16 or 8 bit offsets relative to a compiler-specific base register pointing to or into the __MERGED segment. The offset to the symbols is computed by the linker and written to the byte offsets defined by the $s_o[]$ array.

This is quite similar to the HUNK_DRELOC32, HUNK_DRELOC16 and HUNK_DRELOC8 hunks, except that the latter define base-register relative references into the __MERGED segment that is part of the same HUNK_UNIT, and thus correspond in the C language to symbols defined in the same translation unit. Yet, as all __MERGED segments of all translation units are joined into one common segment, offsets into this segment need to be adjusted upon linking; the positions that require adjustment are provided by the HUNK_DRELOC hunks.

Typically, only 16-bit wide offsets are used, i.e. the EXT_DREF16 references and HUNK_DRELOC16 hunks for external or internal symbols. This is due to the 16-bit wide base-relative addressing mode of the 68000 processor limiting the size of the __MERGED segment to 64K. While 32-bit wide relative offsets would be possible on later members of the 68K processor family, the author is not aware of a compiler that makes use of this possibility. 8-bit references, even though representable as references or hunks, are too limited to be of practical value.

Common symbols are symbols that are referenced in multiple translation units but potentially nowhere defined. The size of the referenced object is given by s_c. If no corresponding symbol definition is found, the linker allocates space of a size that is determined by the maximum of all s_c values referencing the same symbol. Space for the object is then allocated within a BSS hunk by the linker without requiring an explicit symbol definition, and thus creates an object that is zero-initialized by the AmigaDOS loader. This mechanism is mostly required by FORTRAN and is therefore rarely used, and not all linkers support this mechanism. However, the SAS/C compiler can also be configured to emit such common symbols to resolve

zero-initialized objects defined (i.e. without an `extern` keyword) in a header file included from multiple translation units. This would otherwise generate multiple conflicting symbols of the same name.

11.7 Link Library File Format

Link Library files are collections of small compiled or assembled program units that provide multiple commonly used symbols or functions. Unlike AmigaOs libraries which are loaded at run time and shared between programs, 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 of a link library. It contains small frequently used service functions such as `CreateExtIO()`. While newer versions of the *exec.library* include with `CreateIORequest()` a similar function, some manual work was required for creating an `IORequest` structure in exec versions prior 36. To ease development, the former function was made available in a (static) library whose functions are merged with the compiled code.

Link libraries come in two forms: Non-indexed (old style) link libraries, and indexed libraries that are faster to process. Non-indexed link libraries are simply a concatenation of object files in the form presented in section **??** and table **??**. Then, of course, one translation unit as introduced by `HUNK_UNIT` **??** is not necessarily terminated by an `EOF` as specified in table **??**, but possibly followed by a subsequent program unit, starting with another `HUNK_UNIT`.

Non-indexed link libraries do not require any tools beyond a compiler or assembler for building them. The AmigaDOS `Join` command is sufficient to create 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.

Indexed libraries are faster to parse as they contain a compressed index of all symbols defined in the library. It consists at its topmost level of two hunks: one containing the program units, and second hunk containing a symbol table with an index that are repeated until the end of the file.

The overall format of indexed libraries is depicted in table **??**.

Table 11.29: Indexed Library

Size	Code	Syntax
	`do {`	Multiple repetitions of the following
?	`HUNK_LIB [0x3fa]`	Object code modules, see section **??**
?	`HUNK_INDEX [0x3fb]`	Indices into `HUNK_LIB`, see section **??**
	`} while(!EOF)`	Until the end of the file

11.7.1 HUNK_LIB

The `HUNK_LIB` hunk contains the actual payload in the form of multiple code, data or BSS hunks along with their relocation, symbol and debug information. It looks almost like the contents of a `HUNK_UNIT` hunk, with a couple of changes noted below.

Table **??** depicts the syntax of this hunk.

Table 11.30: Hunk LIB Format

Size	Code	Syntax
?	`HUNK_LIB [0x3fa]`	Identifies the start of an indexed library
32	*l*	Length of this hunk in long-words not including the header and this length field

	`do {`	Multiple segments follow
2	m_t	Read the memory type of the next hunk
30	h	Read the next hunk type
	`if (`h `== HUNK_CODE)` `parse_CODE`	Code and constant data, see **??**
	`else if (`h `== HUNK_DATA)` `parse_DATA`	Data, see **??**
	`else if (`h `== HUNK_BSS)` `parse_BSS`	Zero-initialized data, see **??**
	`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 `== HUNK_RELOC32)` `parse_RELOC32`	32-bit relocation, see **??**
	`else if (`h `== HUNK_RELOC32SHORT)` `parse_RELOC32SHORT`	32-bit relocation, see **??**
	`else if (`h `== HUNK_RELRELOC32)` `parse_RELRELOC32`	32-bit PC-relative relocation, see **??**
	`else if (`h `== HUNK_RELOC16)` `parse_RELOC16`	16-bit PC-relative relocation, see **??**
	`else if (`h `== HUNK_RELOC8)` `parse_RELOC8`	8-bit PC-relative relocation, see **??**
	`else if (`h `== HUNK_DRELOC32)` `parse_RELOC32`	32-bit base-relative relocation, see **??**
	`else if (`h `== HUNK_DRELOC16)` `parse_RELOC16`	16-bit base-relative relocation, see **??**
	`else if (`h `== HUNK_DRELOC8)` `parse_RELOC8`	8-bit base-relative relocation, see **??**
	`else if (`h `== HUNK_EXT)` `parse_EXT`	External symbol definition, see **??**
	`else if (`h `== HUNK_SYMBOL)` `parse_SYMBOL`	Symbol definition, see **??**
	`else if (`h `== HUNK_DEBUG)` `parse_DEBUG`	Debug information, see **??**
	`else if (`h `== HUNK_END) break`	abort this segment
	`else ERROR_BAD_HUNK`	an error
32	h	Read next hunk type
	`} while(true)`	Repeated until `HUNK_END`
	`} while(!EOF)`	Repeated with the next hunk until the file ends

The memory type for data, code and BSS segments is derived from m_t in the same way as in object files, see section **??**.

Additional restrictions arise for the `HUNK_EXT` hunk. Since symbol definitions are now included in the `HUNK_INDEX` hunk, they shall be removed from `HUNK_EXT` hunks. The corresponding symbol types to be removed are those with $s_t < 128$ listed in table **??** in section **??**. Unlike symbol definitions, symbol references corresponding to $s_t \geq 128$ shall be retained because only the reference name, but not the type of the reference is included in `HUNK_INDEX`. The translation unit name and the hunk names shall also be stripped, that is, neither `HUNK_UNIT` nor `HUNK_NAME` shall be included in `HUNK_LIB`. The corresponding

names are also defined in HUNK_INDEX by means of the string table included there.

Due to restrictions of HUNK_INDEX, the size of a HUNK_LIB shall not exceed 2^{16} long-words and shall be split over multiple HUNK_LIB, HUNK_INDEX pairs otherwise.

11.7.2 HUNK_INDEX

The HUNK_INDEX hunk contains a string table and indices into the preceding HUNK_LIB.

Table **??** depicts the syntax of this hunk.

Table 11.31: Hunk Index Format

Size	Code	Syntax
32	HUNK_INDEX [0x3fb]	Defines symbols and references into the library
32	l	Length of this hunk in long-words
16	s_l	Length of the string table in **bytes**, shall be even
	i=0	Start with the first symbol
	do {	Repeat over the strings
?	$s_y[i]$	A NUL-terminated (C-style) string
	s_l -= strlen($s_y[i$++$]$)+1	Remove from the length of the symbol table
	} while($s_l > 0$)	Repeat until all s_l bytes are parsed
	do {	Loop over translation units
16	u_o	Offset in bytes of the unit name into the string table
16	h_o	Offset in long words of the first hunk of the unit within HUNK_LIB
16	h_c	Number of hunks within the unit
	for(j=0;j<h_c;j++) {	Loop over all hunks
16	h_n	Offset in bytes of the hunk name into the string table
2	m_t	Memory type of the hunk
14	$\hat{h}$	Abbreviated hunk type
16	x_c	Number of references in the hunk
	for(k=0;k<x_c;k++) {	Loop over references
16	x_n	Offset of the reference name into the string table
	}	
16	d_c	Number of definitions in the hunk
	for(k=0;k<d_c;k++) {	Loop over definitions
16	d_n	Offset of the defined name into the string table
16	$\hat{s}_v$	Abbreviated symbol value
8	a_u	Bits 23-16 of EXT_ABS definition
1	0	This bit shall be 0 to identify a definition
1	a_s	Sign bit and bits 30 to 24 of an EXT_ABS definition
6	$\hat{s}_t$	Abbreviated symbol type
	}	
	}	
	} while(!end)	Repeated until the end of hunk is found

The initial part, the string table, contains all strings that can be used by the rest of the hunk. Strings within this table are indexed as byte offset from the start of the string table, i.e. the first string has offset 0. To enable unnamed hunks, the first entry in a string table shall be the empty string, that is, an isolated 0-byte. The string table is potentially zero-padded to make its length even. To keep the string table as short as possible, it is desirable to keep only unique strings and avoid duplicates.

The rest of HUNK_INDEX contains the offsets into the hunks along with symbols referenced and defined

within them. The first loop runs over all units in the library; there, u_0 provides the name of a unit that would be contained in the stripped HUNK_UNIT hunk, though as byte offset from the start of the string table. The element h_o is the number of long words from the start of the HUNK_LIB hunk to the m_t element in table **??** of the first hunk of the translation unit whose name is provided through u_0. The number of hunks in this unit is given by h_c. The subsequent loop iterates over all hunks in this unit.

The next following element, h_n, defines the name of the hunk, representing the contents of a stripped HUNK_NAME hunk, again as byte offset from the start of the string table. The memory type of the hunk m_t is expressed in two bits, and the value here shall be identical to m_t in the HUNK_LIB hunk, see table **??** in section **??**. The hunk type $\hat{h}$ itself is abbreviated, i.e. only the lower 14 bits of the hunk type h in the HUNK_LIB hunk are stored. Otherwise, $\hat{h}$ shall be identical to h.

The first part of the subsequent data defines references, that is, symbols that are used but not defined within the hunk. The number of references is provided by x_c. The x_n values define the names of these symbols as byte offsets from the start of the string table. The reference type s_t, see table **??**, section **??**, of the symbols is then found in an HUNK_EXT hunk as part of the preceding HUNK_LIB hunk. It is not included in the HUNK_INDEX hunk.

Symbol definitions follow; unlike references, they are represented completely in the HUNK_INDEX hunk and stripped from the HUNK_LIB hunk. The d_c element provides their number and the d_n values identify the names of the symbols as byte offset into the string table. The $\hat{s}_v$ values are the 16 least significant bits of the full symbol value s_v otherwise contained in the HUNK_EXT hunk, see table **??** in section **??**. The abbreviated symbol type $\hat{s}_t$ contains the 6 least significant bits of the full symbol type s_t defined in table **??**. Bits 6 and 7 of s_t are not represented and are always inferred to be 0 as this part of HUNK_INDEX only contains symbol definitions.

In case $\hat{s}_t$ indicates a symbol of type EXT_ABS, the elements a_u and a_s provide additional bits of its (absolute) value. These elements are ignored for all other symbol types. The element a_u stores then bits 23 to 16 of the absolute symbol value, and the bit a_s is replicated into bits 31 to 24 of the symbol, allowing negative values.

As seen from this definition, symbols representing addresses in hunks are limited to 16 bits in size, and thus restricted to 64K within a hunk. This is typically not a problem as link libraries usually contain short service functions. Symbols representing absolute values are limited to values from $-2^{24}+1$ to $2^{24}-1$, and thus, for example, addresses of Amiga custom chips can be expressed. They are split over $\hat{s}_v$, a_u and a_s. Symbols of type EXT_COMMON or EXT_RELCOMMON cannot represented at all in indexed link libraries.

Chapter 12

Direct Packet Communication

The *dos.library* communicates with handlers or file systems via *DosPackets* or short *Packets* — more on this structure in section **??** — which ride on top of exec messages. The message-based inter-process communication system of the exec kernel is described in more detail in [**?**]. DosPackets are send to a `MsgPort` of a handler or file system to request a particular action, such as to open a file or read data. The port is typically the process port `pr_MsgPort` of the handler process. Chapter **??** provides more details on processes and the `Process` structure containing this port. Once the handler has performed the requested action, it replies the packet and delivers through it also a primary and a secondary result code. The primary result code in the packet is typically delivered as result code of a corresponding function of the *dos.library*, whereas the secondary result code is often the error code that can be retrieved by `IoErr()`.

While it is in many cases more practical to interact with handlers through the functions of the *dos.library* listed so far, it is also possible and sometimes even necessary to communicate with the handler on a lower level directly; this becomes necessary if the desired activity is not exposed as function in the library.

The functions and structures in this chapter perform direct communication with handlers through packets and thus form the lower level interface between the *dos.library* and its handlers. Many of the higher level functions such as `Open()` or `Read()` call through the functions in this section.

While most functions of the *dos.library* are synchronous, i.e. wait for the handler to complete the requested action, the direct packet interface also allows asynchronous input and output. This keeps the process initiating an operation running while the handler is working on the requested operation in parallel. The result of the operation is then delivered to a `MsgPort` of the originating process at a later time.

12.1 Request an Action from a Handler and Wait for Reply

The `DoPkt()` function requests an activity from a handler, including arguments, waits for the handler to perform this activity and returns the result. This is the generic synchronous input/output request function through which the *dos.library* routes most of its function[1].

```
result1 = DoPkt(port,action,arg1,arg2,arg3,arg4,arg5) /* since V36 */
D0              D1    D2    D3   D4   D5   D6   D7

LONG DoPkt(struct MsgPort *,LONG,LONG,LONG,LONG,LONG,LONG)
```

This function performs low-level communication to a target message port. The `port` is the `MsgPort` of the handler to contact. Depending on the context, this port should be taken from various sources. If low-level file I/O is to be performed, the best source for the port is the `fh_Type` pointer in the `FileHandle` structure.

[1] Unfortunately, the functions of the library do not call through the `DoPkt()` library vector. This is probably a defect.

If the communication is related to a lock, the `fl_Task` element of the `FileLock` is the recommended source. Notification requests carry in `nr_Handler` a pointer to the port to be used. For activities unrelated to locks, files or notification requests, the `dol_Task` element of the `DosList` structure is yet another source. To obtain such a structure from a path, use the `GetDeviceProc()` function, see section **??**.

`action` identifies the activity to be performed by the handler or file system. Chapter **??** lists the packet types and how they relate to the functions of the *dos.library*.

`arg1` through `arg5` are arguments to the handler. Even though the packet interface supports up to 7 arguments, only up to 5 can be provided through this function. If more arguments become necessary, the packet interface needs to be used manually, see section **??** for the raw packet interface.

This function returns the primary result code of the handler, and installs the secondary result code in `IoErr()`, see also section **??**. In particular, `IoErr()` is always set, even if the handler does not deliver a meaningful secondary result code.

If the caller is a process and the `pr_PktWait` pointer in the `Process` structure is set, `DoPkt()` calls through it to wait for the packet (or rather the message carrying it) to return, see chapter **??** for its prototype. Otherwise, `DoPkt()` waits on `pr_MsgPort` with `WaitPort()` and removes the message through `GetMsg()`. If the caller is a task, the function builds an exec `MsgPort` on the fly and waits on this temporary port — unlike many other functions of the *dos.library*, this function is even callable from tasks.

If the return `MsgPort` contains a message different from the one carrying the issued packet, `DoPkt()` aborts with a dead-end alert of type `AN_QPktFail`, defined in `exec/alerts.h`. Note that this is quite different from exec style communications with (exec) devices through `DoIO()`; the latter function is able to extract the send `IORequest` from the port without creating a conflict if another message is still pending in the port. This problem of packet communication manifests, for example, when attempting to perform I/O operations through the *dos.library* while the Workbench startup message is still queued in the process message port.

Because packets typically require less than 5 arguments, additional function prototypes are supplied that take less arguments. They all access the same vector within the *dos.library*, the only difference is that the function prototypes do not enforce initialization of the data registers carrying the unneeded arguments. These functions are named `DoPkt0()` to `DoPkt5()` and carry 2 to 7 arguments: The target `port`, the requested `action` and 0 to 5 additional arguments.

12.2 Asynchronous Packet Interface

The functions in this section implement the asynchronous packet interface of the *dos.library* and reassemble to a large degree the device interface of the *exec.library* (see also [**?**]), though the functions and structures are somewhat different[2].

To trigger an asynchronous input or output operation, the caller first needs to create a `DosPacket`, see section **??**, fill this packet with the action requested from the handler and zero or more additional arguments, and then send the packet to the handler with `SendPkt()`, see section **??**. The sending process keeps running after the packet has been submitted to the handler, and is then free to perform other activities, for example initiating another operation from a second handler.

The `WaitPkt()` function in section **??** waits for the packet to return and make it accessible to the initiator again. Unfortunately, the *dos.library* does not provide a function to abort an issued packet, even though the `AbortPkt()` function of section **??** suggests otherwise. It is currently non-functional. Also, the library neither provides a function to test whether a packet has already been replied and is thus completed. To do so, the initiator of a packet needs to check its `MsgPort` manually for such a packet. Unfortunately, `WaitPkt()` is not able to test multiple ports for incoming packets or test the status of a packet.

[2]Historically, it seems plausible that the exec design is a copy of the Tripos design and not vice versa.

12.2.1 The DosPacket Structure

Communication with handlers is based on *packets*. For example, the DoPkt() function creates a packet on the fly on the stack, submits it to the handler via the given port, and waits for it to return. This section describes the packet structure and its elements.

A packet is represented by a DosPacket structure documented in dos/dosextens.h:

```
struct DosPacket {
    struct Message *dp_Link;
    struct MsgPort *dp_Port;
    LONG dp_Type;
    LONG dp_Res1;
    LONG dp_Res2;
    LONG dp_Arg1;
    LONG dp_Arg2;
    LONG dp_Arg3;
    LONG dp_Arg4;
    LONG dp_Arg5;
    LONG dp_Arg6;
    LONG dp_Arg7;
};
```

Packets ride on top of exec messages, see [**?**] and exec/ports.h, but they do not extend the Message structure as it would be usually the case. Instead, mn_Node.ln_Name of the exec message is (mis-)used to point to the DosPacket and its dp_Link element points back to the message, thus establishing a two-way linkage between packet and message. The reply port of the message in mn_ReplyPort is *not* used; instead, the message carrying the packet is send back to dp_Port once the handler is done.

Members of the DosPacket structure shall be initialized as follows:

dp_Link shall point to the message which is used for transmitting the DosPacket. The message node name in mn_Node.ln_Name shall be initialized to point back to the DosPacket.

dp_Port shall point to the MsgPort structure to which the packet shall be send back after the handler has completed the requested activity. This is typically, but not necessary the pr_MsgPort of the process sending the packet. See chapter **??** for the definition of the Process structure. DoPkt() uses the process message port of the caller as reply port for the packet if the caller is a process, otherwise a temporary port is created just for the purpose of sending a packet.

dp_Type identifies the action requested from the handler. It shall be filled by the process requesting an activity from a handler and is interpreted by the handler. Chapter **??** lists the documented packet types. This element corresponds to the action argument of DoPkt().

dp_Res1 is the primary result code of the requested activity and filled by the handler before returning the packet. For many, but not for all packet types, this is a Boolean result code that is 0 for failure and non-zero for success. Many packet-based functions of the *dos.library* including DoPkt() return dp_Res1 as their (primary) result.

dp_Res2 is the secondary result code filled by the handler and is typically an error code on failure. Many functions of the *dos.library* install this error code into IoErr(), and so does DoPkt(). Section **??** lists the error codes defined by the *dos.library* handlers should use to communicate error conditions.

dp_Arg1 to dp_Arg7 provide additional arguments to the handler. They shall be filled by the issuer of a packet; which and how many arguments are required depends on the packet type encoded by dp_Type. Most packet types do not require all 7 possible arguments; in such a case, only the necessary arguments need to be initialized. The first 5 arguments correspond to arg1 to arg5 of the DoPkt() function.

12.2.2 Send a Packet to a Handler Asynchronously

The SendPkt() function transmits a packet to a target message port of a handler without waiting for it to return. Instead, a reply port is provided to which the packet will be returned once the handler acted upon it.

```
SendPkt(packet, port, replyport) /* since V36 */
         D1     D2        D3

void SendPkt(struct DosPacket *,struct MsgPort *,struct MsgPort *)
```

This function transmits packet to the handler port, requesting to return it to replyport. The function returns immediately without waiting for the completion of the requested action.

The packet shall be partially initialized; in particular, dp_Link shall point to an exec Message whose mn_Node.ln_Name element points back to packet. SendPkt() *does not* supply or initialize such a message. It is recommended to create packets through AllocDosObject() which also initializes them properly, see below and section **??**.

dp_Type shall be filled with the requested action, i.e. an identifier specifying the type of activity requested from the handler, see chapter **??**. Depending on this type, a subset of the packet arguments dp_Arg1 through dp_Arg7 carry additional information for the requested action and shall be initialized accordingly.

DosPackets can be constructed in multiple ways: AllocDosObject(DOS_STDPKT,NULL) allocates a packet along with a message and initializes the linkage between the structures, see **??** for the full description of this function. It creates a StandardPacket which contains both a Message and a DosPacket. This structure is defined in dos/dosextens.h and looks as follows:

```
struct StandardPacket {
   struct Message   sp_Msg;
   struct DosPacket sp_Pkt;
};
```

AllocDosObject() returns a pointer to a DosPacket structure, i.e. the address of the sp_Pkt element, and *not* a pointer to a StandardPacket, see also section **??**.

Another option is to allocate memory for the packet from the heap and initialize the linkage between the structures manually:

```
struct StandardPacket *CreatePacket(void)
{
  struct StandardPacket *sp;

  sp  = AllocMem(sizeof(struct StandardPacket),MEMF_PUBLIC);

  sp->sp_Msg.mn_Node.ln_Name = (UBYTE *)&(sp->sp_Pkt);
  sp->sp_Pkt.dp_Link         = &(sp->sp_Msg);

  return sp;
}
```

At least the DosPacket structure shall be aligned to a long word boundary to make it and its elements accessible to BCPL handlers. As the Message structure occupies an integer number of long words, the above code is sufficient to ensure this requirement.

Even though it is in principle possible to allocate the packet on the stack with the D_S() macro from section **??**, this idea is discouraged. It only works if the same function or one of its callees also wait for the packet to return, which is hard to enforce in a design based on asynchronous packet communication. Otherwise, the stack frame containing the packet would be destroyed upon returning from the packet constructing function.

12.2.3 Waiting for a Packet to Return

The `WaitPkt()` function waits on the `pr_MsgPort` of the calling process (see chapter **??**) for a packet to return and returns a pointer to the received packet.

```
packet = WaitPkt() /* since V36 */
D0

struct DosPacket *WaitPkt(void);
```

This function receives a packet returning from a handler; it is also implicitly called by `DoPkt()` after sending the messages to the handler[3].

If the `pr_PktWait` pointer in the `Process` structure is set, `WaitPkt()` calls through this function to wait for the arrival of a message, see chapter **??** for details. Otherwise, the `WaitPkt()` function calls `WaitPort()` to wait for the arrival of a message on `pr_MsgPort` of the calling process, and then calls `GetMsg()` to remove it from the port. The function then returns `mn_Node.ln_Name` of the received message, i.e. the packet linked to the message.

This function does not test whether the received message does, actually, belong to a packet. The caller shall ensure that only messages corresponding to `DosPackets` can arrive at the process message port, and shall remove all other messages from this port upfront.

As `WaitPkt()` always waits on the process message port of the caller, this only works if the packet was send with the `replyport` argument of `SendPkt()` set to `pr_MsgPort` of the calling process.

12.2.4 Aborting a Packet

The purpose of the `AbortPkt()` function is to attempt to abort a packet already send to a handler. However, as of the current Os release, it does nothing and is not functional.

```
AbortPkt(port, pkt) /* since V36 */
          D1    D2

void AbortPkt(struct MsgPort *, struct DosPacket *)
```

What this function should do is to scan `port`, presumably the `MsgPort` of the handler to which `pkt` was send, and dequeue it there if the handler is not yet working on it. Then, it would be placed back into the port of its initiator. However, as of V47, *this function does nothing*.

12.3 Reply a Packet to its Sender

The `ReplyPkt()` function returns a packet to its initiator, filling the primary and secondary result codes. This function is intended to be used by handlers and file systems.

```
ReplyPkt(packet, result1, result2) /* since V36 */
           D1      D2       D3

void ReplyPkt(struct DosPacket *, LONG, LONG)
```

[3]Unfortunately, `DoPkt()` and the rest of the *dos.library* does not call through the `WaitPkt()` function vector. This is probably a defect.

This function fills dp_Res1 and dp_Res2 of the packet with result1 and result2, and sends the packet back to the port pointed to by the dp_Port element of the packet, i.e. the initiating port. Note that mn_ReplyPort of the message pointed to by dp_Link is ignored, i.e. packet communication does *not* follow the exec protocol for replying messages.

The pkt argument may be NULL in which case this function does nothing.

The result1 argument is the primary result code and identical to the return code of many *dos.library* functions. dp_Res2 is the secondary result code and typically accessible through IoErr() if the packet is received by DoPkt(), see section **??**.

Chapter 13

Handlers, Devices and File Systems

The *dos.library* does not implement most of its functions itself; operations that are related to files, locks, notification requests and many others are off-loaded to file systems or handlers. Thus, the *dos.library* establishes a virtual file system that provides a common API for its clients, but depends on handlers and file systems to implement its functions.

This chapter provides insight into the interface between the *dos.library* and its handlers, and also lists the features of the handlers included in AmigaDOS.

13.1 The Handler Interface

A handler or a file system is an Amiga process that retrieves commands in the form of `DosPacket` structures; this structure is defined and discussed in section **??**. The main loop of a handler is conceptually similar to a program implementing a graphical user interface, except that the latter retrieves messages via the *intuition* IDCMP port and reacts on the messages whereas handlers receive packets from the `pr_MsgPort` of their processes, or any other port they indicated to their clients.

To interface with a handler, the *dos.library* first needs to find a `MsgPort` of the handler, either from a path or from an already existing object such as a lock or a file handle. For Locks, the `fl_Task` element of the `FileLock` structure contains the port, and for files the `fh_Type` element is the pointer to the port, see sections **??** and **??**. Notification requests carry the pointer in `nr_Handler`. If only a path is given, then section **??** describes how a port and thus a handler responsible for a path is identified.

13.1.1 Locating a Handler from a Path

Handlers are located by the *dos.library* function `GetDeviceProc()`. This function receives an absolute or relative path, and from this path it determines a `MsgPort` through which the handler responsible for the path can be contacted; the function itself is specified in section **??**. All other functions of the *dos.library* requiring to resolve a path to a handler also call through `GetDeviceProc()`[1], e.g. so does `Lock()` and also `DeviceProc()` as legacy function. While `Open()` also calls through `GetDeviceProc()`, it processes a couple of special cases itself.

For a relative path, i.e. a path not containing a colon (“`:`”), `pr_CurrentDir` of the calling process is checked. If non-`ZERO`, the `fl_Task` element of the `FileLock` kept there is a pointer to a `MsgPort` through which packets are send to the handler. If it is `ZERO`, then `pr_FileSystemTask` provides the `MsgPort` to contact. More on this in section **??** and chapter **??**.

If an absolute path to the (pseudo-)device `CONSOLE` provided, the `MsgPort` in `pr_ConsoleTask` is contacted. This port belongs to the current console the calling process is running in. While `CONSOLE`

[1] Unfortunately, they do not call it through the vector of the library, which is probably a defect.

is handled within GetDeviceProc(), the "*", CONSOLE: and NIL: are filtered out by Open(). If a path relative to the pseudo-assign PROGDIR has been passed, the lock in pr_HomeDir is used to identify a handler[2].

If the path is relative to NIL:, no handler exists and the result is NULL. This is not a failure, but rather an indicator to higher level functions that a dummy handler is requested that does nothing. As for paths identifying the console, this case is already filtered out by Open().

For every other absolute path, the *dos.library* walks the device list (see chapter **??** and **??**) to find a suitable MsgPort through which to contact a handler. For this, it compares the string upfront the colon with the dol_Name element of each DosList entry in the device list.

Once a suitable entry has been found, GetDeviceProc() tests dol_Type which contains the type of the entry as given by table **??** in chapter **??**. For volumes, dol_Task is non-NULL if the volume is currently inserted; it is then the pointer to the port of the responsible handler. If it is NULL, the volume is currently not available and then requested from the user if pr_WindowPtr allows to do so. If this requester is aborted or disabled, path resolution fails.

For regular assigns or multi-assigns, dol_Task is a pointer to the port of the handler responsible for the first directory in the assign, and dol_Lock provides the lock to this directory. The same port can also be found in the fl_Type element of this lock. All additional directories in a multi-assign are represented by locks in the AssignList structure, and the port can taken from the corresponding fl_Type element, see chapter **??** for a detailed discussion of this structure.

Late and non-binding assigns do not resolve to a port immediately. Instead, the DosList provides in dol_AssignName a path to the target directory which is resolved recursively into a lock relative to which the provided path is interpreted. The lock from this target directory then contains a suitable handler port. The only difference between late binding and non-binding assigns is that the former decay into regular assigns once a suitable lock has been found.

Finally, if dol_Type is DLT_DEVICE indicating an entry of a handler or file system, dol_Task is a pointer to a MsgPort through which the handler can be contacted, provided it is non-NULL.

13.1.2 Starting a a Handler

If the dol_Task element of a handler entry is NULL, this is an indicator that GetDeviceProc() needs to start a new handler process for the provided path. It first checks whether dol_SegList is ZERO. If it is, the handler code is not yet resident in memory and will be loaded from the file name indicated in dol_Handler through LoadSeg(), see section **??**, and dol_SegList will be filled with the loaded segment list.

Next, a new process is created; details depend on the dol_GlobVec element of the device list entry, see Table **??** in chapter **??**. For C or assembler handlers with a dol_GlobVec value of -1 or -2, the process is started from the first byte of dol_SegList. For all other values of dol_GlobVec, i.e. BCPL handlers, it is run from the START entry of the Global Vector, see also section **??**. To further initialize the handler, a startup package is delivered: for C or assembler handlers, the startup packet is send to the pr_MsgPort of the handler process; for BCPL handlers, the startup packet becomes the first argument of the START function at index 1 of the Global Vector. If the "fake" BCPL startup code from section **??** is used, this packet is either delivered in register a0 of the handler main function, or shall be retrieved from pr_MsgPort if a0 is NULL.

While the handler is starting, GetDeviceProc() waits for the startup packet to return and keeps the device list locked. This is an exclusive lock if dol_GlobVec is larger or equal than -1 and otherwise a shared lock. This implies that attempting to gain access to the device list within the handler startup by LockDosList() can deadlock. Even if such a lock is not requested explicitly, *dos.library* functions can

[2]Unfortunately, GetDeviceProc() has issues in both handling CONSOLE and PROGDIR correctly if no handler or no lock representing these elements is available. Instead, it attempts then to find a device, volume or assign of the name CONSOLE or PROGDIR. It is unclear whether this is intentional or a defect.

require implicitly such a lock, and thus attempting to access files or locks from the handler process should be avoided, not only during startup.

The startup packet consists of a `DosPacket` structure as it is also used to submit requests to the handler, see section **??** for its definition. For the purpose of handler startup, its elements are populated as follows:

Table 13.1: Handler Startup Packet

DosPacket Element	Value
`dp_Type`	`ACTION_STARTUP (0)`
`dp_Arg1`	BPTR to BSTR of path
`dp_Arg2`	Copy of `dol_Startup`
`dp_Arg3`	BPTR to `DosList`
`dp_Res1`	Success indicator
`dp_Res2`	0 or error code
`dp_Arg4`	APTR to `MsgPort` or `NULL`

`dp_Type` is set to `ACTION_STARTUP`, which is defined to be 0. As the startup packet is always received first, there is no need to test for this particular type. In an alternative implementation strategy, it may be processed as `ACTION_NIL` packet within the main handle loop, noting that the encoding of `ACTION_NIL` is also 0.

`dp_Arg1` is set to a BPTR to a BSTR representing the path under which the client of the *dos.library* attempted to access the handler. Note that this is not a `NUL`-terminated C string, but a BSTR whose first element is the size of the string. This is the *full* path which triggered the request for the handler, not only the device name. For the CON-Handler for example, this is the window specification that informs the handler on the position, size and title of the console; this handler does not take window parameters from the subsequent `DosPacket` send for opening a file, see also section **??**.

`dp_Arg2` is a copy of the `dol_Startup` element of the `DosList` structure, see chapter **??**. It is used to configure the properties of the handler. The type that is placed here is depends on the mountlist. While its use is handler specific, it is typically, but not necessarily, a BPTR to a `FileSysStartupMsg` structure for file systems. Other possibilities for `dol_Startup` are a BPTR to a BSTR or an integer. What exactly the handler will receive depends on the mountlist and is discussed in more detail in section **??** and section **??**. There is no algorithm in the `Mount` command nor in the handlers that checks whether the type deposited in `dol_Startup` matches the expectations of the handler.

`dp_Arg3` is a BPTR to the `DosList` structure through which the handler was identified and which the handler may modify according to its needs.

`dp_Arg4` may be set by the handler when replying to the startup packet; it is initialized to `NULL` by the *dos.library*. If it remains `NULL`, then `GetDeviceProc()` delivers in `dvp_Port` the process port `pr_MsgPort` of the handler as destination port for packets, see section **??**. A handler may instruct the *dos.library*, however, to send packets to an alternative port by providing its pointer in `dp_Arg4`, which is then copied into `dvp_Port` instead. Thus, this argument is used as an optional *output value* rather than an input argument.

During startup, the handler may or may not initialize `dol_Task`. `GetDeviceProc()` *does not* initialize it and leaves at `NULL`. A file system process would typically handle multiple files by the same process. To ensure that the *dos.library* sends all requests to the process just started, the file system places a pointer to a `MsgPort` in `dol_Task` of the `DosList` structure received in `dp_Arg3`. This is typically, but not necessarily, the `pr_MsgPort` of the handler process; if it is not, the alternative port shall also be provided in `dp_Arg4`[3]. The Fast File System and file systems in general initialize `dol_Task` with their process port.

[3] Unlike what [**?**] claims, `dp_Arg4` defines `dvp_Port` during file system startup even if also `dol_Task` of the `DosList` structure is populated.

A handler such as the CON-Handler requires a separate process for each window it manages. In such a case, `dol_Task` remains `NULL` and `GetDeviceProc()` will initiate a new process for each path requesting the handler. This does *not* imply that the handler process only receives a single request to open a file in its lifetime, though. Any attempt to open a file named "`*`" or a using a file name relative to the (pseudo-) device `CONSOLE` will create a request through the port stored in `pr_ConsoleTask` of the initiating process *without* creating a new instance of a handler. `CONSOLE:` and "`*`" are not exclusive to the CON-Handler. They (maybe surprisingly) apply to any handler whose port is stored in `pr_ConsoleTask`.

Many (non file-system) handlers leave `dol_Task` uninitialized, though some exceptions exist, e.g. the Queue-Handler serves all requests from a single process to allow inter-process communication. Concluding, handlers decide whether they require a new process on an attempt to access them through leaving `dol_Task` uninitialized.

Once the handler or file system initiated itself from the startup packet, this packet shall be replied. If startup failed, the primary result code shall be `DOSFALSE` and the secondary result code shall be an error code suitable for reporting through `IoErr()`, see section **??** for a list of common error codes. Then, on error, the handler shall release all resources acquired so far and terminate.

On success, the main processing loop of the handler is entered processing requests through `DosPackets` it receives through its port(s).

13.1.3 Handler Main Processing Loop

Once started up, handlers or file systems shall wait for incoming packets. The main loop of a handler then checks its own process port, or all ports it provided for incoming packets. Chapter **??** provides information on all packets documented in AmigaDOS, though third-party handlers may implement additional packet types.

When opening files through the packets documented in section **??** and following, the handler or file system receives a BPTR to a `FileHandle` structure whose `fh_Arg1` element may be initialized to serve as an identifier of the file and associated resources. It is delivered back to the handler on all further operations on the file, such as reading from or writing to it. At this point, the handler may also update the `fh_Type` element of the file handle to have packets related to the file delivered to an alternative port. Section **??** lists all packets interacting with files.

Unlike file handles, locks and the `FileLock` structure shall be build by the file system itself when receiving one of the packets in section **??** and following, e.g. when locking a file system object. The `fl_Task` element of this structure, in particular, shall be initialized to point to a `MsgPort` clients will contact for interacting with the file system on a lock. While this is typically the `pr_MsgPort` of the file system process, it may also provide a custom port instead. Section **??** lists all packet types that interact with locks.

The following code is a sketch of a (non-filesystem) handler implementation, expecting linkage with the assembler code from section **??**:

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

#ifndef ACTION_FORCE
# define ACTION_FORCE 2001
#endif

#ifndef ACTION_STACK
# define ACTION_STACK 2002
#endif
```

```
#ifndef ACTION_QUEUE
# define ACTION_QUEUE 2003
#endif

struct ExecBase   *SysBase;
struct DosLibrary *DOSBase;

LONG __asm __saveds main(register __a0 struct DosPacket *pkt)
{
    struct Process *proc;
    struct Message *msg;
    const UBYTE *path;
    ULONG startup;
    struct DosList *dlist;
    LONG error = 0;
    BOOL open  = 0;

    SysBase = *((struct ExecBase **)(4L));
    proc    = (struct Process *)FindTask(NULL);
    /*
    ** if NULL, this was a C startup, retrieve
    ** the packet manually
    */
    if (pkt == NULL) {
      /* Wait and retrieve the startup message */
      WaitPort(&proc->pr_MsgPort);
      msg = GetMsg(&proc->pr_MsgPort);
      pkt = (struct DosPacket *)msg->mn_Node.ln_Name;
    }
    path    = (const UBYTE *)BADDR(pkt->dp_Arg1);
    startup = pkt->dp_Arg2;
    dlist   = (struct DosList *)BADDR(pkt->dp_Arg3);
    DOSBase = (struct DosLibrary *)OpenLibrary("dos.library",37);

    if (DOSBase == NULL)
      error = ERROR_INVALID_RESIDENT_LIBRARY;

    if (error) {
      /* Reply packet manually - dos did not open */
      pkt->dp_Res1 = DOSFALSE;
      pkt->dp_Res2 = error;
      PutMsg(pkt->dp_Port,pkt->dp_Link);
      return;
    }

    /*
    ** Potentially initialize dol_Task
    ** Uncomment for single-instance handlers.
    ** dlist->dol_Task = &proc->pr_MsgPort;
    */
    ReplyPkt(pkt,DOSTRUE,0);
```

```
    /* main program loop */
    do {
      LONG res1 = DOSFALSE;
      LONG res2 = ERROR_ACTION_NOT_KNOWN;
      WaitPort(&proc->pr_MsgPort);
      msg = GetMsg(&proc->pr_MsgPort);
      pkt = (struct DosPacket *)(msg->mn_Node.ln_Name);
      switch(pkt->dp_Type) {
      case ACTION_FINDINPUT:
      case ACTION_FINDUPDATE:
      case ACTION_FINDOUTPUT:
        open++;
        res1 = DOSTRUE;
        res2 = 0;
        break;
      case ACTION_END:
        open--;
        break;
      case ACTION_READ:
        res1 = -1; /* EOF */
        res2 = 0;
        break;
      case ACTION_WRITE:
        res1 = pkt->dp_Arg3;
        res2 = 0;
        break;
      case ACTION_SEEK:
      case ACTION_SET_FILE_SIZE:
      case ACTION_STACK:
      case ACTION_QUEUE:
      case ACTION_FORCE:
        res1 = -1;
        break;
      default:
        break;
      }
      ReplyPkt(pkt,res1,res2);
    } while(open);

    CloseLibrary((struct Library *)DOSBase);

}
```

This example handler first checks whether it received a startup packet through the BCPL binder, and if not, retrieves it from its process port. It then allocates its resources, in this case it only opens the *dos.library*. If that fails, the startup packet is replied with an error code, though without having the library available, this is a manual step. Otherwise, it is replied indicating success.

The main processing loop follows: The handler receives packets from this process port, and checks for the type. For each request to open a file, a use counter is incremented, and for each request to close one, the counter is decremented. In this simple example, no other activity is performed as this dummy handler serves no purpose. Read and write requests are answered with an `EOF` condition or success indicating that all bytes had been written.

The remaining cases implement error handling: A handler or file system receiving a packet it does not implement shall set the dp_Res2 element of the packet to ERROR_ACTION_NOT_KNOWN. The primary result code dp_Res1 shall be set for non-implemented packets according to the packet type as shown in the following table:

Table 13.2: Primary Result Code for Unimplemented Packets

dp_Type	dp_Res1
ACTION_READ	-1
ACTION_WRITE	-1
ACTION_SEEK	-1
ACTION_SET_FILE_SIZE	-1
ACTION_STACK	-1
ACTION_QUEUE	-1
ACTION_FORCE	-1
all others	0

The packet shall then be replied. This ensures that clients of the handler or file system will receive a result that is an indication of an error.

Once the use counter reaches zero again, the handler releases all its resources and terminates processing. The handler shutdown is discussed in more detail in section **??**.

13.1.4 Handler Shutdown

A handler that does not initialize the dol_Task element of its DosList structure should keep a counter that is incremented for each object it creates, and decremented for each object deleted. For example, if the handler supports opening files, then the initialization of each file handle should increment the counter, and each file handle closed through ACTION_END should decrement the counter. Similarly, if locks or notification requests are supported, every lock or request created should increment it, and every lock unlocked and every request canceled should decrement it. Once the use counter reaches 0, the handler process should die by releasing all of its resources and falling off its main function. This ensures that AmigaDOS is not congested by creating more and more processes of the same handler that, effectively, cannot be contacted anymore because its process port is not referenced anymore.

A typical example for such a handler is the CON-Handler that requires a new process for each window opened. Even though there is only one window per console, it can be referenced by multiple files as the console can also be reached through "*" and "CONSOLE:". The window will be closed when each of these files had been closed[4].

File systems such as the FFS, however, typically *do* initialize dol_Task and thus can be reached even if all files, locks or notification requests on the volumes they manage have been released. Thus, in addition to tracking these resources, file systems should check for incoming packets of the type ACTION_DIE. If the file system is aware that any of its resources are still in use, e.g. some files or locks are still open, ACTION_DIE shall fail. Otherwise, the file system may reply to this packet with success and should attempt to shutdown.

The pr_MsgPort or any other ports of such a handler can still contain packets that have not yet been worked on after ACTION_DIE has been received and replied. In order to avoid a deadlock, the packets pending in the input queue still need to be replied, for example using the default return codes from table **??** in section **??**.

Despite such precautions, ACTION_DIE cannot be implemented in a fully reliable way as some of the MsgPort(s) of the file system could still be cached by client programs even without having access to any of its resources. Section **??** contains further details on ACTION_DIE.

[4]This is a simplification, ignoring AUTO and WAIT parameters, see section **??** for details

13.2 The CON-Handler

The CON-Handler implements the console of AmigaDOS, which hosts for example the shell. It not only serves the graphical console based on an intuition window, but also consoles on the serial port or other devices. The AUX-Handler is only a minimal disk-based wrapper that locates the CON-Handler in the Kickstart and launches it with a suitable startup packet. The CON-Handler therefore implements the `CON`, `RAW` and `AUX` devices.

Even though the AmigaDOS Shell is the most prominent user of the console, the two components are otherwise completely separate. The Shell can be run on any AmigaDOS device — and in fact is, when executing scripts — and the console can be used by any other program, for example by `Ed`.

Quite similar to file systems, the CON-Handler does not implement all of its functionality itself. It rather depends on services of exec devices. For the graphical console, it creates an intuition window and initializes the *console.device* to run within it. For the serial console, i.e. `AUX`, it operates on top of the *serial.device*, but can be made to use any other interface device with proper mount parameters, see also section **??** for examples how to mount a customized `AUX` handler.

13.2.1 CON-Handler Path for Graphical Consoles

The path through which a stream to the `CON` and `RAW` devices is opened determines the size and position of the window, and defines features of the console that runs in this window; for the `AUX` device, it defines connection parameters such as the baud rate and the parity.

This section discusses the parameters for the graphical console, section **??** lists the options for `AUX`. Angle brackets "`< >`" name the options for the purpose of referencing them, they are *not* part of the actual parameter string supplied to the handler.

```
CON:<left>/<top>/<width>/<height>/<title>[/<options>]
```

The `<left>` parameter determines the left edge of the window within which the console appears; it is measured in pixels. If this parameter is not present or negative, intuition is instructed to pick a default, which itself defaults to a window position under the mouse pointer.

The `<top>` parameter is the top edge of the window, also measured in pixels. If this parameter is not present or negative, intuition is instructed to pick a default.

The `<width>` parameter is the width of the window, including all window decorations. If this is not present or negative, the handler instructs intuition to pick a default, which is the width of the screen. If no parameters are present at all, i.e. the path is just `CON` or `RAW` without any parameters, the width of the window is set to 640 pixels for legacy reasons.

The `<height>` parameter is the height of the window, including all window decorations. If this parameter is not present, the window height is set to 100 pixels for legacy reasons. If this parameter is negative, the window height will be the height of the screen. Width and height are not handled symmetrically, this is intentional.

The `<title>` parameter is the string that is put into the drag bar of the window. To insert the forward slash ("`/`") into the string, it is escaped with a backslash, i.e. "`\/`" creates a single forwards slash that does not act as a parameter separator[5], the backslash is escaped by itself, i.e. "`\\`" creates a single backslash in the title.

All following path components configure options for the graphical console; multiple of these parameters may be combined, separated by forwards slashes ("`/`") from each other:

`CLOSE` adds a close gadget to the window. What happens if the user presses on this gadget depends on the mode the console is operating in, see section **??**, whether any file handles are open to the console,

[5] Given the overall syntax of AmigaDOS, the asterisk $\star$ would have been a more logical choice.

and whether any read requests are pending. In particular, the options AUTO and WAIT also impact how this gadget works.

NOCLOSE is the negative form of the above option and removes a potentially added close gadget from the window.

AUTO delays opening the window to the point where an attempt is made to read data from the console or print text into it. If neither read nor write requests are pending, the close gadget will also close the window without queuing an end-of-file condition to read requests; the window will pop open again as soon as read or write requests are received. Unfortunately, an ACTION_DISK_INFO will lock the window open, thus disabling this parameter, see section **??** for alternatives to this packet or how to regain the AUTO feature. AUTO does not work for the RAW device or in the raw mode for reasons explained below.

WAIT waits for the user to close the window. With this option present, the window stays open even though the last file handle has been closed, and it will only close when explicitly requested by pressing the close gadget. This option does not work for the RAW device or in raw mode either. The purpose of this option is to leave the output of a program in a window visible even after the program creating the window terminated.

If read requests are waiting on the console, it depends on the console mode, see section **??**, how the console reacts on the close gadget, but it will *not* close the window by itself. If the console is operating in raw mode and a process has requested to receive window close events with the CSI 11{ sequence, then a CSI raw event will be send to the next reading stream, thus indicating to an application — such as an editor — the request to close the window. The raw event interface for requesting and delivering input events in the form of CSI sequences is *not* implemented by the CON-Handler, but rather inherited from the *console.device* and therefore documented in [**?**].

That the options AUTO and WAIT do not work for RAW is also a side-effect of the above: These options require interpretation of the raw events send by the *console.device*, but in raw mode the console only forwards all received data to its clients directly.

In the cooked and medium mode, an EOF condition is generated if the window close gadget is pressed and the console is still in use by at least one file handle. It is again up to the receiving process how to react on an end of file. If the process then closes its file handle to the console, and this handle was the last open handle, the CON-Handler will either shut down directly and thus as part of its shutdown code will also close the window, or will stay open if the WAIT option is present and wait for the user to press the close gadget — probably once more.

If the console is not in raw mode, the keyboard combination Ctrl+\ also triggers an EOF condition. As for the close gadget, this keyboard combination is not functional in raw mode and there sends the ASCII FS control character, i.e. the code 0x1c. The same keyboard combination also closes a console whose file handles have all been closed and which only stays open due to the WAIT option.

SMART enforces smart refresh of the window. If this is enabled, more RAM is required for saving console graphics hidden behind other windows, and it is then not necessary to re-render the contents of the console if it is moved upfront another window. In most cases, rendering speed improves only insignificantly, but at the price of a larger RAM footprint.

SIMPLE enforces simple refresh of the window. If a partially hidden console window is moved upfront and made visible again, its contents will be reprinted. This does usually not cost a lot of time, saves RAM and is also the default.

INACTIVE prevents that the window is receiving the input focus automatically when it is opened. The user has to click into it to activate it and thus to be able to type into it.

BACKDROP instructs intuition to place the window behind all other windows on a screen. This works best in conjunction with NOBORDER, an empty window title, the NOSIZE, NODRAG and NODEPTH options and a console redirected to a public screen. If the window is made as large as the screen, this results in a full-screen console.

NOBORDER removes the window decorations; if a title is present, the drag bar will still appear. Thus, this option is ideally combined with an empty title string.

NOSIZE removes the size gadget of the console on the bottom right edge of the window and thus creates a fixed-size window.

NODRAG removes the dragbar at the top of the window and makes the window non-movable. A potential application is to create a non-movable console as screen background.

NODEPTH removes the depth arrangement gadget from the top right border of the window such that the console window can no longer be depth arranged.

WINDOW instructs the console to hijack an already open window and run the console within it. This window will also be closed if the console closes, i.e. the console owns the window from this point on and controls its life-time. The address of the Window structure is provided in hexadecimal behind the parameter, optionally separated by spaces, e.g. WINDOW 200AFC0. An optional "0x" string may appear upfront the hexadecimal window address. This parameter replaces a legacy startup mechanism by which the console could also be placed into an already opened window, see also **??**. The window structure is documented in [**?**] and defined in intuition/intuition.h.

SCREEN provides a name of a public screen on which the console shall appear. The public screen name follows the path component, optionally separated by spaces, e.g. SCREEN myProgram.1. If the screen name is *, then the front-most public screen will be used as host for the console.

ALT provides an alternative window position and dimension to which the window can be toggled by its top-right zoom gadget. The alternative window placement provides left edge, top edge, width and height, similar to the main window placement, though numerical arguments are separated by comma (",") and not by the forwards slash. The alternative position and size shall follow directly behind the ALT keyword, for example ALT 0,0,320,200.

ICONIFY equips the window with an iconification gadget. This requires that the ConCip program is running as it loads the console icon. If the iconification gadget is pressed, this icon will appear on the Workbench, and the window will disappear. The window will be forced open when a incoming read or write request requires interaction with the console, or when the icon on the Workbench is double-clicked. Similar to the AUTO option, windows will loose the ability to become iconified if a program requests the window pointer through ACTION_DISK_INFO, see again section **??** for details.

Similar to AUTO, ICONIFY also works not quite as expected in raw mode. As reaction to a click on the iconification gadget, the *console.device* will send a raw event CSI sequence the CON-Handler will not interpret, but instead forward blindly to its clients. Thus, iconification is to be performed by the process reading from a console operating in raw mode, and not by the console itself.

13.2.2 CON-Handler Path for Serial Consoles

The following section covers the AUX device which is also managed by the CON-Handler in AmigaDOS version 47. In earlier versions, the device was operated by an independent handler implemented in BCPL that was phased out. Therefore, the description in this section only applies to AmigaDOS 47 and beyond.

If the console is a serial console, e.g. mounted as AUX-Handler, another set of parameters in its path becomes available that configures the serial connection; if the console is run on any other than the *serial.device* because it was mounted with custom parameters as described in section **??**, the underlying exec device shall support the SDCMD_SETPARAMS command through which the parameters in the path are forwarded.

Parameters for the handler come from three sources: First, from the Serial preferences of the system. These default parameters are overridden by the mountlist for AUX, described separately in section **??**. Third, the parameters from the mountlist can be overridden by the path from which AUX: is opened.

The syntax of the path is as follows:

```
AUX:<baud>/<control>[/<options>]
```

baud defines the baud rate of the serial console, e.g. 9600. If this parameter is not provided, it is taken from the BAUD keyword in the mountlist, and if it is not provided there either, the settings come from the Serial preferences, see also section **??**.

control defines the number of data bits, the parity and the number of stop bits, all concatenated into a string of 3 characters. The number of data bits is a digit between "5" and "8" and does not include the parity bit. The parity is either "N" for no parity, "E" for even and "O" for odd parity, or "M" for mark and "S" for space parity. The number of stop bits is either the digit "0", "1" or "2". A rather typical setting is 8N1, i.e. 8 data bits, no parity, one stop bit. This is also the optimal configuration for the *serial.device*. If the control parameter is not present in the path, it is taken — with identical encoding — from the CONTROL parameter of the mountlist, and if it is not present there, from the Serial preferences editor.

As for the graphical console, several optional parameters follow that may be combined:

RAW forces the console into the raw mode by which input characters are not echoed and no line buffering takes place. If AUX is mounted as HANDLER and not as EHANDLER, setting the STARTUP mount parameter to 1 also switches to this mode, but makes all other options in the mountlist unavailable. Section **??** provides more details on these two keywords in the mountlist.

UNIT defines the unit of the device over which the connection shall be made if multiple units exist. The *serial.device* only provides unit 0, but third party handlers may offer additional units. The unit follows the keyword as decimal number, optionally separated by spaces, e.g. UNIT 1. The unit can also be configured through the mountlist UNIT keyword.

AUTO indicates that the serial connection shall not be opened immediately, but only after the first attempt is made to write data to it or to read data from it. This is similar to the AUTO keyword for the graphical console.

WAIT indicates that the serial connection stays open even after every file handle to the console was closed. To shut down the serial connection, an ASCII FS character = Hex 0x1c need to be send by the user. This character is created on the Amiga keyboard with Ctrl+\. This also mirrors the WAIT keyword of the graphical console.

13.2.3 CON-Handler Startup and Mount Parameters

In addition to the parameters communicated in the path by which a console is opened, the console is also configured through the handler startup mechanism and thus by the mountlist. It provides defaults for the serial console, including the line buffer mode described in section **??**.

The mount parameters for CON and RAW are hard-coded into the Kickstart ROM, though the mountlist for AUX is available in the DEVS:DosDrivers directory and can be altered there; users can create custom mountlists for additional devices based on the CON-Handler by specifying L:AUX-Handler either as HANDLER or EHANDLER. All the AUX-Handler does is to collect startup parameters and forward them directly to the CON-Handler, making it customizable.

The following example mountlist creates a serial console on top of a device named duart.device unit 0; in order to supply a custom device and a baud rate, the mount entry needs to be an EHANDLER:

```
EHandler = L:Aux-Handler
Device   = duart.device
Unit     = 0
Baud     = 9600
Control  = "8N1"
```

If mounted with the HANDLER keyword instead, it will operate on top of the serial.device, and the STARTUP value then specifies the console mode, see section **??**. Unfortunately, due to the way how Mount operates, STARTUP is not available for mount lists based on the EHANDLER keyword, and the DEVICE,

UNIT, BAUD and CONTROL parameters are not available with the HANDLER keyword. Section **??** and following provide more information on the keywords.

The GLOBVEC entry in the mountlist is irrelevant, due to a startup code similar to that listed in section **??**. The AUX-Handler can operate both as a C/Assembler or as a BCPL handler.

The handler startup mechanism in general is specified in section **??**. As described there, dp_Arg2 of the startup packet is a copy of the dol_Startup entry of the DosList created from the mountlist. The string from which the CON-Handler receives the path described in sections **??** and **??** is taken from dp_Arg1 of the startup packet, and not — as one might expect — from the packets opening file handles.

The following table specifies the values of dol_Startup the CON-Handler recognizes:

Table 13.3: CON-Handler Startup Code

dol_Startup	Description
< 0	RAW: raw mode, within a window pointed to by dp_Arg1
0	CON: cooked mode, create a new window
1	RAW: raw mode, create a new window
2	CON: medium mode, create a new window
3	RAW: reserved for future extensions
BPTR with bit 30 cleared	raw mode open on a device specified by a FileSysStartupMsg
BPTR with bit 30 set	cooked mode open on a device specified by a FileSysStartupMsg

The values 0 to 3 configure the line buffer mode and correspond to the mode selected by SetMode(), compare also with section **??**. The buffer mode 3 is currently not used and reserved for future extensions. Mode 2 is a new mode introduced in AmigaDOS 47. Its primary purpose is to support the Shell with additional features. It is described along with all other modes in section **??**.

The case of a negative dol_Startup value is a legacy startup mechanism of the console. In this configuration, dp_Arg1 of the startup packet, see section **??**, is not a BPTR to a path as usual, but instead a pointer (not a BPTR) to an intuition Window structure; the CON-Handler then creates a graphical console within this window. It is quite hard to make practical use of this startup mechanism and it is thus discouraged, but [**?**] provides example code how to trigger it. It should be considered deprecated. A better mechanism to run a console in an already opened window is by the WINDOW argument described in section **??**.

In all other cases, dol_Startup is a BPTR to a FileSysStartupMsg described in more detail in section **??**. The fssm_Device and fssm_Unit elements of this structure provide a device and unit the console shall run on. While the CON device does not use this startup mechanism, the CON-Handler receives such a startup message through the AUX-Handler if it is mounted as EHANDLER, and it then contains the DEVICE and UNIT parameters from its mountlist. Thus, a console can be run on any kind of device provided it supports CMD_WRITE to print text, CMD_READ to receive keystrokes and SDCMD_SETPARAMS to configure it.

As BPTRs are upshifted by two bits to gain a regular pointer, the otherwise unused bit 30 of the BPTR is used for mode selection. If this bit is 1, the console will be a cooked console with line buffer, and if this bit is 0, the console is raw.

13.2.4 CON-Handler Buffer Modes

In addition to the window or serial parameters communicated in the path, the console can also be configured through the SetMode() function, see section **??** and table **??** in that section, allowing to alter its properties after opening it. The above function sets the buffer mode of the console, regardless of whether the console runs in a window or over the serial interface. SetMode() allows to convert a CON: window into a RAW: window or create an equivalent of a RAW: handle from an AUX: connection, just that it operates over the serial interface.

The console buffer mode determine how and at which time user input is provided to a process reading from the console. The CON-Handler supports in total three buffer modes.

The buffer mode 0 corresponds to a regular `CON:` window and also to the `AUX:` console. In this so called *cooked mode*, the console echoes every keystroke on the window, and provides elementary line editing functions such as cursor movements. Only when the user presses `Return`, an entire line of data enters the output buffer, and then becomes available for an `ACTION_READ` request. Thus, programs can only read entire lines from the console, they do not receive individual keystrokes.

In the cooked mode, the CON-Handler interprets the following keystrokes:

Table 13.4: Keystrokes Interpreted in Cooked Mode

Keystroke	Function
`Cursor left`	Move left one character
`Cursor right`	Move right one character
`Cursor up`	Do nothing*
`Cursor down`	Do nothing*
`Shift+Cursor up`	Do nothing*
`Shift+Cursor down`	Do nothing*
`Shift+Cursor left`	Move to the start of line
`Shift+Cursor right`	Move to the end of line
`Shift+Cursor up`	Move to the start of line*
`Shift+Cursor down`	Erase line*
`TAB`	Inserts a TAB control code*
`Shift+TAB`	Do nothing*
`Return`	Complete line and send it to clients
`Ctrl+A`	Move to the start of line
`Ctrl+B`	Erase entire line
`Ctrl+C`	Sends signal 12
`Ctrl+D`	Sends signal 13
`Ctrl+E`	Sends signal 14
`Ctrl+F`	Sends signal 15
`Ctrl+H`	Identical to Backspace
`Ctrl+J`	Insert a Line Feed
`Ctrl+K`	Kill characters into yank buffer
`Ctrl+M`	Identical to Return
`Ctrl+Q`	Resume output
`Ctrl+R`	Do nothing*
`Ctrl+S`	Suspend output
`Ctrl+U`	Delete beginning of line
`Ctrl+W`	Delete word
`Ctrl+X`	Erase entire line
`Ctrl+Y`	Paste characters from yank buffer
`Ctrl+Z`	Move to end of line
`Ctrl+\`	Signal an end-of-file
`Backspace`	Delete character upfront cursor
`Del`	Delete character under cursor

* Interpreted differently in medium mode, see table **??**

The *yank buffer* keeps characters that have been cut out with `Ctrl+K`. Its contents can be reinserted into the console with `Ctrl+Y`. It operates independently of the clipboard and is local to each console window.

The line feed inserted by Ctrl+J splits the current line at the cursor position and keeps editing the lower line, that is, it inserts a line feed into the output buffer at the cursor position. If finally Return is pressed, then all lines combined, including the line feeds between them, are transmitted to processes reading from the console. In case the Shell is reading from the console, its interpreter splits input at line feed characters and therefore executes the commands on each line separately.

Ctrl+W erases the characters to the left of the cursor position up to the previous blank space, i.e. it erases the word to the left of the cursor. Ctrl+U erases all characters to the left of the cursor position.

Ctrl+S suspends the output and blocks any program from outputting text on the console. Output is resumed with Ctrl+Q.

Ctrl+\ creates a end-of-file condition for reading programs, i.e. the console will respond to pending read requests by returning zero bytes.

The keys Ctrl+C through Ctrl+F are special in the sense as they signal SIGBREAKB_CTRL_C to SIGBREAKB_CTRL_F to the most recent process that printed to the console or requested data from it. This is usually an indicator to the receiving program to interrupt processing or interrupt a script. If the most recent reading and writing processes are different, the corresponding signal will be send twice; a program can also explicitly request receiving signals instead of the most recently reading program by the DosPacket ACTION_CHANGE_SIGNAL, see section **??**.

Unlike other operating systems, AmigaDOS does not have the ability to asynchronously terminate processes; thus, programs should check for the above signals periodically, e.g. by CheckSignal() as described in section **??**, and interrupt their program flow when one of them has been received. By convention, Ctrl+C is supposed to interrupt a program, while Ctrl+D interrupts a shell script and is thus tested by the Shell. Ctrl+E is currently unassigned. Ctrl+F makes the control window of Commodities visible. The pattern matching functions from chapter **??** offer the option check such signals while scanning a directory tree, but the *dos.library* is otherwise ignorant to the signals.

One particular difference between the V47 console and its predecessors is that the console no longer offers a history. Rather, the history functions are now part of the Shell which operates in buffer mode 2 described below. The history is not the only change, some keystrokes operate differently in this *medium mode*. As for the history, they request a particular function from the shell, or any other program that operates the console in this mode, see below.

The buffer mode 1 corresponds to a RAW: window and is denoted the *raw mode*. In this mode, the console reports every keystroke individually to its clients, without interpreting them. Read requests pending in the console are satisfied for every single keystroke, even if they request more bytes than generated by the keystroke. Keystrokes are not even echoed on the console, this is up to the program receiving them. Thus, for example, cursor movements are reported as multi-byte control sequences, but do not actually move the cursor. The only exception are the keystrokes Ctrl+C to Ctrl+F which are always interpreted and send break signals instead of their ASCII codes. All other keys remain uninterpreted, even Ctrl+\ does *not* create an EOF condition in raw mode, but rather generates the control code 0x1c (ASCII FS).

The control sequences are either single bytes from the ASCII control character set between 0x00 and 0x1f, or ECMA-48 control sequences [**?**] with some additions from AmigaDOS, starting with a CSI *control sequence introducer* character, code 0x9b. The sequences are generated by the *console.device*, or to be more specific, by the keymap that is currently active. [**?**] and [**?**] provide details and specify the sequences. The *console.device* is also able to generate CSI sequences when it receives intuition events such as resizing the window or attempting to close the window. In order to receive such events, they need to be requested by CSI <n> {, and they are unsubscribed by CSI <n> }, where <n> indicates the event type, for details see again [**?**]. It is noteworthy that a console starting in raw mode *does not* subscribe to the window close event, though consoles opening in cooked mode do, in order to react on the close gadget. The CON-Handler only forwards such events from the *console.device* or any other device it is connected to if operating in the raw mode, but it interprets them in the cooked and medium mode, and, for that reason, cannot react on close window events or any other events in raw mode.

Similar to creating control sequences, the *console.device* also interprets control sequences and reacts accordingly, e.g. by moving the cursor or erasing its window. Such functions are *not* implemented by the CON-Handler which only passively forwards them to the device to which it is connected. Thus, in particular, if the console is connected to a terminal via a serial connection, then it is up to the connected terminal to interpret them.

The buffer mode 2 is not available under a device name, even though one could in principle create a suitable mountlist for it. It is called the *medium mode*. The mode is reserved for shells and forwards TAB expansion requests and history functions to them. While the console continues to provide line editing features for the keys listed in table **??**, some of the keys marked with an asterisk in this table are in the medium mode reported immediately to programs reading from the console without waiting for the `Return` key to be pressed. They create *TAB Report* CSI sequences that are interpreted by the Shell and trigger there functions such as expanding a pattern into a list of files, or iterating through the command history.

A TAB Report sequence looks as follows:

```
CSI <m>;<s>;<c>U <line>
```

Here `CSI` is again the *control sequence introducer* of the byte value `0x9b`. The `<m>` value is a decimal number that identifies the keystroke and also the function the Shell is requested to perform. Such functions browse in the history or expand a file name pattern. The Shell re-inserts the result of such a function back into the console with an `ACTION_FORCE` packet, see section **??**.

The parameters `<s>` and `<c>` provide the length of the current line input buffer of the console the user is editing, and the (1-based, i.e. cursor at the left edge corresponds to `<c>` = 1) cursor position within this buffer. Both `<s>` and `<c>` are also encoded as ASCII decimal numbers.

The `U` is a literal character and identifies this sequence as TAB Report. `<line>` is a copy of the current line that is being edited by the user. For many sequences, this input is used as a template the Shell interprets as part of the requested action, e.g. for a TAB expansion: there, the command line argument within which the cursor is positioned, as reported by the `<c>` parameter of the CSI sequence, is used to generate a wildcard the Shell uses to find possible candidates for the expansion of the file name template the cursor is located in.

The keystrokes that trigger such TAB Report sequences, along with their purpose, are as follows:

Table 13.5: TAB Report CSI Sequences

Keystroke	Sequence
`Cursor up`	`<m>`=2, move upwards in the history
`Cursor down`	`<m>`=3, move downwards in the history
`Shift+Cursor up`	`<m>`=4, search history upwards
`Shift+Cursor down`	`<m>`=5, search history downwards
`Ctrl+R`	`<m>`=6, search history upwards, partial pattern
`Ctrl+B`	`<m>`=10, rewind history
`TAB`	`<m>`=12, iterate forwards through expansion
`Shift+TAB`	`<m>`=13, iterate backwards through expansion

The only difference between the cases `<m>`=6 and `<m>`=4 is that the former only uses the line input buffer up to the cursor position as search pattern and ignores everything beyond it. All remaining values for `<m>` are currently reserved, even though some third-party implementations assign meanings to them.

Thus, the flow of operations if a user presses the `Cursor up` key in the console is as follows: First, the console delivers a `CSI U` TAB Report sequence to the Shell. The Shell will then check its history for the previously issued command, and will insert the line from the history with `ACTION_FORCE` back into the console. The latter is a special `DosPacket` that emulates keystrokes, i.e. the Shell *types* the previous command from the history into the console. The advantage of this construction is that the command history

becomes available to programs operating in the Shell, and can be retrieved or modified from there, see section **??**.

Before executing programs, the Shell switches the console back to the regular cooked mode for backwards compatibility, and re-enables the medium mode as soon as it regains control[6].

13.3 The Port-Handler

The Port-Handler is responsible for the PAR, SER and PRT devices and thus represents the Amiga hardware interfaces as AmigaDOS devices. Clearly, PAR interfaces to the *parallel.device*, PRT to the *printer.device* and SER to the *serial.device*. The difference between SER and AUX is that the former processes raw streams of bytes transmitted over a serial connection, whereas AUX provides line buffering and line editing capabilities and thus implements a console[7].

13.3.1 Port-Handler Path

Access to the Port-Handler is configurable through the mountlist and the path through which it is opened. In this section, configuration through the path is discussed. While the Port-Handler was not configurable in AmigaDOS versions 40 and below, AmigaDOS 45 upgraded it. The path, depending on how the handler is mounted, looks as follows:

```
SER:[[BAUD=]<baud>/][[CONTROL=]<control>/][NOWAIT/][UNIT=<n>]
PAR:[UNIT=<n>]
PRT:[TRANSPARENT/][RAW/][UNIT=<n>]
```

The brackets are not part of the path and shall not be included. Square brackets indicate optional elements, angle brackets parameters that shall be substituted by a value.

The Port-Handler parses the following parameters from its path:

BAUD sets the baud rate, which is only relevant for serial connections and thus SER:. It specifies the number of bits per second transmitted over a serial connection. If this parameter is not present, the default from the mountlist is used, see section **??**, and if not present there, the Serial preferences provide the baud rate. Optionally, this parameter may be introduced with the keyword BAUD which shall be separated from the decimal baud rate by spaces or an equals sign, e.g. BAUD 9600. Without the keyword, it shall be the first parameter in the path.

CONTROL specifies additional parameters of the serial connection such as number of data bits, parity and number of stop bits, represented as 3 letter string. It is also only relevant for the SER device. If this parameter is not present, the CONTROL parameter of the mountlist provides a control string of identical encoding, and if not present there, the settings from the Serial preferences are used.

The argument of CONTROL follows the same convention as the one for the AUX terminal. The first digit between "5" and "8", specifies the number of data bits; the second letter is the parity, which is either N, E, O, M or S, indicating no parity, even or odd parity, or mark or space parity. The last digit is the number of stop bits and is a number between "0" and "2". A typical control string is 8N1 indicating 8 data bits, no parity and 1 stop bit, which is also the optimal setting for the *serial.device*.

The CONTROL parameter is either matched by position as second component of the path, or by its keyword, which shall be separated by a blank space or an equals-sign from its value, e.g. CONTROL 8N1.

TRANSPARENT is a Boolean switch only relevant to the PRT device. If present, it indicates that CSI sequences written over this device will not be translated to the printer, but will be send untranslated (as-is).

[6]This has the side effect that the Shell can also operate in a console opened as RAW:.

[7]Even in the raw mode, AUX is different from SER; for example, the former creates a signal on Ctrl+C, the latter sends the byte 0x03.

By default, the PRT device accepts the ECMA-48 (see [**?**]) and AmigaDOS defined CSI sequences (see [**?**]) that are device independent and translates them to printer specific sequences by the *printer.device*. With this option, PRT: expects raw control sequences that are printer and model specific. This option matches by keyword, its position in the path is irrelevant.

RAW is a Boolean switch that, if present, disables translation of the AmigaDOS newline character 0x0a to 0x0a 0x0d pairs, i.e. a line feed followed by carriage return. This switch only applies to the PRT device and is not used otherwise. It matches by keyword, its position is irrelevant.

UNIT takes a numeric argument and provides the unit of the device that shall be used. This makes most sense for PRT: as the *printer.device* supports multiple units. While the ROM-based *serial* or *parallel.device* only support a single unit each, the Port-Handler can also be mounted on a third-party devices provided through a custom mountlist, see section **??** for details. The unit number is represented as decimal number separated from the keyword by spaces or an equals-sign, e.g. UNIT 1. This option applies to all devices mounted on the Port-Handler, and matches by keyword.

NOWAIT is Boolean switch that, if present, avoids blocking when reading from a connection, and only applies to SER:. If no data is available at the serial port, a connection configured with NOWAIT will report an end-of-file condition. Otherwise, the stream blocks until data becomes available. This option matches by keyword.

Another possibility to implement non-blocking reading from the serial port is to use WaitForChar(), see section **??**. This function blocks until either the specified time has elapsed, or input is available on the port. Support for this function in the Port-Handler was introduced in AmigaDOS 45.

13.3.2 Port-Handler Startup and Mount Parameters

The Port-Handler is also configurable through its mountlist, though mount parameters for PAR, SER and PRT are hard-coded into the Kickstart ROM and thus cannot be customized. However, users may create custom mountlists and mount the handler under a custom device name.

As most handlers, the Port-Handler is parametrized through dol_Startup, transmitted in dp_Arg2 in the handler startup packet, see also section **??**. The following values are recognized:

Table 13.6: Port Handler Startup Code

dol_Startup	**Description**
0	SER: serial input and output through the serial.device
1	PAR: parallel output through the parallel.device
2	PRT: printer output through the printer.device
BPTR	serial input/output with parameters from the FileSysStartupMsg

The values 0 to 2 represent the handlers SER, PAR and PRT mounted by the Kickstart, and are supported by all AmigaDOS versions. They can be recreated by a mountlist as follows:

```
Handler = L:Port-Handler
Startup = <n>
GlobVec = 0
```

where <n> is one of the values above. Unfortunately, the STARTUP keyword is mutually exclusive to a device and unit specification, and thus the Port-Handler in the above configuration always uses the unit 0 of the default devices according to table **??**, unless the unit is set explicitly in the path, see section **??**. As seen in the above mountlist, the Port-Handler relies for legacy reasons on the BCPL startup mechanism and *cannot* be run as C/Assembler handler, even though it is nowadays implemented in C.

The last option in table **??** corresponds to a custom mountlist based on the EHANDLER keyword, see also section **??**. This requires AmigaDOS version 45 or better. The DEVICE and UNIT keywords then provide

the name of the device and the unit the handler runs on. If the mountlist includes in addition a BAUD value, it is used as default value for the baud rate which can be overridden by the BAUD parameter in the path described in section **??**. The CONTROL parameter in the mountlist, if present, controls the serial settings. Its encoding is identical to the CONTROL parameter in the path, see section **??**.

The following mountlist creates a device similar to SER that operates on top of the (third-party provided) duart.device unit 1, and configures it by default to 19200 baud, 8 bits, no parity and 1 stop bit:

```
EHandler = L:Port-Handler
Device   = duart.device
Unit     = 1
Baud     = 19200
Control  = "8N1"
GlobVec  = 0
```

In the absence of additional sources of information, such a mountlist creates a SER style device and assumes the device to accept the command set of the *serial.device*. However, the Port-Handler uses the AmigaDOS 43 extension NSCMD_DEVICEQUERY to identify devices that operate similar to the *parallel.device* or *printer.device* and then adjusts its mode of operation accordingly. In such a case, the BAUD and CONTROL parameters in the mountlist are irrelevant and ignored, and the device becomes a lookalike of either PAR or PRT, which also accepts those parameters listed in **??** in their path that correspond to the nature of the device.

13.4 The Queue-Handler

The Queue-Handler provides inter-process communication on the basis of AmigaDOS file handles. It is used by the Shell to implement pipes; they collect the output of one command and provide it as input to another, without requiring to store the output in a temporary file. The reading end of a pipe blocks as long as no data becomes available at the writing end, and the writing end blocks if the reading end is not able to consume data fast enough. If the reading end closes the pipe, the Queue-Handler attempts to abort the writing end by sending a SIGBREAKF_CTRL_C signal to it.

Even though the Queue-Handler is disk-based, AmigaDOS 47 already mounts it during its startup under the name PIPE. While a handler of the same name and a similar purpose already existed in prior versions of AmigaDOS, it had to be mounted manually and lacked some of the features of the most recent version.

13.4.1 Queue-Handler Path

Pipes are created and identified by a unique name; reading and writing ends of the same name are connected together. The name itself does not matter; the Shell constructs a pipe name from the process indicator and a second unique number that is incremented for each pipe used. The Queue handler can be either opened in MODE_OLDFILE, which represents the reading end of the pipe, or MODE_NEWFILE, which corresponds to the writing end. The third mode, MODE_READWRITE, is not supported. For Open() and its modes see section **??**.

The empty file name establishes the special case of the anonymous pipe. The empty name represents the pipe that is already connected to the standard input or standard output of the process contacting the Queue-Handler. If the open mode is MODE_OLDFILE, it will attempt to reuse the pipe connected to the standard input of the process, or on MODE_NEWFILE the pipe connected to the standard output. If the file handle in the standard input or standard output does not belong to the Queue-Handler, opening the anonymous pipe fails. Its purpose is in Shell command lines such as

```
Join A B to PIPE: | More
```

where the writing end of the pipe is explicitly addressed as anonymous pipe as `PIPE:`. The Shell connected the standard output of the `Join` command to a named pipe created for the purpose of the `|` operator, see also section **??**, and the file name `PIPE:` connects to this named pipe. This construction is necessary here because the command line syntax of `Join` does not offer to write to its output stream as prepared by the Shell but only to an explicit file name. Anonymous pipes were introduced in AmigaDOS 45.

The path of the Queue-Handler may include options configuring it; the complete specification is as follows:

```
PIPE:name[/quantumsize[/buffercount]]
```

The `name` is the name of the pipe and serves as its identifier, see above.

The `quantumsize` is the size of a block the Queue-Handler uses to buffer data between the writing and the reading end. It is encoded as an ASCII decimal number and measured in bytes. Written data is is queued up until either a buffer is full or the writing end is closed, and the full block is then made available to the reading end. This minimizes the communication overhead as the reading process does not need to wake up for every single byte. The default quantum size is 1024 bytes.

The `buffercount` configures the number of buffers the Queue-Handler makes available to the writing end until it blocks. It is also encoded as decimal number. If the configured number of buffers, all sized according to the `quantumsize`, are full without being retrieved by the reading end, the writing end blocks. This avoids that a fast writing process floods the memory of the system. The special value of 0 indicates that system memory may be filled up to a safety margin of 64K. The default buffer count is 8, i.e. for a quantum size of 1024 bytes at most 8K of data can be waiting in a pipe.

13.4.2 Queue-Handler Startup and Mount Parameters

The Queue-Handler can also be configured through a mountlist, though as it is already mounted by the ROM, the name of a custom mounted device cannot be `PIPE`. The mount parameters used by the Kickstart correspond to the following mountlist:

```
Handler         = L:Queue-Handler
Startup         = 0
GlobVec         = -1
```

which mounts the Queue-Handler with its default parameter described in **??**. The handler is implemented in C and thus does not require a Global Vector, thus `GlobVec` shall be set to -1. To customize the Queue-Handler, a mountlist such as the following can be used:

```
EHandler        = L:Queue-Handler
Device          = no.device
BlockSize       = 1024
Buffers         = 8
GlobVec         = -1
```

The `BLOCKSIZE` keyword sets the quantum size, that is the size of a single buffer in bytes, and the `BUFFERS` keyword the number of buffers. The argument of the `DEVICE` parameter is irrelevant, but shall be present as the `Mount` command requires it.

The above mountlist creates a `FileSysStartupMsg` as specified in section **??** and places it into the `dol_Startup` element in its `DosList` structure. The device and unit found there are irrelevant, though the environment vector in `fssm_Environ` discussed in section **??** contains two relevant entries:

`de_SizeBlock` set the quantum size, and thus corresponds to the default of the first optional argument in the path and the `BLOCKSIZE` parameter in the mountlist.

`de_NumBuffers` sets the number of such buffers for each pipe and corresponds to the second optional argument of the path or the `BUFFERS` parameter in the mountlist.

13.5 The RAM-Handler

The RAM-Handler is a ROM-based file system that places its data in available RAM of the system. It implements almost all of the file system packets listed in chapter **??**, with the exception of packets specific to interactive handlers in section **??** and `ACTION_SET_OWNER`, see **??**.

Starting with version 47, the RAM-Handler provides one extended feature, namely external links, also explained in section **??**. Such a link points to a target outside of the RAM-Disk; while external links are initially empty, they will be filled with the contents of the link target as soon as they are accessed. The RAM-Handler then copies the link target into RAM, making it accessible under the link origin, or an object within the link origin. The link source is from that point on independent from the link target and can be modified without affecting the linked object.

This feature is used within AmigaDOS for the `ENV` directory containing the currently active system preferences; the `ENV` assign points to it. This allows to save RAM by copying only those files to `ENV:` that are actually accessed. A separate handler is not necessary.

The RAM-Handler does not interpret any arguments in its startup packet and cannot be configured through it. The `STARTUP` value prepared by AmigaDOS is 0 and not interpreted by the RAM-Handler. As the RAM-Handler is not represented as file nor registered as file system in the *FileSystem.resource*, it cannot be mounted through a custom mountlist.

The RAM-Handler is not responsible for the reset-resident `RAD` device. The latter is driven by the Fast File System described in section **??** mounted on the *ramdrive.device*, which is a ROM-resident device that makes part of the system RAM available as a block device, and unlike the RAM-Handler, not as a file system. Even though the *ramdrive.device* could in principle carry any other file system, it initializes its contents such that it reassembles the structure of an empty FFS volume and thus does not require a separate initialization step after mounting.

13.6 The Fast File System

The Fast File System (FFS) is the standard Amiga file system and as such included in the ROM. However, the bootstrap code of Amiga host adapters are typically able to load an updated version of this (and other) file systems from the *Rigid Disk Block* (RDB) of the boot disk and make it available to the system. The *System-Startup* module described in section **??** makes updates also available for all other devices, see there for more details.

The FFS serves from the same code multiple variants or flavors of the file system that differ slightly in the structure and organization of data on disk, all listed in table **??** of section **??**. Here the disk structure for all variants is described. The FFS is configured by a mountlist through the keywords and structures listed in section **??**. Section **??** provides further details.

As a hierarchical file system, the FFS supports all file-system relevant packets listed from section **??** onward; packets specific to interactive handlers listed in section **??** are an exception and not implemented. The FFS does support record locking, soft- and hard links and notification requests. External links remain currently exclusive to the RAM-Handler and are not supported. Latest versions of the FFS can also be shut down by `ACTION_DIE` described in section **??**.

13.6.1 FFS Startup and Mount Parameters

The FFS is customized through multiple sources. First of all, by the `DosList`, the `FileSysStartupMsg` pointed to by `dol_Startup` discussed in section **??**, which again contains a pointer to the environment vector, the `DosEnvec` structure, which is specified in section **??**. These structures are setup by the ROM for the floppy drives, the auto-booting host adapter reading options from the RDB or the `Mount` command parsing them from a mountlist. Section **??** provides a list of the configurable options and the corresponding

entries in the mountlist. Actually, the environment vector *is* historically the configuration vector of the AmigaDOS file system which became the de-facto configuration mechanism for many other handlers and file systems.

In some situations, however, the FFS will override the parameters found in the above sources, and therefore ignore parameters in the mountlist. In case the device name is either "trackdisk.device" or "carddisk.device", the FFS will instead request the disk geometry from the underlying exec device. For this, it issues a TD_GETGEOMETRY command when a medium is inserted, the FFS starts up or access to the volume granted by Inhibit(...,DOSFALSE), see section **??**.

Adjusting to the device geometry can also be enabled for FFS version 45 and up for all other devices by the mountlist parameter SUPERFLOPPY, which corresponds to the ENVF_SUPERFLOPPY flag in the de_Interleave element of the environment vector. This flag, along with the environment vector is defined in the file dos/filehandler.h and also introduced in section **??**.

When reinitializing, the FFS updates from the DriveGeometry structure filled by TD_GETGEOMETRY its environment vector. The following algorithm is used to adjust the data in the environment vector from the DriveGeometry structure documented in devices/trackdisk.h:

```
void AdjustEnv(struct DosEnvec *env,
               const struct DriveGeometry *dg) {
    env->de_SizeBlock      = dg->dg_SectorSize >> 2;
    env->de_HighCyl        = dg->dg_Cylinders - 1;
    env->de_Surfaces       = dg->dg_Heads;
    env->de_BlocksPerTrack = dg->dg_TrackSectors;
}
```

All other entries of the environment vector remain untouched. The right-shift for the sector size is necessary because the environment vector measures it in long words rather than bytes as provided by the drive geometry, and dg_Cylinders is a count, whereas de_HighCyl is an inclusive upper bound. In particular, dg_BufMemType is not copied over, and therefore de_BufMemType shall be setup correctly by the mountlist or the Rigid Disk Block.

In case the ENVF_SCSIDIRECT flag is set in addition to ENVF_SUPERFLOPPY, the FFS does not use TD_GETGEOMETRY, but collects the parameters for the drive geometry using the SCSI MODE_SENSE command (see for example [**?**]) requesting all mode pages. The number of sectors and the sector size is taken from the SCSI block descriptor in the mode page header; additionally a SCSI READ_CAPACITY is issued and if successful, the data returned by it overrides the information from the previous step.

The number of heads and the number of blocks per track is obtained through a second MODE_SENSE command reading the SCSI Rigid Disk Page. If this fails or the result is not sensible, the FFS attempts to find some suitable values for the number of heads and the track size by testing values from 16 downwards for the number of heads and stopping whenever it finds a value that divides the total sector count.

The algorithm by which the FFS attempts to obtain the disk geometry through SCSI is not necessarily identical to the algorithm the underlying exec device would use to compute geometry information for the DriveGeometry structure, and thus super floppies mounted with the ENVF_SCSIDIRECT flag set are not necessarily compatible to those mounted without them.

A second configurable parameter is the number of blocks kept in the file system cache. It is originally read from de_NumBuffers in the environment vector, but the AddBuffers() function described in section **??** or the ACTION_MORE_CACHE packet, see section **??**, can be used to adjust this parameter anytime.

13.6.2 The Boot Block

The FFS flavor is initially read from de_DosType element of the environment vector, see section **??**. However, the initial choice is overridden either by formatting (initializing) a volume, or during the (quick)

validation that is triggered if a volume becomes available to the FFS, i.e. if a medium is inserted, the FFS is starting up with the medium inserted or is un-inhibited.

During formatting, the flavor is taken from `dp_Arg2` of the `ACTION_FORMAT` packet, see **??**. If `dp_Arg2` is none of known FFS flavors from table **??** in **??**, the currently active flavor remains in force and is used to initialize the disk structure.

During disk insertion, or if a volume becomes accessible to the FFS by other means such as mounting it or uninhibiting a volume through the `Inhibit()` function (see **??**), the FFS reads the flavor from the first long-word of the *boot block*, even if this block is placed on a hard-disk and not actually used for booting. It is found at sector

$$\texttt{BootBlock} = \texttt{de\_LowCyl} \times \texttt{de\_Surfaces} \times \texttt{de\_BlocksPerTrack}$$

of the device. Note that this is not necessarily the sector 0 if the volume is a partition of a hard disk. The remaining data of the boot block is not used by the FFS but for bootblock booting, namely for floppy disks and for those partitions whose `de_BootBlocks` element (see **??**) in the environment vector is larger than 0.

The boot block does not only identify the FFS flavor, it is also a component of the Kickstart boot process implemented by the `strap` module of the Kickstart ROM. In short, the Kickstart supports two boot mechanisms, boot block booting described here and boot point booting, which depends on an Autoconfig ROM implementing the boot procedure, [**?**] contains the details. For harddisks, the latter boot protocol is most relevant and typically used, though in principle boot block booting as for floppy disks is also supported.

The structure of the boot block is follows:

Table 13.7: Boot Block

Long-word Offset	Content	Notes
0	`DosType`	Encodes the flavor of the FFS
1	`Chksum`	Checksum over this block
2	`RootBlk`	Sector of the Root Block
3	`Code`	Boot code
⋮		

`DosType` contains the file system identifier (or the flavor) of the FFS. It is one of the constants from table **??** in section **??**, or the special value `'BOOT'` which also indicates a bootable (non-FFS) partition. All other values are rejected by the `strap` module, it does not allow bootblock booting from them.

The `DosType` entry does not, in general, define the file system type, i.e. other file systems do not (necessarily) install their identifier here. It is only read by the FFS to configure itself. That the `Info()` function of section **??** returns `DosType` in `id_DiskType` is a side effect of the FFS implementation; it reads the type, and from that decides whether it is able to interpret the disk structure. If `DosType` is none of the constants in table **??**, it does not touch the partition, but leaves the identifier from `DosType` in `id_DiskType` for other programs to observe.

The special value -1 in `DosType` currently puts the FFS into a state where it assumes that there is no disk present. This is, however, only a side effect of its implementation and shall not be relied upon.

`Chksum` is the checksum over the bootblock. It is computed in a somewhat different way from the checksum the FFS defines for all other block types as it is the `strap` module that implements the boot mechanism, and not the file system. It is computed over all long words of `de_BootBlocks` blocks read by `strap` for booting, including the carry which is added into the sum as well. The entire sum over all boot blocks, including the `Chksum`, must be -1 in order to enable execution of the boot code.

The following small assembler code verifies the checksum and expects in register `a0` the pointer to the

loaded boot blocks, and in register d0 the total byte size of the boot block as computed by

$$\texttt{d0} = \texttt{de\_BootBlocks} \times \texttt{de\_SizeBlock} \times 4$$

It returns a non-zero result code in case the boot code is suitable for booting, and zero in case the checksum is not valid:

```
CheckBootBlock:
                move.l  d0,d1
                lsr.l   #2,d1
                moveq   #0,d0
                bra.s   smbbSumDBF
smbbSumLoop:
                add.l   (a0)+,d0
                bcc.s   smbbSumDBF
                addq.l  #1,d0
smbbSumDBF:
                dbf     d1,smbbSumLoop
                not.l   d0
                seq.b   d0
                ext.w   d0
                ext.l   d0
                rts
```

RootBlk is the sector number of the root block, see section **??**, or at least it should be. However, nothing in the system depends on its value, and the FFS does not take the position of the root block from here. Instead, the FFS computes it from the disk geometry as specified in **??**. Actually, RootBlk is deposited by the Install command, which does not even attempt to adjust its value depending on whether a double or high density floppy disk is made bootable. Thus, program code cannot rely on this value being correct.

Code is the boot code in 68000 machine code that is executed if strap accepts the DosType and the Chksum is valid. All the default boot code does is that it locates the *dos.library* resident module and initializes it. Under AmigaDOS 36 and later, it also sets the EBF_SILENTSTART flag in the *expansion.library* to prevent the System-Startup component from opening the boot console, see section **??** for additional details.

13.6.3 Disk Keys and Sectors

The FFS does not address sectors through their physical sector number, but through their *key* which is relative to the start of the partition, and measured in units of blocks, not in units of sectors. The relation between the physical sector S and the key K is given by the following equation:

$$S = \texttt{de\_LowCyl} \times \texttt{de\_Surfaces} \times \texttt{de\_BlocksPerTrack} + K \times \texttt{de\_SectorPerBlock}$$

In particular, the boot block has the key 0. The sector number S is used directly for SCSI transfer which is enabled by the ENVF_SCSIDIRECT flag in the de_Interleave element of the environment vector.

For trackdisk style commands, e.g. TD_READ or TD_READ64 that are used otherwise, exec device drivers address data by a byte offset instead of a sector number. It is computed by multiplying S by de_SizeBlock $\times$ 4. The additional factor of 4 is required because de_SizeBlock is in units of longwords and not a byte count.

Even though de_SizeBlock and de_SectorsPerBlock could be any number, the FFS, and likely many other file systems only accept powers of 2 here. Thus, any multiplication or division by these numbers are easily implementable as shifts.

13.6.4 The Root Block

The Root Block represents the root directory of a volume and contains the objects in the root directory, the creation date of the volume and its name. It corresponds to the path "`:`" and the `ZERO` lock.

The root block is placed in the middle of the volume at key[8]:

$$\left\lfloor \frac{\texttt{de\_Reserved} + \left\lfloor \frac{(\texttt{de\_HighCyl} - \texttt{de\_LowCyl} + 1) \times \texttt{de\_Surfaces} \times \texttt{de\_BlocksPerTrack}}{\texttt{de\_SectorsPerBlock}} \right\rfloor - 1}{2} \right\rfloor$$

Note that the element name `de_BlocksPerTrack` is actually misleading as it is the size of a track in (physical) sectors and not in (logical) blocks or keys. The brackets $\lfloor \cdot \rfloor$ indicate rounding down to the next integer.

The structure of the root block is as follows:

Table 13.8: FFS Root Block

Long-word Offset	Content	Notes
0	`Type`	shall be 2, this is the BCPL constant `T.SHORT`
1	`0`	header key, always 0 in root
2	`0`	highest sequence number, always 0 in root
3	`HTSize`	entries in the hash table = Block size - 56
4	`0`	reserved for future use
5	`Chksum`	LW sum over block is 0
6	`Hash`	hash chains for objects in the root
L-50	`BMFlag`	-1 if Bitmap is valid
L-49	`BMKeys`	keys of the bitmap
L-24	`BMExt`	key of the bitmap extension block
L-23	`Days`	timestamp of last directory change
L-22	`Mins`	
L-21	`Ticks`	
L-20	`Name`	volume name as BSTR
L-12	`0`	reserved for future use
L-11	`0`	reserved for future use
L-10	`Days`	timestamp of last volume change
L-9	`Mins`	
L-8	`Ticks`	
L-7	`Days`	volume creation time
L-6	`Mins`	
L-5	`Ticks`	
L-4	`0`	reserved for future use
L-3	`0`	reserved for future use
L-2	`DCache`	key of directory cache block if used
L-1	`SecType`	shall be 1, this is the BCPL constant `ST.ROOT`

Each entry in the above table is 4 bytes, i.e. one long word large. As the FFS supports multiple block sizes, some elements of this structure are placed relative to the end of the block. Such element are indicated

[8] In [**?**], the number of sectors per block is not included as earlier FFS versions did not support this parameter.

in the above table as $L - x$ where L is the block size in long works, i.e.

$$L = \texttt{de\_SizeBlock} \times \texttt{de\_SectorsPerBlock}$$

Dashed table boundaries indicate that the corresponding elements extend over multiple long words. In particular, the list of hash-keys is a variably sized number of long words large, namely $L - 56$ long words.

The block type is indicated by `Type` and `SecType`. For the root block, these elements are 2 and 1, respectively.

`HTSize` is the number of hash keys stored in the root block. The FFS stores in the root block $L - 56$ hash keys, from offset 6 to $L - 51$.

`Chksum` is a checksum over the block. Its value is computed such that the long word sum over all long words, including the checksum, is zero.

`Hash` stores the hash-chain of all objects in the root directory. The FFS computes for each file system object a *hash* that is derived from its name. As multiple objects can have the same hash and thus hash conflicts can arise, all objects of the same hash are kept in a singly linked list. The head of each list is kept in the `Hash` array, and the end of a hash list is indicated by key 0. This is actually the key of the bootblock (see **??**), which shall be included in the `de_Reserved` blocks at the start of the partition that cannot be allocated for disk objects.

How the FFS computes the hash from the name depends on the FFS flavor. The following algorithm computes the hash key from the object name, represented as a BSTR, i.e. the first character is the length of the string:

```
ULONG ComputeHash(UBYTE *name, BOOL isInternational)
{
  ULONG size = *name++; /* String size, this is a BSTR */
  ULONG hash = size;    /* Initial hash is string length */

  while(size) {
   ULONG c = *name++;
   /* Is this an international flavour? */
   if (isInternational) {
    if (c != 0xf7 && ((c >= 0xe0 && c <= 0xfe) ||
                      (c >= 0x61 && c <= 0x7a))) {
     c -= 0x20; /* make upper case */
    }
   } else if (c >= 0x61 && c <= 0x7a) {
     c -= 0x20; /* make upper case */
   }
   hash = (hash * 13 + c) & 0x7ff;
  }
  return hash % HTSize;
}
```

In the above, `HTSize` is the value of the field of the same name in the disk root block. As seen from the code, the non-international versions of the FFS only convert the ASCII characters between 'a' and 'z' to upper case, whereas the international version performs this conversion also for all letters in the Latin-1 supplement set. The FFS *does not* use the *utility.library* for the upper-case conversion and thus hashing does not depend on the selected locale. However, it depends on the Latin-1 encoding and will not map characters correctly for many other encodings. Latin-15 only replaces the international currency symbol with the Euro-sign which is outside the range of characters that are converted to upper case. The algorithm above therefore also works correctly for Latin-15.

BMFlag is DOSTRUE in case the bitmap is valid. The bitmap keeps information on which blocks of the volume are allocated and which are free. More on this block is found in section **??**. If the quick validation on disk insertion finds that this field is 0, then the disk status is set to non-validated. The FFS then recomputes the bitmap by a complete directory scan, recursively locating all objects on disks and marking the blocks they reside in as occupied.

BMKeys is an array containing the keys of the bitmap blocks. How many bitmap blocks are required to represent the bitmap depends on the size of the volume and the size of a bitmap block. Unused keys are 0.

BMExt is the key of a bitmap extension block if the above bitmap array is not sufficient to represent all blocks of the volume. This block is described in section **??**.

The fields at offsets $L-23$ to $L-21$ form a DateStamp structure as specified in chapter **??**. The date and time there indicate the last change within the root directory.

Name is the volume name, encoded as BSTR; the first byte is therefore the size of the name. This field is 8 long words large. While this is sufficient space for 31 characters, one extra character is reserved, limiting the volume name to 30 characters.

The fields at offsets $L-10$ to $L-8$ form a DateStamp structure that is updated on every change of the volume.

The fields at offsets $L-7$ to $L-5$ are also a DateStamp structure that represent the time and date at which the volume was initialized. The packet ACTION_SERIALIZE_DISK updates this date, too, but it remains otherwise unchanged even if objects in the root directory change.

DCache is the key of the directory cache list of blocks. It is only used for the flavors of the FFS that utilize such a cache. This block is described in section **??**.

Unlike user directories, the root block lacks a list of hard links that point to it. This has the consequence that the FFS does not allow to create hard links to the root directory of a volume.

13.6.5 The User Directory Block

The user directory block represents a sub-directory of the volume root or another user directory. It is enqueued in the hash-chain corresponding to its name within its parent block.

The user directory block exists in multiple variants, depending on the flavor of the FFS. Table **??** applies to all FFS variants except the long file name variants DOS\06 and DOS\07.

Table 13.9: FFS User Directory Block

Long-word Offset	Content	Notes
0	Type	shall be 2, this is the BCPL constant T.SHORT
1	OwnKey	key of this block (self-reference)
2	0	reserved for future use
3	0	reserved for future use
4	0	reserved for future use
5	Chksum	LW sum over block is 0
6	Hash	hash chains for objects in the directory
L-50	0	reserved for future use
L-49	Owner	reserved for owner ID
L-48	PrtBits	protection bits as in section **??**
L-47	0	reserved for future use
L-46	Comment	directory comment as BSTR

L-26	0	reserved for future use
L-23	Days	timestamp of last directory change
L-22	Mins	
L-21	Ticks	
L-20	Name	directory name as BSTR
L-12	NameX1	name extension for DOS\08
L-11	0	reserved for future use
L-10	BckLink	key of first hard link to this object
L-9	NameX2	name extension for DOS\08
L-4	NxtHash	next key of the same hash
L-3	Parent	key of the parent directory
L-2	DCache	key of directory cache block if used
L-1	SecType	shall be 2, this is the BCPL constant ST.USERDIR

Type is the primary type of this block. It is always 2, which is the BCPL constant T.SHORT

OwnKey is the key of this object, i.e. it is the key of the block itself.

Chksum is the checksum over the entire block. It is chosen such that the long word sum over the entire block, including this field, is zero.

Hash contains for user directories all the hashes of the file system objects contained within.

OwnerID is reserved for the owner ID of this directory that can be set with SetOwner(), see section **??**. As the FFS has no means to validate the directory owner, this field does not bear any practical meaning.

PrtBits are the protection bits of this directory. The FFS actually ignores the protection bits on directories, but stores the value here anyhow. The protection bits are explained in section **??**.

Comment contains a potential comment for this directory. The comment is represented as a BSTR with the comment length in the first character. There is room for 80 bytes, i.e. the maximum comment size is 79 characters.

The fields at offset $L - 23$ to $L - 21$ form a DateStamp structure, see chapter **??**, that identifies the timestamp of the last change of this directory.

Name is the name of this directory encoded as BSTR with the name length in the first byte; even though 32 bytes are available, the FFS reserves one character, thus limiting the maximal directory name size to 30.

NameX1 and NameX2 are name extensions that are used by the DOS\08 flavor of the FFS. They add additional 24 bytes of storage for the directory name. Again, one character is reserved, limiting the maximum directory name size in this variant to 54. The directory name extends from Name, then overflows into NameX1 and from there to NameX2.

BckLink is the key of the first hard link to this directory. The link of this key replaces the original directory header block in case the directory itself is deleted, and the link is converted to a directory.

NxtHash is the key of the next object using the same hash as this directory itself.

Parent is the key of the parent directory, or the key of the root block in case this directory is located within the volume root.

DCache is the key of the first directory cache block. This key is only used for FFS flavors with directory caching enabled. Otherwise, it stays 0.

SecType along with Type identifies the type of this block. The value of this field shall be 2, identifying this as a user directory block.

For the long file name enabled flavors of the FFS, namely `DOS\06` and `DOS\07`, this block looks somewhat different:

Table 13.10: Long-Filename FFS Directory User Header Block

Long-word Offset	Content	Notes
0	`Type`	shall be 2, this is the BCPL constant `T.SHORT`
1	`OwnKey`	key of this block (self-reference)
2	`0`	reserved for future use
3	`0`	reserved for future use
4	`0`	reserved for future use
5	`Chksum`	LW sum over block is 0
6	`Hash`	hash chains for objects in the directory
L-50	`0`	reserved for future use
L-49	`Owner`	reserved for owner ID
L-48	`PrtBits`	protection bits as in section **??**
L-47	`0`	reserved for future use
L-46	`NaC`	name and comment BSTR
L-18	`CmtBlk`	key of comment block if necessary
L-17	`0`	reserved for future use
L-16	`0`	reserved for future use
L-15	`Days`	timestamp of last directory change
L-14	`Mins`	
L-13	`Ticks`	
L-12	`0`	reserved for future use
L-11	`0`	reserved for future use
L-10	`BckLink`	key of first hard link to this object
L-9	`0`	reserved for future use
L-4	`NxtHash`	next key of the same hash
L-3	`Parent`	key of the parent directory
L-2	`0`	reserved for future use
L-1	`SecType`	shall be 2, this is the BCPL constant `ST.USERDIR`

Compared to the regular directory header block, the time stamp of the last modification date is relocated to offsets $L-15$ and following, and the name and comment fields are replaced by a new field at offset $L-46$. All other fields remain unchanged.

As this flavor of the flavor of the FFS does not offer a directory cache, the field at $L-2$ remains unused and thus always contains the key 0.

`NaC` contains both the file name and file comment of the directory as two BSTRs, directly placed next to each other. The file name comes first, followed by the comment, if any. This field is capable of storing 112 bytes, but the FFS reserves one byte making in total 110 bytes available. In case the total size required for file and comment is too large and they cannot be hosted by this field, an additional comment block is created keeping only the comment, and this field then only keeps the file name.

`CmtBlk` is the key of the comment block, keeping the file comment if the `NaC` field is too short. If no comment block is needed, this field is 0. The comment block is described in section **??**.

13.6.6 The File Header Block

The file header block represents a file in a directory or the volume root. It is enqueued in one of the hash-chains of the root block or its parent user directory block. This block exists in multiple variants, depending on the flavor of the FFS. Table **??** applies to all FFS variants except the long file name variants `DOS\06` and `DOS\07`.

Table 13.11: FFS File Header Block

Long-word Offset	Content	Notes
0	`Type`	shall be 2, this is the BCPL constant `T.SHORT`
1	`OwnKey`	key of this block (self-reference)
2	`BlkCnt`	number of data block keys included
3	`0`	reserved for future use
4	`Data1st`	first data block of the file
5	`Chksum`	LW sum over block is 0
6	`DataBlk`	first $L-56$ data blocks of the file
L-50	`0`	reserved for future use
L-49	`Owner`	reserved for owner ID
L-48	`PrtBits`	protection bits as in section **??**
L-47	`Size`	size of the file in bytes
L-46	`Comment`	file comment as BSTR
L-26	`0`	reserved for future use
L-23	`Days`	timestamp of last file change
L-22	`Mins`	
L-21	`Ticks`	
L-20	`Name`	file name as BSTR
L-12	`NameX1`	name extension for `DOS\08`
L-11	`0`	reserved for future use
L-10	`BckLink`	key of first hard link to this object
L-9	`NameX2`	name extension for `DOS\08`
L-4	`NxtHash`	next key of the same hash
L-3	`Parent`	key of the parent directory
L-2	`FileExt`	key of first file extension block
L-1	`SecType`	shall be -3, this is the BCPL constant `ST.FILE`

`Type` is the primary type of this block. It is always 2, which is the BCPL constant `T.SHORT`

`OwnKey` is the key of this object, i.e. it is the key of this block itself.

`BlkCnt` is the number of occupied data block keys included in this block. Due to the limited number of slots, this number is smaller or equal than $L-56$.

`Data1st` is the key of the first data block of the file. This data block is actually made available twice, once here, and once again at offset $L-51$. Probably the key at this offset was historically used for sequential access into the file, whereas the block list at offset $L-6$ and following was used for random access.

`Chksum` is the checksum over the entire block. It is chosen such that the long word sum over the entire block, including this field, is zero.

`DataBlk` is an array containing the first $L - 56$ data blocks of the file. The array is filled from its bottom-end, i.e. $L - 51$ contains the key of first data block, $L - 52$ the second key and so on. If this array overflows, additional blocks are in one or more file extension blocks chained at $L - 2$. This block type is defined in section **??**.

`OwnerID` is reserved for the owner ID of this file that can be set with `SetOwner()`, see section **??**. As the FFS has no means to validate the file owner, this field does not bear any practical meaning.

`PrtBits` are the protection bits of the file, encoded as in section **??**. The four least-significant bits corresponding to the readable, writable, executable and deletable features are stored inverted, i.e. a bit being 0 indicates that the corresponding feature is available. This is the same encoding as in the `FileInfoBlock` structure.

`Comment` contains a potential comment for this file. The comment is represented as a BSTR with the comment length in the first character. There is room for 80 bytes, i.e. the maximum comment size is 79 characters.

The fields at offset $L - 23$ to $L - 21$ form a `DateStamp` structure that identifies the timestamp of the last change to this file.

`Name` is the name of this file encoded as BSTR with the name length in the first byte; even though 32 bytes are available here, the FFS reserves one character, thus limiting the maximal file name size to 30 characters.

`NameX1` and `NameX2` are name extensions that are used by the `DOS\08` flavor of the FFS. They add additional 24 bytes of storage for the file name. Again, one character is reserved, limiting the maximum file name size for this variant to 54. The file name extends from `Name`, then overflows into `NameX1` and from there to `NameX2`.

`BckLink` is the key of the first hard link to this file. The link of this key replaces the original directory header block in case the file itself is deleted, and then is converted from a link to a file.

`NxtHash` is the key of the next object using the same hash as this file.

`Parent` is the key of the directory containing this file, or the key of the root block in case this file is located in the volume root.

`FileExt` is the key of the first file extension block. This key is only used if the file requires more than $L - 56$ blocks. Otherwise, it stays 0.

`SecType` along with `Type` identifies the type of this block. The value of this field shall be -3, identifying this as file header block.

For the long file name enabled flavors of the FFS, namely `DOS\06` and `DOS\07`, this block looks somewhat different:

Table 13.12: Long-Filename FFS File Header Block

Long-word Offset	Content	Notes
0	`Type`	shall be 2, this is the BCPL constant `T.SHORT`
1	`OwnKey`	key of this block (self-reference)
2	`HighSeq`	total number of blocks occupied
3	`0`	reserved for future use
4	`Data1st`	first data block of the file
5	`Chksum`	LW sum over block is 0
6	`DataBlk`	first $L - 56$ keys of data blocks
⋮		
L-50	`0`	reserved for future use
L-49	`Owner`	reserved for owner ID
L-48	`PrtBits`	protection bits as in section **??**

L-47	0	reserved for future use
L-46	NaC	name and comment BSTR
L-18	CmtBlk	key of comment block if necessary
L-17	0	reserved for future use
L-16	0	reserved for future use
L-15	Days	timestamp of last file change
L-14	Mins	
L-13	Ticks	
L-12	0	reserved for future use
L-11	0	reserved for future use
L-10	BckLink	key of first hard link to this object
L-9	0	reserved for future use
L-4	NxtHash	next key of the same hash
L-3	Parent	key of the parent directory
L-2	FileExt	key of first file extension block
L-1	SecType	shall be -3, this is the BCPL constant ST.FILE

Compared to the regular file header block, the time stamp of the last modification date is relocated to offsets $L-15$ and following, and the name and comment fields are replaced by a new field at offset $L-46$.

NaC contains both the file name and file comment as two BSTRs, directly placed next to each other. The file name comes first, followed by the comment. This field is capable of storing 112 bytes, but the FFS reserves one byte making in total 110 bytes available. In case the comment is too long and cannot be placed in this field, an additional comment block is created keeping only the comment, and this field then only keeps the file name.

CmtBlk is the key of the comment block, keeping the file comment if the NaC field is too short. If no comment block is needed, this field is 0. The comment block is described in section **??**.

13.6.7 The Soft and Hard Link Block

The soft- and hard link block represent a soft link or a hard link to another file system object. Similar to the file header and user directory blocks, this block exists in two variants, depending on the flavor of the FFS. Table **??** applies to all FFS variants except the long file name variants DOS\06 and DOS\07.

Table 13.13: FFS Link Block

Long-word Offset	Content	Notes
0	Type	shall be 2, this is the BCPL constant T.SHORT
1	OwnKey	key of this block (self-reference)
2	0	reserved for future use
3	0	reserved for future use
4	0	reserved for future use
5	Chksum	LW sum over block is 0
6	Target	link target for soft links
L-50	0	reserved for future use
L-49	Owner	reserved for owner ID
L-48	PrtBits	protection bits as in section **??**
L-47	0	reserved for future use

L-46	`Comment`	link comment as BSTR
L-26	`0`	reserved for future use
L-23	`Days`	timestamp of link creation
L-22	`Mins`	
L-21	`Ticks`	
L-20	`Name`	link name as BSTR
L-12	`NameX1`	name extension for `DOS\08`
L-11	`Link`	key of link target for hard links
L-10	`BckLink`	key of next hard link to the same object
L-9	`NameX2`	name extension for `DOS\08`
L-4	`NxtHash`	next key of the same hash
L-3	`Parent`	key of the parent directory
L-2	`0`	reserved for future use
L-1	`SecType`	identifies the type of the link

`Type` is the primary type of this block. It is always 2, which is the BCPL constant `T.SHORT`

`OwnKey` is the key of this object, i.e. it is the key of this block itself.

`Chksum` is the checksum over the entire block. It is chosen such that the long word sum over the entire block, including this field, is zero.

`Target` is the path of the link target for soft links. This is stored as `NUL`-terminated C-string, not as BSTR[9]. The maximum path name that can be stored here is $(L-56)\times 4-1$ characters long. Unfortunately, all versions of the FFS currently do not check the maximum link name and damage the file system structure if an attempt is made to create a soft link to a path that is too long. For a default floppy with 512 bytes per sector, this field leaves room for paths of up to 287 characters.

`OwnerID` is reserved for the owner ID of this link that can be set with `SetOwner()`, see section **??**. As the FFS has no means to validate the link owner, this field does not bear any practical meaning.

`PrtBits` are the protection bits of the link, encoded as in section **??**. However, the practical value of these protection bits is close to zero as locking a link provides a lock to the linked object, and thus the protection bits stored here are not actually checked and neither accessible, except by walking a directory through `ExNext()` or `ExAll()`, see section **??** and following. For hard links, the FFS keeps the protection bits of the link and the link target synchronized, but clearly cannot do so for soft links.

`Comment` contains a potential comment for this link. The comment is represented as a BSTR with the comment length in the first character. The block has room for 80 bytes, i.e. the maximum comment size is 79 characters. Comments of links *can* be set with `SetComment()` as the FFS does not attempt to follow links when setting comments. Thus link and link target have separate comments.

The fields at offset $L-23$ to $L-21$ form a `DateStamp` structure that identifies the timestamp of the creation of the link. For hard links, this field bears no practical meaning as a `SetFileDate()` (see section **??**) will update the date of the link target and not the date of the link itself. An attempt to change the date of a soft link creates an `ERROR_IS_SOFT_LINK` and thus instructs the client of the FFS to rather redirect the request to the link target. Thus, this date can be seen when walking a directory, but it cannot be changed.

`Name` is the name of the link encoded as BSTR with the name length in the first byte; even though 32 bytes are available here, the FFS reserves one character, thus limiting the maximal size to 30 characters.

[9]The information in [**?**] that this is a BSTR is incorrect

NameX1 and NameX2 are name extensions that are used by the DOS\08 flavor of the FFS. They add additional 24 bytes of storage for the directory name. Again, one character is reserved, limiting the maximum size for this variant to 54. The link name extends from Name, then overflows into NameX1 and from there to NameX2.

Link is the key of the file header or user directory key for hard links, that is, the key of the link target. For soft links, this field is not used and set to 0.

BckLink is the key to the next hard link to the same link target, or 0 if there is no further hard link to the same target. Thus, all links to the same target are chained through BckLink. This field is 0 for soft links.

NxtHash is the key of the next object using the same hash as this directory entry.

Parent is the key of the directory containing this link, or the key of the root block in case this link is directly in the volume root.

SecType along with Type identifies the type of this block. It can be either -4, that is ST.LINKFILE for a hard link to a file, or 4, which is ST.LINKDIR, for a hard link to a directory. This field is 3, corresponding to ST.SOFTLINK, for soft links.

For the long file name enabled flavors of the FFS, namely DOS\06 and DOS\07, this block looks somewhat different:

Table 13.14: Long-Filename FFS Link Block

Long-word Offset	Content	Notes
0	Type	shall be 2, this is the BCPL constant T.SHORT
1	OwnKey	key of this block (self-reference)
2	0	reserved for future use
3	0	reserved for future use
4	0	reserved for future use
5	Chksum	LW sum over block is 0
6	Target	link target for soft links
L-50	0	reserved for future use
L-49	Owner	reserved for owner ID
L-48	PrtBits	protection bits as in section **??**
L-47	0	reserved for future use
L-46	NaC	name and comment BSTR
L-18	CmtBlk	key of comment block if necessary
L-17	0	reserved for future use
L-16	0	reserved for future use
L-15	Days	timestamp of last directory change
L-14	Mins	
L-13	Ticks	
L-12	0	reserved for future use
L-11	Link	key of link target for hard links
L-10	BckLink	key of next hard link to the same object
L-9	0	reserved for future use
L-4	NxtHash	next key of the same hash
L-3	Parent	key of the parent directory
L-2	0	reserved for future use
L-1	SecType	identifies the type of the link

Compared to the regular link block, the time stamp of the last modification date is relocated to offsets $L-15$ and following, and the name and comment fields are replaced by a new field at offset $L-46$.

`NaC` contains both the link name and comment as two BSTRs, directly placed next to each other. The link name comes first, followed by the comment. This field is capable of storing 112 bytes, but the FFS reserves one byte making in total 110 bytes available. In case the comment is too long and cannot be placed in this field, an additional comment block is created keeping only the comment, and this field then only keeps the file name.

`CmtBlk` is the key of the comment block, keeping the file comment if the `NaC` field is too short. If no comment block is needed, this field is 0. The comment block is described in section **??**.

13.6.8 The File Extension Block

The file extension block keeps keys of additional file data blocks in case the $L-56$ keys in the file header block are not sufficient to keep all keys. It looks as follows:

Table 13.15: File Extension Block

Long-word Offset	Content	Notes
0	`Type`	shall be 16, this is the BCPL constant `T.LIST`
1	`OwnKey`	key of this block (self-reference)
2	`BlkCnt`	number of data block keys included
3	`0`	reserved for future use
4	`0`	reserved for future use
5	`Chksum`	LW sum over block is 0
6	`DataBlk`	next $L-56$ keys of data blocks
L-50	`0`	reserved for future use
L-3	`Parent`	key of the file header block
L-2	`NextExt`	key of next file extension block
L-1	`SecType`	shall be -3, this is the BCPL constant `ST.FILE`

`Type` is the primary type of this block. It is always 16, which is the BCPL constant `T.LIST`

`OwnKey` is the key of this block itself.

`BlkCnt` is the number of data block keys included in this block. Due to the limited number of slots, this number is smaller or equal than $L-56$.

`Chksum` is the checksum over the entire block. It is chosen such that the long-word sum over the entire block, including this field, is zero.

`DataBlk` is an array containing the next $L-56$ data blocks of the file. The array is filled from its bottom end, i.e. $L-51$ contains the key of first data block referenced in this extension block, $L-52$ the second key and so on. If this array overflows, additional blocks are provided in another file extension block whose key is provided at offset $L-2$.

`Parent` is the key of the file header block of the file whose data block keys are extended by this block.

`NextExt` is the key of the next file extension block if this block is not sufficient to keep all data block keys. Otherwise, if this list the last file extension block, it is 0.

`SecType` along with `Type` identifies the type of this block. The value of this field shall be -3, that is `ST.FILE`, identifying this block as belonging to a file.

13.6.9 The Comment Block

This block keeps a file comment for the long file name FFS flavors `DOS\06` and `DOS\07` in case the file header, user directory or link block does not provide sufficient room to keep both the file name and the comment.

Table 13.16: Comment Block

Long-word Offset	Content	Notes
0	`Type`	shall be 64, this is `T.COMMENT`
1	`OwnKey`	key of this block (self-reference)
2	`Parent`	key of the header block
3	`0`	reserved for future use
4	`0`	reserved for future use
5	`Chksum`	LW sum over block is 0
6	`Comment`	object comment as BSTR
26	`0`	reserved for future use

`Type` identifies the type of this block. The value placed here shall be 64, corresponding to the constant `T.COMMENT`.

`OwnKey` is the key of this block itself.

`Parent` is the key of the file header, user directory or link block to which the comment in this block applies and which is extended by this block.

`Chksum` is the checksum over the entire block. It is chosen such that the long-word sum over the entire block, including this field, is zero.

`Comment` is the comment, stored as a BSTR with the first character containing its size. This field is 80 bytes large, sufficient for comments of 79 characters.

The remaining bytes of this block shall be 0 and remain available for future extension.

13.6.10 The Directory Cache Block

This block type is only used by the directory cache flavors of the FFS, namely `DOS\04` and `DOS\05`. It keeps, in a more compact form, the contents of directories. This block looks as follows:

Table 13.17: Directory Cache Block

Long-word Offset	Content	Notes
0	`Type`	shall be 33, this is `T.DIRLIST`
1	`OwnKey`	key of this block (self-reference)
2	`Parent`	key of the user directory block
3	`NumNtry`	number of entries in this block
4	`NextBlk`	key of the next directory cache block
5	`Chksum`	LW sum over block is 0
6	`Entries`	Directory content (see below)

`Type` identifies the type of this block; the constant put here is actually `T.DIRLIST_KEY` = 32 or'd with the version of the directory cache data, which is currently 1.

`OwnKey` is the key of this block itself.

Parent is the key of the directory header block of the user directory cached here, or the key of the root block if this is the cache of the volume root directory.

NumNtry is the number of directory entries cached in this block. Each entry has a structure as indicated in table **??**. Such entries cannot extend over block boundaries; if a new entry does not fit entirely into a block, another directory cache block is allocated. A directory cache block may also contain 0 entries as these blocks are never released. Thus, directory caches can grow very large, and they are only rebuild when the disk requires full validation, i.e. its bitmap becomes invalid.

NextBlk is the key of the next directory cache block for the same directory, or 0 in case this is the last block.

Chksum is the checksum over the entire block. It is chosen such that the long word sum over the entire block, including this field, is zero.

Entries contains the payload data of the directory cache. It consists of zero or more entries of the following variably sized structure:

Table 13.18: Directory Cache Entry

Size (bits)	Content	Notes
32	Key	key of the referenced object
32	Size	size of the object in bytes
32	PrtBits	protection bits of the object
32	OwnerID	owner ID of the object
16	Days	timestamp of last change
16	Mins	
16	Ticks	
8	SecType	secondary type of the object
variable	Name	object name as BSTR
variable	Comment	object comment as BSTR
pad(16)	padding	padding to 16-bit boundary

The fields of this structure are as follows:

Key is the key of the file header block, the user directory block or the link block of this directory entry, depending on the type of the file system object stored here.

Size is the byte size of the object, or 0 for links and directories.

PrtBits are the protection bits as represented in the FileInfoBlock structure.

OwnerID is reserved for a 32-bit group and owner ID that can be set by the SetOwner() function (see **??**). However, as the FFS has no means to verify access rights to an object, this field bears no practical meaning.

Days, Mins and Ticks are the timestamp of the last time the corresponding object was modified and a copy from the corresponding header block provided by Key. However, unlike there, only 16 bits are available for each field. This is sufficient if the DateStamp structure is normalized, i.e. each field is as small as possible.

SecType is a copy from the SecType field of the file, user directory or link block indexed by Key. All possible values can be represented by a signed byte and are thus abbreviated here in an 8-bit field.

Name is a copy of the Name field of the file, user directory or link, though only the minimal number of bytes are copied, i.e. $N + 1$ bytes for a file name of size N. The first byte of Name is its length, i.e. it is a BSTR without NUL-termination.

Comment follows directly after the last byte of Name and is a copy of the Comment field of the file, user directory or link block; again, only the minimal amount of bytes are copied to the directory cache, i.e.

the length byte and the comment itself. This forms again a BSTR without any NUL termination, and not a C string.

padding is an optional padding byte to make the entire structure an even number of bytes large such that the key of the next directory entry is on an even address in memory.

The directory cache does not store the targets of hard links or soft links; in particular, if the contents of a directory of a cache-enabled file system is listed, this information is gained from the regular directory structure.

Within a directory cache block, zero or more directory cache entries follow each other; their count is provided by the NumNtry field in the directory cache block. If the FFS has to delete entries from the directory, it moves entries within the current block upwards over the released entry. In worst case, no entries remain in a directory cache block. Such blocks are not released, but remain available to accept new entries.

While the directory cache increases the performance of listing directory contents, keeping the directory cache in sync with the regular directory structure requires additional overhead as directory cache blocks need to be allocated, filled, and entries be moved within the blocks. Thus, while performance for listing directory contents can improve, the performance for creating and deleting file system objects decreases. Usage of the directory cache is therefore discouraged.

13.6.11 The Data Block

Data blocks contain the payload data of files. It comes in two variants: The OFS flavors of the FFS, namely DOS\00, DOS\02, DOS\04 and DOS\06 keep redundant information in the data block that makes the file system structure very robust against media corruption; however, as this information needs to be stripped off before the payload data is delivered to FFS clients, and thus the block contents cannot be moved by DMA into client supplied buffers, these variants are slow.

All remaining variants, including DOS\08, only keep payload data in the data blocks. This enables the FFS to directly transmit data from the medium to the target buffer of the client if the de_Mask allows it. Section **??** provides more information on DMA and its pitfalls. If the host adapter offers DMA, the CPU is not involved in copying the data and thus these variants of the FFS are faster, though also less robust. However, modern media rarely corrupt data, unlike floppy disks, and therefore these FFS flavors are generally recommended. The OFS flavors are only useful for slow and unreliable data carriers.

The following table describes the structure of an OFS data block:

Table 13.19: OFS Data Block

Long-word Offset	Content	Notes
0	Type	shall be 8, this is T.DATA
1	Header	key of the file header
2	SeqNum	sequence number of this block
3	Size	data bytes in this block
4	NxtBlk	next data block of this file
5	Chksum	LW sum over block is 0
6	Data	payload data

Type is the primary type of this block. It is always 8, which is the constant T.DATA

Header is the key of the file header block this file belongs to.

SeqNum is the sequential number of this block within the file. The first data block of the file has the sequential number 1, the next one 2 and so on.

Size is the number of valid bytes within this data block. Valid data does not necessarily extend to the

end of the block.

`Chksum` is the checksum over the entire block. It is chosen such that the long word sum over the entire block, including this field, is zero.

`Data` is the actual payload data of the block. It consists of `Size` bytes forming the contents of the file.

The FFS data block does not have any structure, it contains only payload data. This has the consequence that a disk scan, e.g. by a disk salvage tool, cannot safely identify whether a block carries administration information of the disk, or is rather a data block that, by pure coincidence, reassembles an administration block of one of the types listed in this section. Various disk salvage tools fell into this pitfall identifying blocks as administration blocks that were, actually, data blocks allocated for a file.

13.6.12 The Bitmap Block

Bitmap blocks keep a bitmap — one bit per key — describing which keys are already occupied for administration or payload data, and which keys are still available. Depending on the size of the volume, one or many bitmap blocks exist. If the 25 keys available in the root block are not sufficient, bitmap extension blocks described in section **??** are needed.

The structure of a bitmap block is as follows:

Table 13.20: Bitmap Block

Long-word Offset	Content	Notes
0	`Chksum`	LW sum over block is 0
1	`Bitmap`	bitmap of available blocks

`Chksum` is the checksum over the entire block. It is chosen such that the long word sum over the entire block, including this field, is zero.

`Bitmap` holds for every available key administrated by this bitmap a bit that indicates whether that key is available or not. If the bit is 1, the key is free, and if 0, the key is released.

Bits are addressed in groups of long words such that the least significant bit of each long word corresponds to the lowest key and the most significant bit of a long word to the highest key within this long word. The least significant bit of the long word at offset 1 of the first bitmap of a volume corresponds to the key `de_Reserved`, i.e. the reserved blocks at the start of a volume *are not* represented in the bitmap. As key 0 corresponds to the boot block and this block keeps the flavor of the FFS, and potentially boot code, `de_Reserved` cannot be 0 as otherwise the FFS could allocate it as key, and thus overwrite parts of its administration information. While the FFS could, in principle, always reserve key 0 for such purpose in the bitmap, no such provisions are made.

Identifying whether a particular key is allocated is demonstrated by the following algorithm: It takes the number of long words per block (e.g. 128 for a standard floppy disk, i.e. 512 bytes per block), the number of reserved blocks, the key to investigate and the key of the root block. It assumes that `readKey()` brings the key provided by its argument to memory, and that this function returns a pointer to an array of `ULONG`s representing the block contents:

```
/* Bring key to memory */
ULONG *readKey(ULONG key);

/* Offsets of block fields */
#define BMKeys 49
#define BMNext 1
#define BMExt  24
```

```
/* Check whether a particular key is allocated */
LONG isKeyAllocated(ULONG longsperblock,ULONG reservedblocks,
                    ULONG key,ULONG rootkey)
{
ULONG  keysperbitmap;
ULONG  bitmap;
ULONG  keyinbitmap;
ULONG  longoffset;
ULONG *block;
ULONG  bitinlong;

  /* compute the number of keys per bitmap */
  keysperbitmap = (longsperblock - 1) * 32;
  /* compute the bitmap index in all bitmaps */
  bitmap        = (key - reservedblocks) / keysperbitmap;
  /* compute the key within the bitmap */
  keyinbitmap   = (key - reservedblocks) % keysperbitmap;
  /* compute the LW offset within the bitmap */
  longoffset    = keyinbitmap / 32 + 1;
  /* compute the bit within the long */
  bitinlong     = keyinbitmap % 32;
  /* read the root block */
  block         = readKey(rootkey);

  /* Check whether the bitmap is linked in the root
  ** block or not. The first 25 are.
  */
  if (bitmap < 25) {
     /* Bring the proper keymap into memory */
     block = readKey(block[bitmap + longsperblock - BMKeys]);
  } else {
     LONG extension,keyoffset;

     /* Compute the extension block required,
     ** and key offset within the extension block
     ** to the bitmap block.
     */
     extension = (bitmap - 25) / (longsperblock - 1);
     keyoffset = (bitmap - 25) % (longsperblock - 1);
     /* read the first extension block */
     block     = readKey(block[longsperblock - BMExt]);
     /* Follow the link chain of extension
     ** blocks to find the right one
     */
     while(extension > 0) {
       block   = readKey(block[longsperblock - BMNext]);
       extension--;
     }
     /* Now read the right bitmap */
     block     = readKey(block[keyoffset]);
```

```
  }

  /* check the bit corresponding to the key */
  if (block[longoffset] & (1UL << bitinlong)) {
    return DOSTRUE; /* is allocated */
  } else {
    return DOSFALSE; /* is free */
  }
}
```

13.6.13 The Bitmap Extension Block

The bitmap extension block keeps the keys of additional bitmap blocks in case the number of bitmap keys in the root block (25, namely) are not sufficient. This block has the following structure:

Table 13.21: Bitmap Extension Block

Long-word Offset	Content	Notes
0	`BMKeys`	additional $L-1$ bitmap keys
L-1	`BMNext`	key of another bitmap extension block

This block, along with the bitmap, does not carry the usual block identifiers, i.e. a primary and a secondary type. This is probably because bitmaps are disposable and can be recreated by a validation process if the FFS needs them.

`BMKeys` is an array of $L-1$ keys, each of which contains a bitmap for the subsequent part of the volume. The slots in this block are allocated top to bottom, with non-used entries set to 0.

`BMNext` is the key of another extension block if this extension block is not sufficient. It is 0 in case this is the last bitmap extension block.

13.6.14 The Deleted Block

The FFS also marks unused administration blocks as deleted to ensure that a disk scan does not confuse them with a used block. This change of the block type does not happen to data blocks of deleted files.

Table 13.22: Deleted Block

Long-word Offset	Content	Notes
0	`Type`	shall be 1, this is `T.DELETED`
1	`junk`	whatever remained here

This makes it particularly easy for disk salvage tools to identify which keys are actually still in active use and which have been scratched on purpose.

Chapter 14

Packet Documentation

Packets — or `DosPackets` by the name of the structure — are the mechanism by which the *dos.library* communicates with file systems and handlers, and by which the library delegates function to them. Many functions of the library are only a thin wrapper around this packet interface, and run internally into the `DoPkt()` function which creates and transmits such packets, then waits for the response of the handler and retrieves the result codes. Chapter **??** introduces this function, and also specifies in section **??** the structure of such packets.

When a handler receives a packet, it checks its `dp_Type` element for the requested action, retrieves the parameters from it, executes the requested function, then fills the packet with the results of this function and replies it to the originating process. Section **??** provides more details and also includes a minimal handler as an example.

This chapter documents the possible values of `dp_Type` and thus the possible requests the *dos.library* and the users of the direct packet interface can submit. It is most relevant for implementers of handlers and file systems that want to learn which packet types a handler is required to support. However, the list is also interesting for authors of application programs as they will find here packet types triggering functions that are only available through the direct packet interface, i.e. `DoPkt()` (or in general the functions of chapter **??**) and not through any other function of the *dos.library*.

While this list covers all packets currently defined by AmigaDOS, third party handlers may support additional packets; section **??** lists some generally known packets that are not part of AmigaDOS but have been used in the past, though this list is not necessarily complete. In general, `dp_Type` values between 2050 and 2999 are reserved for private purposes.

14.1 Packets for File Interactions

The packet types listed in this section implement functions of the *dos.library* that operate on or with files, such as the library functions listed in chapter **??**. They open and close files, read data from files or write data to files, or set the position of the file pointer. These packets shall be supported by almost any handler and are not exclusive to file systems. This section also list the packets implementing the record locking interface of section **??**, which is an extension made by AmigaDOS 36.

The arguments of the packets usually mirror the arguments of the corresponding *dos.library* functions closely, though are often represented by their BCPL equivalents, i.e. BPTRs instead of regular pointers or BSTRs instead of `NUL`-terminated C strings. The conversion of the C strings to BSTRs are performed transparently within the *dos.library*. Unfortunately, it means that even handlers written in Assembler or C need to deal with legacy BCPL data types.

14.1.1 Opening a File for Shared Access

The packet ACTION_FINDINPUT initializes a FileHandle structure for shared access to a file.

Table 14.1: ACTION_FINDINPUT

DosPacket Element	Value
dp_Type	ACTION_FINDINPUT (1005)
dp_Arg1	BPTR to FileHandle
dp_Arg2	BPTR to FileLock
dp_Arg3	BPTR to BSTR of the path
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is used by the Open() function of the *dos.library* which copies the packet type from its second argument, namely accessMode. The macro ACTION_FINDINPUT therefore expands to the same value as MODE_OLDFILE, see table **??** of section **??**.

dp_Arg1 is a BPTR to a FileHandle structure whose fh_Arg1 handle should be initialized to a value the handler or file system may use later on to identify the file opened by this packet, see also section **??** for the specification of the file handle. All other packets that operate on files do not include the FileHandle as one of their arguments, but instead the value placed in fh_Arg1. Interactive handlers, i.e. handlers that implement ACTION_WAIT_CHAR, shall also set the fh_Port element of the FileHandle to DOSTRUE to announce that they are interactive.

A handler or file system may also replace the fh_Type element of the FileHandle structure by a pointer to an alternative port subsequent requests related to this file will be delivered to. This is a second mechanism by which a file system or handler may disambiguate files. Otherwise, fh_Type keeps its original value, namely the port to which the request to open the file was delivered.

dp_Arg2 contains a BPTR to a FileLock structure identifying the directory from which the file system starts searching for the file to be opened. If this lock is ZERO, then the file system shall assume that the provided path is relative to the root directory of the currently inserted volume. Handlers that do not implement locks, in particular interactive handlers, ignore this argument. The Open() function obtains this lock through GetDeviceProc(), and it is typically the current directory of the calling process, or the lock of an assign taken from the file name argument, section **??** documents the details.

dp_Arg3 contains a BPTR to a BSTR providing the path including the name of the file to open. The path is relative to the directory in dp_Arg2, and the part beyond the colon (“:”) if it exists, or all of the path, shall be used by the handler to determine the file to open. For interactive handlers, its interpretation is up to the handler, see sections **??** or **??** for examples. The same holds for file systems, but it is advisable to follow the conventions of section **??**, namely to separate path components by forward slashes (“/”), and that the last component is the name of the file to open.

All other arguments of the packet shall be ignored. On success, the handler shall set dp_Res1 to DOSTRUE, otherwise to DOSFALSE for error. On error, dp_Res2 shall be set to an error code from dos/dos.h, see also section **??** for the list, or shall be set to 0 on success.

The purpose of this packet is to prepare a FileHandle structure for shared access, allowing both read and write operations. If a file system cannot locate the requested file, this packet *shall not* create it but rather fail with the error code ERROR_OBJECT_NOT_FOUND.

Handlers such as the Port-Handler or the CON-Handler may already open their resources as part of the startup-packet handling and thus may not need to perform a lot of activities here, except incrementing a use counter, see also section **??**; the lock in dp_Arg2 is irrelevant for them, and if the path was already interpreted during the handler startup, see section **??**, then dp_Arg3 may also be ignored. Interactive handlers shall be prepared to receive in dp_Arg3 the file name “⋆” or a path relative to CONSOLE: in

which case the newly created file shall use the resources of the already running handler process. In particular, the above paths create duplicates of an already open file, see also section **??**.

Because the CON-Handler does not initialize `dol_Task` of the `DosList` structure, each attempt to open a file through its device name, i.e. "`CON:`", "`RAW:`" or "`AUX:`", will request a new handler process; attempting to re-open the console by "`*`" or relative to "`CONSOLE:`" will instead re-use an already running handler.

This is different for file systems which do initialize `dol_Task` and thus operate from a single process. In this case `fh_Arg1` in the `FileHandle` structure shall be initialized as it is delivered to subsequent packets that operate on the file, and thus allows the file system to identify the file to work with. For file systems, both the lock provided in `dp_Arg2` and the path in `dp_Arg3` are required for locating the file to open. The path from `dp_Arg3` is logically appended to the path locked by `dp_Arg2`, i.e. the file system starts interpreting the former by scanning the directory hierarchy at the position given by `dp_Arg2`, or the root directory of the inserted volume if `dp_Arg2` is `ZERO`.

A special case arises if `dp_Arg3` is the empty string; in such a case, `dp_Arg2` shall be already a lock to the file to open. This is implied by the above algorithm: Walking a directory tree ends whenever the path terminates, and in this case it terminates right at the lock from `dp_Arg2`.

File systems shall be aware that the same file can be opened multiple times in shared access, and that shared access also includes write access to the file. It is therefore advisable to split information on the file into two objects: one local object that represents the state of the file as seen from each file handle, and a global file specific object that stores the properties of the file common to them. Local states of the file are, for example, the position of its file pointer which is different for each file handle, global states are for example the size of the file, i.e. its end-of-file position. A pointer to the local object could be placed in `fh_Arg1`, and it could contain a pointer to a global object that contains also a counter how often a file is accessed, and which is released at the point all file handles to the file are closed.

14.1.2 Opening a File for Exclusive Access

The packet `ACTION_FINDOUTPUT` initializes a `FileHandle` structure for exclusive access to a file that is either created if it does not exist, or replaces an existing file along with its contents.

Table 14.2: `ACTION_FINDOUTPUT`

DosPacket Element	Value
`dp_Type`	`ACTION_FINDOUTPUT (1006)`
`dp_Arg1`	BPTR to `FileHandle`
`dp_Arg2`	BPTR to `FileLock`
`dp_Arg3`	BPTR to BSTR of the path
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet is used by the `Open()` function of the *dos.library* which copies the packet type from its second argument, namely `accessMode`. The macro `ACTION_FINDOUTPUT` therefore expands to the same value as `MODE_NEWFILE`, see table **??** of section **??**.

The arguments of the packet are initialized as for `ACTION_FINDINPUT`, see section **??**, and the primary and secondary result codes `dp_Res1` and `dp_Res2` shall be set likewise. Also, file systems should deposit a unique value identifying the file in `fh_Arg1` of the file handle, and interactive handlers shall set `fh_Port` of the file handle to `DOSTRUE`.

For handlers that are not file systems, this packet may share the implementation with the packet type `ACTION_FINDINPUT`, as handlers typically do not interpret the access mode under which a resource is opened.

This is different for file systems: if the file identified by dp_Arg2 and dp_Arg3 does not exist, though all directories but the last component of the path in dp_Arg3 do exist, then this packet shall not fail. Instead, it shall create a file in the target directory under the name given by the last component of the path. In short, while this packet shall create non-existing files, it shall not attempt to create intermediate directories between the directory in dp_Arg2 and the file.

If the file identified by dp_Arg2 and dp_Arg3 already does exist, and it is not opened by any other file handle, the contents of the existing file is removed, and a new empty file is created in its place. If opened by other handles, this packet shall fail with ERROR_OBJECT_IN_USE. Regardless of whether an existing file was replaced or a new file created, this packet grands exclusive access, and the file cannot be opened by any other file handle until this handle is closed again.

What exactly happens if a file is replaced that is the target of hard links is implementation dependent. The FFS replaces the file content such that the links point to the new file. The RAM-Handler detaches the links such that the links point to the old content, and creates a new independent file of the new content.

14.1.3 Opening or Creating a File for Shared Access

The packet ACTION_FINDUPDATE initializes a FileHandle structure for shared access to a file, potentially creating the file if it does not yet exist.

Table 14.3: ACTION_FINDUPDATE

DosPacket Element	Value
dp_Type	ACTION_FINDUPDATE (1004)
dp_Arg1	BPTR to FileHandle
dp_Arg2	BPTR to FileLock
dp_Arg3	BPTR to BSTR of file name
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is used by the Open() function of the *dos.library* which copies the packet type from its second argument, namely accessMode. The macro ACTION_FINDUPDATE therefore expands to the same value as MODE_READWRITE, see table **??** of section **??**.

The arguments of the packet are initialized as for ACTION_FINDINPUT, see section **??**, except that the packet shall create the file in case it does not exist. The primary and secondary result codes dp_Res1 and dp_Res2 shall be set likewise. Also, file systems should deposit a unique value identifying the file in fh_Arg1 of the file handle, and interactive handlers shall set fh_Port of the file handle to DOSTRUE.

For handlers that are not file systems, this packet may share the implementation with the packet type ACTION_FINDINPUT, as handlers typically do not interpret the access mode under which a resource is opened.

For file systems, this packet works quite similar to ACTION_FINDINPUT described in section **??** except that it creates the file if it does not exist. If it does exists, its contents remain intact. In either case, the file is opened for shared access and other handles may read from or write to it in parallel.

The purpose of this packet changed, unfortunately, with AmigaDOS 36 and what is documented above is valid for this and all newer versions. The BCPL implementation of the OFS, and thus AmigaDOS 34 and before used this packet to open an existing file in exclusive mode, quite the reverse from what the packet does today. More details on this change is found in section **??**.

14.1.4 Opening a File from a Lock

The packet ACTION_FH_FROM_LOCK initializes a FileHandle structure from a lock on an existing file. Whether this access is shared or exclusive depends on the type of the lock. Upon success, the lock is absorbed into the FileHandle.

Table 14.4: ACTION_FH_FROM_LOCK

DosPacket Element	Value
dp_Type	ACTION_FH_FROM_LOCK (1026)
dp_Arg1	BPTR to FileHandle
dp_Arg2	BPTR to FileLock
dp_Res1	Boolean result code
dp_Res2	Error code

This packet type implements the OpenFromLock() function of the *dos.library*, see section **??**. It opens a file from a lock. For hierarchical file systems, this is close to an ACTION_FINDINPUT packet with dp_Arg3 set to an empty string, except that dp_Arg2 is absorbed. This packet is new in AmigaDOS 36.

To uniquely identify the file and resources associated to it, the handler may, and file systems should, place an identifier in the fh_Arg1 element of the FileHandle provided in dp_Arg1. This value is delivered as argument of subsequent packets operating on the file and allows file systems to identify the target to work on. More on this in section **??**. Interactive handlers shall also set fh_Port of the file handle to DOSTRUE.

When successful, the lock in dp_Arg2 is absorbed into the file and shall be released when the file handle is closed by ACTION_END, see **??**. It is no longer available to the process initiating this packet. In typical implementations, the lock or internal resources associated to the lock become part of the object placed in fh_Arg1 of the file handle.

Before replying the packet, dp_Res1 shall be set by the handler to DOSTRUE on success, or DOSFALSE on error. On success, dp_Res2 shall be set to 0, or to an error code from dos/dos.h otherwise.

14.1.5 Closing a File

The packet ACTION_END releases all file system internal resources related to a file and, if it was opened for exclusive access, makes it accessible again.

Table 14.5: ACTION_END

DosPacket Element	Value
dp_Type	ACTION_END (1007)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Res1	Boolean result code
dp_Res2	Error code

The *dos.library* uses this packet to implement the Close() function, see section **??**.

dp_Arg1 contains the fh_Arg1 element of the FileHandle structure corresponding to the file that is supposed to be closed. This value may be used by the file system or handler to uniquely identify the resources associated to the file, see section **??** for more information.

This packet shall write out any pending changes, close the file, release all resources associated to it and, in case the file was opened in exclusive mode, shall make the file accessible again. This packet *does not* release the FileHandle structure, it does not even get access to it — this step is performed by the *dos.library* if this packet indicates success.

If there are still any packets blocking on the file to be closed, e.g. an ACTION_READ on an interactive stream waiting for user input, such requests shall be canceled and replied as well.

Before replying to the packet, dp_Res1 shall be set to DOSTRUE on success or DOSFALSE on failure. On success, dp_Res2 shall be set to 0, or to an error code otherwise.

If this packet is not successful, the *dos.library* will not release the memory of the FileHandle structure, and as such, it remains available for future operations. Thus, indicating failure implies that the file and resources associated to it remain available, and clients of a file system or handler possibly continue to access it. It is unfortunately unclear how a client will react on a failed ACTION_END.

14.1.6 Reading from a File

The packet ACTION_READ reads data from a file system or handler and advances the file pointer accordingly.

Table 14.6: ACTION_READ

DosPacket Element	Value
dp_Type	ACTION_READ (82)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Arg2	Pointer to the buffer
dp_Arg3	Number of bytes to read
dp_Res1	Bytes read or -1
dp_Res2	Error code

This packet implements the Read() function of the *dos.library*, see **??**. Packet arguments contain the following data:

dp_Arg1 is a copy from fh_Arg1 of the FileHandle structure and may be used by the file system or handler to identify the file. Note that it is *not* the FileHandle itself; instead, the handler may create and insert an identifier of file into the FileHandle when opening it, see **??** for details.

dp_Arg2 is a pointer (not a BPTR) to the buffer to be filled.

dp_Arg3 is the number of bytes to read.

If a file system mounted on an exec device implements this packet by reading data from the device, it should check the supplied target buffer against de_Mask in the environment vector of the file system. If the buffer start or end address binary and'ed with the one's complement of the mask is non-zero, the file system should not attempt to directly fill the buffer from the underlying device. Instead, it should first read the data into a buffer allocated with the memory requirements from de_BufMemType, and copy the buffer contents manually to the target buffer. Also, file systems should never read more than de_MaxTransfer bytes at once, and potentially break up the transfer into multiple reads. These workarounds are unfortunately necessary for some broken device implementations, see section **??** for further details.

If successful, the file pointer of the file shall be advanced by the number of bytes that could be read from it. On an error, the file pointer shall remain unaltered.

Before replying this packet, the handler shall fill dp_Res1 with the number of bytes that could be transferred to the buffer, or -1 for an error. The number of read bytes may also be 0 if no data could be transferred, for example because the end-of-file has been reached.

Unlike file systems, interactive handlers typically block if no data can be read, i.e. do not respond to the packet immediately. The packet is replied as soon as at least a single byte becomes available. They then deliver at most dp_Arg3 bytes of the data that became available, even if this is less data than requested. Details, however, depend on the handler and its configuration; the Port-Handler, for example, can be configured such that it does not block if no data is available on the serial port, but to return immediately with 0 bytes in the buffer, see section **??**. An alternative approach to prevent blocking for an undetermined time span that shall be supported by all interactive handlers is ACTION_WAIT_CHAR specified in section **??**. The latter does not remove any bytes from the stream, they remain available for a subsequent ACTION_READ.

In case of an error, i.e. if the primary result code is -1, dp_Res2 shall be filled with an error code from dos/dos.h or the list from section **??**, otherwise it shall be set to 0.

Note that there are no separate packet types corresponding to the buffered IO functions from section **??**. Instead, the *dos.library* functions at the caller side manage the buffer, monitor its fill state and potentially call into Read() which then generates this packet.

14.1.7 Writing to a File

The packet ACTION_WRITE writes data to a file system or handler and advances the file pointer accordingly.

Table 14.7: ACTION_WRITE

DosPacket Element	Value
dp_Type	ACTION_WRITE (87)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Arg2	Pointer to the buffer
dp_Arg3	Number of bytes to write
dp_Res1	Bytes written or -1
dp_Res2	Error code

This packet implements the Write() function of the *dos.library*, see **??**. The elements of the packet are populated as follows:

dp_Arg1 is a copy from fh_Arg1 of the FileHandle structure and may be used by the file system or handler to identify the file.

dp_Arg2 is a pointer (not a BPTR) to the buffer containing the data to be transferred.

dp_Arg3 is the number of bytes to write.

If a file system mounted on an exec device implements this packet by writing data to the device, it should check the supplied target buffer against de_Mask in the environment vector of the file. If the buffer start or end address binary and'ed with the one's complement of the mask is non-zero, the file system should not attempt to directly write the buffer to the underlying device. Instead, it should first manually copy the data into a buffer allocated with the memory requirements from de_BufMemType, and then write the buffer contents to the device. Also, file systems should never write more than de_MaxTransfer bytes at once, and potentially break up the transfer into multiple writes. These workarounds are unfortunately necessary for some broken device implementations, see section **??** for further details.

If successful, the file pointer of the file shall be advanced by the number of bytes that could be written. On an error, the file pointer shall remain unaltered.

Before replying this packet, the handler shall fill dp_Res1 with the number of bytes that could be transferred from the buffer, or -1 for an error condition. In case of an error, i.e. for the primary result code -1, dp_Res2 shall be filled with an error code, otherwise it shall be set to 0.

Note that there are no separate packet types corresponding to the buffered IO functions from section **??**. Instead, the *dos.library* functions at the caller side manage the buffer, monitor its fill state and potentially call into Write() which then generates this packet.

14.1.8 Adjusting the File Pointer

The packet ACTION_SEEK sets the file pointer relative to a base location.

Table 14.8: ACTION_SEEK

DosPacket Element	Value
dp_Type	ACTION_SEEK (1008)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Arg2	File pointer offset
dp_Arg3	Seek mode from Table **??** in **??**
dp_Res1	Previous file position
dp_Res2	Error code

This packet implements, to a major degree, the `Seek()` function of the *dos.library*, see **??**. The elements of the packet contain the following data:

`dp_Arg1` is a copy from `fh_Arg1` of the `FileHandle` structure and may be used by the file system or handler to identify the file.

`dp_Arg2` defines the new location of the file pointer relative to a position identified by `dp_Arg3`, it comes from the second argument of `Seek()`. This is an offset that may be positive or negative.

`dp_Arg3` defines the position to which `dp_Arg2` is relative. It is one of the modes in table **??** of section **??** and is therefore identical to the third argument of `Seek()`.

If the resulting position of the file handle as computed from `dp_Arg2`, its current value and the mode from `dp_Arg3` is outside of the file, i.e. either negative or beyond the end-of-file, the attempt shall fail with the error code `ERROR_SEEK_ERROR`.

Before replying this packet, the handler shall fill `dp_Res1` with the previous value of the file pointer, i.e. before making the requested adjustment, or to -1 in case an error occurred. In the latter case, `dp_Res2` shall be filled with an error code, otherwise it shall be set to 0. In case of an error, the file pointer shall remain unaltered.

If a handler, for example an interactive handler, does not implement this packet, it shall still identify it and set `dp_Res1` to -1 — *and not 0* — and `dp_Res2` to `ERROR_ACTION_NOT_KNOWN` as this allows users to identify the error, see also table **??** in section **??**.

Unfortunately, it is not clearly defined how this packet should interact with files that are larger than 2GB. A suggested implementation strategy is to interpret `dp_Arg2` as unsigned 32 bit offset for the mode `OFFSET_BEGINNING` and extend it to 64 bits, and sign-extend it for `OFFSET_CURRENT` to 64 bits. For the mode `OFFSET_END`, it is suggested to first zero-extend the 32 bit negative of `dp_Arg2` to 64 bits, and then negate it in 64 bits. The 64 bit offset formed by the above steps is either the new file pointer, or should be added to the 64-bit file pointer or the 64-bit file size to form the new file pointer. The 32 least significant bits of the of the current 64-bit file pointer shall be returned in `dp_Res1`. Unfortunately, that leaves it to the client to check `dp_Res2` in case the result is -1 and thus to distinguish a successful seek from an error condition.

While this packet type implement the core of the `Seek()` function, the latter is also aware of the (caller-side) buffer of the `FileHandle` and performs potentially a flush of this buffer. The packet cannot, of course, perform this step as it does not have access to the buffer of the `FileHandle`.

14.1.9 Setting the File Size

The packet `ACTION_SET_FILE_SIZE` adjusts the size of a file, either truncating it or extending it beyond its current end of file.

Table 14.9: `ACTION_SET_FILE_SIZE`

DosPacket Element	Value
`dp_Type`	`ACTION_SET_FILE_SIZE (1022)`
`dp_Arg1`	`fh_Arg1` of the `FileHandle`
`dp_Arg2`	File size adjustment
`dp_Arg3`	Mode from Table **??** in **??**
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet implements the `SetFileSize()` function of the *dos.library*, see **??**. This packet was introduced in AmigaDOS 36. The elements of the packet contain the following data:

`dp_Arg1` is a copy from `fh_Arg1` of the `FileHandle` structure and may be used by the file system or handler to identify the file.

dp_Arg2 defines the new size of the file, or equivalently the new position of the end of file, relative to a position identified by dp_Arg3. This offset is coming from the second argument of SetFileSize().

dp_Arg3 defines the position dp_Arg2 is relative to. It is one of the modes in table **??** of section **??** and is therefore identical to the third argument of SetFileSize(). The new end-of-file position of the file can therefore be set relative to the start of the file, i.e. dp_Arg2 is the new size of the file, relative to the current file pointer, or relative to the current end-of-file.

When truncating a file, all contents beyond the new end of file are lost. When extending a file, the contents of the file in the extended region is *undefined*. Unlike other operating systems, AmigaDOS does not enforce zero-initialization of the extended region. The file pointers of all file handles accessing this file shall be clamped to the new file size if necessary, but shall remain unaltered otherwise.

Unfortunately, it is not clearly defined how this packet should interact with files that are larger than 2GB. A suggested implementation strategy is to interpret dp_Arg2 as unsigned 32 bit offset for the mode OFFSET_BEGINNING and extend it to 64 bits, and sign-extend it for OFFSET_CURRENT to 64 bits. For the mode OFFSET_END, it is suggested to first zero-extend the 32 bit negative of dp_Arg2 to 64 bits, and then negate it in 64 bits. The 64 bit offset formed this way is either the new file size directly, or shall be added to the current file pointer or the current file size to obtain the target file size.

Before replying this packet, the handler shall fill dp_Res1 with a Boolean success indicator, DOSTRUE in case it could extend or truncate the file as requested, or DOSFALSE in case of error. In case of success, dp_Res2 shall be set to 0, otherwise it shall be set to an error code.

If a handler, for example an interactive handler, does not implement this packet, it shall still identify it and set dp_Res1 to -1 — *and not 0* — and dp_Res2 to ERROR_ACTION_NOT_KNOWN as this allows users to identify the error, see also table **??** in section **??**.

Additional information on this packet is found in section **??** which describes the *dos.library* function that is based on it.

14.1.10 Locking a Record of a File

The packet ACTION_LOCK_RECORD locks a record of a file. Such a record lock neither prevents reading from nor writing to the locked region, it just prevents locking an overlapping region with a conflicting lock, see section **??** for details how the protocol is supposed to be used.

Table 14.10: ACTION_LOCK_RECORD

DosPacket Element	Value
dp_Type	ACTION_LOCK_RECORD (2008)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Arg2	Start offset of record
dp_Arg3	Length of record
dp_Arg4	Type of lock from table **??** in **??**
dp_Arg5	Timeout (if applicable) in ticks
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the LockRecord() function of the *dos.library* and attempts to lock a record of a file, given as start offset from the beginning of the file and a byte size. There is no packet corresponding to LockRecords(). Instead, the latter locks records sequentially. This packet was added in AmigaDOS 36. Arguments of the packet are populated as follows:

The file is identified by dp_Arg1 which is a copy from fh_Arg1 of the FileHandle structure[1].

[1]The information on dp_Arg1 in [**?**] is incorrect.

The record to lock is identified by dp_Arg2 and dp_Arg3 which correspond to the offset and length arguments of the LockRecord() function.

The mode by which the record is supposed to be locked is given by dp_Arg4, it identifies whether the access is shared or exclusive, and whether a timeout is applied or not. The mode is a value from table **??** in section **??**, and more information on the modes is found there.

If the mode from dp_Arg4 includes a timeout, see table **??**, the file system shall wait at most dp_Arg5 ticks for the record to become available. Otherwise, dp_Arg5 is ignored and the packet shall fail immediately if locking is not possible.

If locking the record is possible, or possible within the timeout, then dp_Res1 shall be set to DOSTRUE when replying this packet. In such a case, dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code. If a timeout is provided, and the lock could not be obtained within the timeout because an overlapping region is already locked by a conflicting record lock, the file system shall signal ERROR_LOCK_TIMEOUT as error code. If no timeout was given and the lock could not be obtained, the the error code ERROR_LOCK_COLLISION shall be signaled instead.

14.1.11 Release a Record of a File

The packet ACTION_FREE_RECORD releases a record lock on a record of a file.

Table 14.11: ACTION_FREE_RECORD

DosPacket Element	Value
dp_Type	ACTION_FREE_RECORD (2009)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Arg2	Start offset of record
dp_Arg3	Length of record
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the UnLockRecord() function of the *dos.library* and releases a lock on a subset of a file, identified by start position and length. The UnLockRecords() function is not implemented by a separate packet; instead, the *dos.library* sequentially calls UnLockRecord() for each record. This packet was introduced in AmigaDOS 36. Arguments of the packet are populated as follows:

The file is identified by dp_Arg1 which is a copy from the fh_Arg1 element of the FileHandle structure.

The record to release is given by the start offset within the file provided by dp_Arg2 and the byte size of the record in dp_Arg3.

On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 to 0. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code. If, in particular, the specified region is not locked, this packet shall fail with ERROR_RECORD_NOT_LOCKED as error code.

14.2 Packets for Interacting with Locks

The packets listed in this section implement the *dos.library* functions listed in chapter **??**; they create, release or duplicate locks, or create directories returning a lock. A lock is an abstract representation of a file system object, granting shared or exclusive access to it. If the lock represents a directory, it can be used to replace the current directory of a process. If it represents a file, a file handle can be opened from it. Locks can also be used to retrieve metadata such as comments or protection bits of a file system object. Locks never represent soft- or hard links, they are resolved when obtaining the lock, and the lock then represents the target of the link.

14.2.1 Obtaining a Lock

The ACTION_LOCATE_OBJECT packet locates an object — a file or a directory — on a file system and creates from it a lock identifying the object uniquely.

Table 14.12: ACTION_LOCATE_OBJECT

DosPacket Element	Value
dp_Type	ACTION_LOCATE_OBJECT (8)
dp_Arg1	BPTR to FileLock
dp_Arg2	BPTR to BSTR of the object name
dp_Arg3	Mode of the lock
dp_Res1	BPTR to FileLock
dp_Res2	Error code

This packet creates a lock from a path and a directory to which this path is relative, also represented by a lock. As such, this packet implements the Lock() function of the *dos.library* as found in section **??**.

dp_Arg1 is a BPTR to a FileLock structure that represents the object to which the path given in dp_Arg2 is relative. If dp_Arg1 is ZERO, the path in dp_Arg2 is relative to the root of the currently inserted volume. The Lock() function determines this lock from the GetDeviceProc() function, which delivers for example the current directory of the caller or the lock of an assign identified from the path.

dp_Arg2 is a BPTR to a BSTR of an absolute path or a path relative to dp_Arg1 of the object to be locked. That is, the location of the object to be locked is logically formed by appending the path in dp_Arg2 to the directory identified by the lock in dp_Arg1. If the string in dp_Arg2 is empty, the object to be locked is identical to the object locked by dp_Arg1.

dp_Arg3 is the type of the lock to be created as defined in table **??** in section **??**. The value SHARED_LOCK requests a non-exclusive lock on an object; multiple shared locks can be held on the same object. The value EXCLUSIVE_LOCK requests an exclusive lock. Attempting to exclusively lock an object that is already locked shall fail, and attempting to lock an object that is already exclusively locked shall fail as well[2]. Unfortunately, some programs call Lock() with an invalid value in accessMode, and file systems should be prepared to receive such values in dp_Arg3. They should be considered equivalent to SHARED_LOCK.

As opening a file implies getting access rights on the file to be opened similar to locks, an exclusively locked file cannot be opened in any mode from its name, but only by ACTION_FH_FROM_LOCK. A file locked in a shared mode cannot be replaced by a packet of type ACTION_FINDOUTPUT, but is accessible in any other mode. A shared lock does therefore not protect the file contents from getting changed, but the file from getting replaced. An exclusively locked directory cannot be deleted, but objects within it can be created, deleted or changed.

If a lock can be granted, the file system shall create a FileLock structure representing the locked object. Such structures are allocated, maintained and released by file systems and not the *dos.library*. The FileLock shall be initialized as follows (see also **??**):

fl_Task shall point to a MsgPort through which the maintaining file system can be contacted, which is typically, but not necessarily the process message port pr_Port of the file system itself.

fl_Volume shall be a BPTR to the DosList structure representing the volume on which the locked object is located, see chapter **??**.

fl_Access shall be filled with dp_Arg3, identifying the type of the lock.

fl_Link may be used to queue up locks on a volume that is currently not accessible. As locking an object on a non-inserted volume is not possible, this element is only filled when a volume containing locked objects is ejected or becomes otherwise inaccessible. The file system will then store all such locks in a singly

[2] As the root directory is always also represented as ZERO lock, an exclusive lock on this directory is probably not very exclusive, which is why [**?**] suggests not to permit exclusive locks on the volume root.

linked list starting at dol_LockList of the DosList and chained by fl_Link. If the same volume is then re-inserted into another drive, another instance of the same file system is able to pick up these locks and manages them instead; this requires patching fl_Task to point to the port of the file system that takes over the lock. Thus, for example, a lock on an object in drive DF0 will remain valid even if the volume is removed and re-inserted into DF1.

fl_Key may be used for the purpose of the file system for uniquely identifying the locked object. It is an opaque value as far as the *dos.library* is concerned.

On success, the BPTR to the created FileLock shall be stored in dp_Res1 and dp_Res2 shall be set to 0. On error, dp_Res1 shall be set to ZERO and dp_Res2 to an error code from dos/dos.h identifying the source of the problem, see also section **??** for a list of error codes.

14.2.2 Duplicating a Lock

Despite its misguiding name, the packet ACTION_COPY_DIR creates a copy of a (shared) lock, regardless of whether it locks a directory or a file. FileLocks shall *not* be copied by making a byte-wise (shallow) copies of their memory representation.

Table 14.13: ACTION_COPY_DIR

DosPacket Element	Value
dp_Type	ACTION_COPY_DIR (19)
dp_Arg1	BPTR to FileLock
dp_Res1	BPTR to FileLock
dp_Res2	Error code

This packet implements the DupLock() function of the *dos.library*, see section **??**, and attempts to replicate the lock passed in dp_Arg1. If this argument is ZERO, the packet shall create a lock on the root directory of the currently inserted volume.

On success, dp_Res1 is filled with the BPTR to a copy of the FileLock passed in, and dp_Res2 is set to 0. On error, dp_Res1 is set to ZERO and dp_Res2 to an error code. While [**?**] indicates that dp_Res1 may be ZERO on an attempt to replicate the ZERO lock, file systems shall instead create a shared lock on the root directory as otherwise application programs may misinterpret the result as failure.

This packet is identical to ACTION_LOCATE_OBJECT with dp_Arg3 set to SHARED_LOCK and dp_Arg2 set to an empty string.

14.2.3 Finding the Parent of a Lock

The packet ACTION_PARENT obtains a shared lock on the directory containing a locked object.

Table 14.14: ACTION_PARENT

DosPacket Element	Value
dp_Type	ACTION_PARENT (29)
dp_Arg1	BPTR to FileLock
dp_Res1	BPTR to FileLock
dp_Res2	Error code

This packet implements the ParentDir() function of the *dos.library* (see section **??**) and attempts to create a (shared) lock on the directory containing the object identified by dp_Arg1. If no parent exists because dp_Arg1 is a lock on the root directory or the ZERO lock, the file system shall set dp_Res1 to ZERO and dp_Res2 to 0, indicating that this is not a failure case. Note that dp_Arg1 may also be the lock on a file in which case a lock on the containing directory shall be created.

This packet is subtly different from attempting to lock the path "/" relative to the lock dp_Arg1. The latter fails with an error code ERROR_OBJECT_NOT_FOUND on the attempt to find the parent of the root directory. In all other cases, the two approaches to lock the parent directory work alike.

On success, this packet shall fill dp_Res1 with a BPTR to a FileLock representing the parent directory of dp_Arg1, or ZERO if dp_Arg1 is a lock on the root directory. dp_Res2 shall be set to 0 on success. On error, dp_Res1 is set to ZERO and dp_Arg2 to an error code.

14.2.4 Duplicating a Lock from a File Handle

The packet ACTION_COPY_DIR_FH creates a shared lock from an opened file represented by a file handle; that is, it locks the object accessed by the handle.

Table 14.15: ACTION_COPY_DIR_FH

DosPacket Element	Value
dp_Type	ACTION_COPY_DIR_FH (1030)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Res1	BPTR to FileLock
dp_Res2	Error code

This packet implements the DupLockFromFH() function of the *dos.library*, see **??**. It was added in AmigaDOS 36. Creating a lock from a file handle works only if the file has been opened in a non-exclusive mode, i.e. either in MODE_READWRITE or in MODE_OLDFILE, and results in a shared lock on the opened file.

dp_Arg1 is a copy of the fh_Arg1 element of the FileHandle structure, see section **??**, and thus serves to identify the opened file. This handle shall remain unchanged and usable after this packet returns.

On success, dp_Res1 shall be filled by the BPTR to the FileLock created. In such a case, dp_Res2 shall be set to 0. On error, dp_Res1 shall be set to ZERO and dp_Res2 to an error code.

14.2.5 Finding the Parent Directory of a File Handle

The packet ACTION_PARENT_FH obtains a lock on the directory containing a file identified by a file handle.

Table 14.16: ACTION_PARENT_FH

DosPacket Element	Value
dp_Type	ACTION_PARENT_FH (1031)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Res1	BPTR to FileLock
dp_Res2	Error code

This packet implements the ParentOfFH() function of the *dos.library*, see section **??**, and as such creates a shared lock on the directory that contains the file opened to the provided file handle. Same as the function, this packet was added in AmigaDOS 36.

dp_Arg1 is a copy of the fh_Arg1 element of the FileHandle structure and thus identifies the file whose parent directory is to be locked.

Unlike ACTION_COPY_DIR_FH, this packet does not fail if the file was opened in exclusive mode. It will, of course, fail if the parent directory is exclusively locked.

On success, dp_Res1 is filled with a BPTR to the FileLock created, and dp_Res2 is set to 0. On error, dp_Res1 is set to ZERO and dp_Res2 to an error code.

14.2.6 Creating a new Directory

The packet ACTION_CREATE_DIR creates a new directory and returns an exclusive lock to it.

Table 14.17: ACTION_CREATE_DIR

DosPacket Element	Value
dp_Type	ACTION_CREATE_DIR (22)
dp_Arg1	BPTR to FileLock
dp_Arg2	BPTR to BSTR of the directory name
dp_Res1	BPTR to FileLock
dp_Res2	Error code

This packet implements the CreateDir() function of the *dos.library*, see section **??**. The arguments are as follows:

The lock in dp_Arg1 is the directory to which the path in dp_Arg2 is relative. The CreateDir() function obtains it from the GetDeviceProc() function applied to the path.

The path in dp_Arg2 is logically concatenated to the directory represented by the lock in dp_Arg1, or is relative to the root directory of the currently inserted volume if dp_Arg1 is ZERO. The name of the directory to be created is given by the last component of the path in dp_Arg2, all components in the path upfront must exist or the packet shall fail with the error ERROR_OBJECT_NOT_FOUND, i.e. this packet shall not attempt to create directories recursively.

If the path in dp_Arg2 describes an already existing file system object, that is, if the target already exists as a file or directory, this packet shall fail with the error code ERROR_OBJECT_EXISTS.

On success, this packet returns an exclusive lock to the directory created in dp_Res1, and dp_Res2 is set to 0. On error, dp_Res1 is set to ZERO and dp_Res2 is set to an error code.

14.2.7 Comparing two Locks

The packet ACTION_SAME_LOCK compares two locks on the same file system and checks whether they lock the same object.

Table 14.18: ACTION_SAME_LOCK

DosPacket Element	Value
dp_Type	ACTION_SAME_LOCK (40)
dp_Arg1	BPTR to FileLock
dp_Arg2	BPTR to FileLock
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is the core of the SameLock() function of the *dos.library* (see **??**), which, however, first checks whether the locks to compare are on the same file system. Only if so, the file system corresponding the two locks is contacted with this packet to compare them. This packet was introduced in AmigaDOS 36.

The arguments dp_Arg1 and dp_Arg2 are BPTRs to the two FileLocks to compare.

Upon reply, dp_Res1 shall be set to DOSTRUE if the two locks are on the same object, and in that case, dp_Res2 shall be set to 0. If the two locks are on two different objects, then dp_Res1 shall be set to DOSFALSE and dp_Res2 shall also be set to 0. On error, dp_Res1 is set to 0 and dp_Res2 to an error code[3].

If the file system does not implement this packet, the *dos.library* SameLock() function instead compares the fl_Key elements of the two locks, and assumes that the locks point to the same object if their keys are identical.

[3]The documentation in [**?**] on dp_Res1 is incorrect.

14.2.8 Changing the Mode of a Lock or a File Handle

The packet ACTION_CHANGE_MODE changes the access mode of a lock or a file handle, thus potentially granting exclusive access — if possible — or lowering the rights to shared access.

Table 14.19: ACTION_CHANGE_MODE

DosPacket Element	Value
dp_Type	ACTION_CHANGE_MODE (1028)
dp_Arg1	Object type from table **??** in **??**
dp_Arg2	BPTR to FileLock or FileHandle
dp_Arg3	Target mode
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the ChangeMode() function of the *dos.library* and shall therefore change the access mode of files or locks to exclusive or shared access mode. This packet was added in AmigaDOS 36.

dp_Arg1 identifies the object whose mode is to be changed. This is a value from table **??** in section **??**, i.e. either CHANGE_LOCK to adjust the type of a lock, or CHANGE_FH to change the access mode of a file handle.

dp_Arg2 is the BPTR to the object whose mode is to be changed. If dp_Arg1 is CHANGE_LOCK, this is a BPTR to a FileLock structure, if dp_Arg1 is CHANGE_FH, it is a BPTR to a FileHandle structure. Note that this is unusual as files are typically identified by their fh_Arg1 element and not by the handle itself. This packet is an exception.

dp_Arg3 is the target mode. This is either SHARED_LOCK for shared access to the file or the lock, or EXCLUSIVE_LOCK for exclusive access to the file or the lock[4]. However, as information on this packet is sparse and application programs can call the corresponding ChangeMode() function with incorrect target modes, file systems should also accept ACTION_FINDINPUT and ACTION_FINDUPDATE for shared access and ACTION_FINDOUTPUT for exclusive access.

Note that it is not always possible to change the mode of a lock or a file to exclusive access, namely if it is accessed by a second file handle or locked a second time. In such a case, this packet shall fail.

Upon reply, dp_Res1 shall include a Boolean success indicator, DOSTRUE for success or DOSFALSE for failure. In the former case, dp_Res2 shall be 0, in the latter case, it shall contain an error code.

14.2.9 Releasing a Lock

The packet ACTION_FREE_LOCK releases a lock on a file system object.

Table 14.20: ACTION_FREE_LOCK

DosPacket Element	Value
dp_Type	ACTION_FREE_LOCK (15)
dp_Arg1	BPTR to FileLock
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the *dos.library* function UnLock() (see **??**) and releases a FileLock. A BPTR to the lock to release is provided in dp_Arg1. If this argument is ZERO, the file system shall ignore the request and return success.

[4]The information in [**?**] is incorrect on this subject matter.

As first step of precaution, file systems should zero the `fl_Task` element of the `FileLock` structure because the *dos.library* ignores attempts to unlock such locks. This improves system stability a bit if applications erroneously attempt to release a lock twice. In a second step, file system shall release all internal resources related to the lock, until finally the `FileLock` structure itself shall be disposed; it is *not* allocated or released by the *dos.library*.

If the volume the locked object is located on is currently not inserted, and the lock has not been claimed by another file system, i.e. its `fl_Task` element still points to a port of this file system and it is still in the `dol_LockList`, the attempt to unlock the lock shall be satisfied, and the lock shall be removed from this list and then released.

When replying this packet, `dp_Res1` shall be set to `DOSTRUE` on success and `DOSFALSE` on error. On success, `dp_Res2` shall be set to 0, otherwise it shall be set to an error code. There are actually only few reasons why this packet could fail, probably because the passed in BPTR is invalid and does not point to a valid `FileLock`. The `UnLock()` function does not return this result code, and also ignores `dp_Res2`. Thus, the *dos.library* has no means to forward such errors to its callers.

14.3 Packets for Examining Objects

The packets in this section implement the functions of section **??** on packet level, i.e. they retrieve from the file system information on objects identified by locks. They also allow to iterate over directories and receive information from all objects within. Unfortunately, implementing the packets in this section in a robust way is nontrivial as directory contents can change while iterating over it.

14.3.1 Examining a Locked Object

The `ACTION_EXAMINE_OBJECT` packet retrieves information such as file name, comment and protection bits from a lock on an object and fills a `FileInfoBlock` with this data.

Table 14.21: `ACTION_EXAMINE_OBJECT`

DosPacket Element	Value
`dp_Type`	`ACTION_EXAMINE_OBJECT (23)`
`dp_Arg1`	BPTR to `FileLock`
`dp_Arg2`	BPTR to a `FileInfoBlock` structure
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet implements the `Examine()` function of the *dos.library*, see section **??**. It fills the `FileInfoBlock` structure pointed to by `dp_Arg2` with information on the object identified by the lock in `dp_Arg1`. See section **??** for the specification of this structure.

However, as the AmigaDOS handler design is based on BCPL, small differences exist between how this packet operates and how the `Examine()` function provides its results to the caller. In specific, the `fib_FileName` and `fib_Comment` elements shall be filled with a BSTRs, i.e. the first character is the length of the file name or the comment. The conversion to a `NUL`-terminated C string is performed by the *dos.library* external to the file system.

The elements `fib_DirEntryType` and `fib_EntryType` shall be filled with a value from table **??** in section **??**, identifying the type of the object. As programs are not consistent on the element they read, both should be filled with the same value. The value 0 should be avoided as some programs identify directories by testing against a positive value whereas others check for a non-negative value, i.e. interpret 0 as a directory.

The fields `fib_OwnerUID` and `fib_OwnerGID` shall be set to 0, unless the file system has an idea on user and group IDs — unfortunately AmigaDOS cannot make use of these values anyhow, and it is not

documented how these fields shall be interpreted. fib_Reserved shall be left alone, in particular file systems *shall not* use it to store internal state information. Such information may *only* go into fib_DiskKey, which is an element application programs, i.e. callers of the *dos.library*, shall not interpret.

If dp_Arg1 is ZERO, then the root directory of the currently inserted volume shall be examined. In such a case, fib_FileName shall be filled with the name of the volume.

This packet is also used to initiate a scan over a directory to examine all objects within in it. Then dp_Arg1 is the lock of the directory to be scanned. File systems may use this packet to initialize internal state information stored in this lock, section **??** provides more information why this might be necessary. Depending on the file system, the ZERO lock may then be not suitable to scan the contents of the root directory, in which case applications need to obtain a lock on "**:**" explicitly.

This function returns a Boolean success code in dp_Res1, it shall be either set to DOSTRUE in case information on the locked object could be retrieved and was inserted into the FileInfoBlock given by dp_Arg2. In case of success, dp_Res2 shall be set to 0. In case of an error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.3.2 Scanning Directory Contents

The ACTION_EXAMINE_NEXT continues a scan over the directory contents and delivers information on the next subsequent entry in a directory. Such a scan is started by an ACTION_EXAMINE_OBJECT on the directory to be scanned; the first ACTION_EXAMINE_NEXT then provides information on the first entry in this directory and each subsequent packet to the corresponding next entry.

Table 14.22: ACTION_EXAMINE_NEXT

DosPacket Element	Value
dp_Type	ACTION_EXAMINE_NEXT (24)
dp_Arg1	BPTR to FileLock
dp_Arg2	BPTR to a FileInfoBlock structure
dp_Res1	Boolean result code
dp_Res2	Error code

This function continues a directory scan by providing information on the next subsequent object in the directory identified by the lock in dp_Arg1. Information on the object is filled into the FileInfoBlock structure pointed to by the BPTR in dp_Arg2. Similar to ACTION_EXAMINE_OBJECT discussed in section **??**, the file name and comment shall be provided as BSTRs and not as NUL-terminated C strings. Conversion to the latter format is performed within the *dos.library*.

Depending on the file system, it may be impossible to scan the root directory through the ZERO lock in dp_Arg1. Failing with ERROR_INVALID_LOCK is permissible in such a case.

Unlike Unix like file systems, AmigaDOS does not keep entries in directories that correspond to the directory itself or its parent directory, i.e. AmigaDOS file systems do not carry "**.**" or "**..**" directory entries. Even if alien file system structures contain such entries, they should not be accessible by this packet and thus be hidden from AmigaDOS applications.

Unfortunately, this packet is one of the hardest to implement as directory contents can change between two ACTION_EXAMINE_OBJECT packets. In particular, file systems shall handle the situation gracefully in which the object from the previous iteration identified by fib_DiskKey has been deleted, moved or replaced by another object, or in which dp_Arg1 is a different lock than the lock that was used to start the scan by ACTION_EXAMINE_OBJECT. The file system may only assume that the lock in dp_Arg1 is a lock on the same directory on which the scan has been started. Similarly, dp_Arg2 may have changed from the last iteration, and the file system shall only depend on fib_DiskKey and fib_DirEntryType being identical compared to the last iteration, all other entries can be potentially trashed or inconsistent. Thus, the file system shall only use these two elements of the FileInfoBlock to store state information. Also,

file systems shall not depend on application programs scanning directories up to the last entry; application programs can abort a directory scan at an arbitrary point, yet file system shall release all resources required for storing state information of the scan after at least the lock on the scanned directory is released.

A possible implementation strategy is to store full state information in the lock given by dp_Arg1, though keep sufficient information in fib_DiskKey to rebuild the full information in case the lock is released and replaced during the scan. In the simplest possible case, fib_DiskKey could be a counter that enumerates the elements in the directory, whereas the lock contains the full state information to access the element directly. In case the lock is replaced, ACTION_EXAMINE_NEXT would find the state information in the lock missing, and would then scan forward to the element enumerated by fib_DiskKey. While this is less efficient than using the (now missing) state information in the lock, it will at least provide information on a valid object in the directory. Also, if an object is removed or moved out of the scanned directory, the file system would update the state information kept within the lock to the directory, though loosing the lock would at least continue the scan within the directory, even though not necessary from the same object. Storing a block number (for disk-based file systems) or a pointer to an object (for RAM-based file systems) is, however, a bad strategy as the corresponding disk block or RAM location may have found other uses at the time the next object is examined. Unfortunately, such implementation defects are rather common and have been found in multiple file systems of AmigaDOS in the past.

This packet shall provide a Boolean success code in dp_Res1. In case the next object in a directory could be successfully examined and information on it could be stored in FileInfoBlock, dp_Res1 shall be set to DOSTRUE and dp_Res2 to 0. In case of an error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code. In case the end of the directory has been reached and no further entries remain to be examined, this error code shall be ERROR_NO_MORE_ENTRIES.

14.3.3 Examining Multiple Entries at once

The ACTION_EXAMINE_ALL packet scans a directory supplying multiple entries at once, potentially filtering them through a pattern or through a hook.

Table 14.23: ACTION_EXAMINE_ALL

DosPacket Element	Value
dp_Type	ACTION_EXAMINE_ALL (1033)
dp_Arg1	BPTR to FileLock
dp_Arg2	APTR to a buffer to fill
dp_Arg3	Size of the buffer
dp_Arg4	Type defining the requested entries
dp_Arg5	Pointer to ExAllControl structure
dp_Res1	Continuation flag
dp_Res2	Error code

This packet implements the ExAll() function of the *dos.library*, see section **??**, and thus takes parameters similar to the above function. Note that unlike many other packets, dp_Arg2 and dp_Arg5 are regular pointers and not BPTRs[5].

This packet was introduced in AmigaDOS 36, but if the file system does not implement it, i.e. returns ERROR_ACTION_NOT_KNOWN, then the *dos.library* emulates it using ACTION_EXAMINE_OBJECT and ACTION_EXAMINE_NEXT.

Packet arguments are as follows:

dp_Arg1 is a lock on the directory to be examined. This lock cannot be ZERO by the requirements of ExAll(), see section **??**.

[5]The information on dp_Arg5 in [**?**] is incorrect, it is really a pointer and not a BPTR

dp_Arg2 is a pointer to a word aligned buffer to be filled with information on the objects found in the directory, it is *not* a BPTR. This buffer shall be filled with a singly linked list of ExAllData structures, see again section **??** for the definition of this structure. Only the elements requested by dp_Arg4 are filled. Remaining entries are undefined and need not to be present at all, i.e. only the minimal amount of information requested may be filled in, using the unused elements to store subsequent ExAllData structures.

In addition, file systems shall align the ExAllData in the buffer to word boundaries to allow their interpretation also by the less capable members of the 68K family; that is, the ed_Next element pointing to a subsequent structure shall be even. For the last filled structure, this pointer shall be set to NULL to terminate the list, even though the number of structures filled shall also be made available in the eac_Entries element of the ExAllControl structure. In other words, the file system shall build a NULL-terminated singly linked list of ExAllData structures that are properly aligned.

dp_Arg3 is the size of the buffer in bytes dp_Arg2 points to, and into which the result of the scan shall be filled.

dp_Arg4 defines which elements of the ExAllData are filled. The encoding of this argument is given by table **??** in section **??**.

dp_Arg5 is a pointer to an ExAllControl structure, also defined in section **??**. A detailed description of this structure is also provided there. Note that this is also a regular pointer, not a BPTR.

The file system shall provide in the eac_Entries element of the ExAllControl structure the number of ExAllData structures it could fit into the target buffer pointed to by dp_Arg2.

The file system may store internal state information of the directory scanner in eac_LastKey. This state information could, for example, correspond to the fib_DiskKey in the FileInfoBlock identifying an element in a directory at which to continue an interrupted scan. It is zero-initialized before the first packet, and thus the file system may assume that a key value of 0 identifies the start of the scan.

If non-NULL, eac_MatchString is a parsed pattern as generated by the ParsePatternNoCase() function, see section **??**. If this pattern is present, only directory entries whose name matches the pattern shall be filled into the target buffer.

eac_MatchFunc provides even finer control of which objects are filled into the target buffer; it supplies a pointer to a Hook structure whose h_Entry function shall be called for each candidate directory entry, e.g. by the CallHookPkt() function of the *utility.library*. Its object argument is then a pointer a LONG containing the type from dp_Arg4, and its message argument a pointer to the ExAllData structure filled with a candidate entry. If the hook returns non-zero, the ExAllData copied into the buffer shall be considered accepted and shall remain in the buffer; if it returns zero, this candidate entry is rejected and shall not appear in the output. Calling conventions of this hook are also described in section **??**.

The same precautions as for ACTION_EXAMINE_NEXT hold for ACTION_EXAMINE_ALL, too. In particular, the file system shall be prepared for the directory to get modified during scans, by either adding, removing, renaming or moving objects out of or into the directory. Using a disk block or a pointer to a structure representing a specific object through eac_LastKey is therefore discouraged. Similarly, on subsequent requests forming a scan of a directory, the ExAll() interface requires that dp_Arg1 is a lock to the same directory, though it need not necessarily be the identical lock as used in the previous iteration over the same directory. In other words, eac_DiskKey shall hold sufficient state information to reconstruct the point from which the scan can be continued. It is, however, ensured that the pointer to the ExAllControl structure stored in dp_Arg5 does not change between requests, and the file system may depend on the pointer having the same value as on a the previous (incomplete) request for the directory. Also, file systems may depend on the client sending a packet of type ACTION_EXAMINE_ALL_END to abort a scan before reaching the end of the directory.

The ACTION_EXAMINE_ALL packet shall be replied with dp_Res1 set to DOSTRUE if not all entries could be fit into the buffer, i.e. if dp_Arg3 bytes were not sufficient to hold all matching directory entries and at least another iteration over the directory is necessary to supply (some of) the missing entries. In

such a case, dp_Res2 shall also be set to 0. Such an incomplete scan is either continued with another ACTION_EXAMINE_ALL packet, or will be aborted by an ACTION_EXAMINE_ALL_END packet.

In case the end of the directory has reached and all directory entries could fit into the buffer, dp_Res1 shall be set to 0, and dp_Res2 to ERROR_NO_MORE_ENTRIES. In case the directory scan had been aborted due to an error, dp_Res1 shall be set to 0 and dp_Res2 to an error code. In both cases, all resources corresponding to the scan shall be released.

14.3.4 Aborting a Directory Scan

The packet ACTION_EXAMINE_ALL_END aborts a directory scan started with ACTION_EXAMINE_ALL that returned with a non-zero result in dp_Res1. It allows file systems to release resources associated to the scan.

Table 14.24: ACTION_EXAMINE_ALL_END

DosPacket Element	Value
dp_Type	ACTION_EXAMINE_ALL_END (1035)
dp_Arg1	BPTR to FileLock
dp_Arg2	APTR to a buffer to fill
dp_Arg3	Size of the buffer
dp_Arg4	Type defining the requested entries
dp_Arg5	Pointer to struct ExAllControl
dp_Res1	Boolean success flag
dp_Res2	Error code

This packet implements the ExAllEnd() function of the *dos.library*, see section **??**, and aborts a partial directory scan started with ExAll() but for which neither the end of the directory has been reached nor an error code has been received. This packet may be used by an implementing file system to release any temporary resources allocated for the purpose of the scan. It was introduced in AmigaDOS 39.

Arguments are populated as follows:

dp_Arg1 is the lock on the directory on which a partial scan has been started. It is not necessarily the original lock passed into ACTION_EXAMINE_ALL, but a lock on the same directory.

dp_Arg2 and dp_Arg3 are a pointer to a buffer and its size. This buffer, however, should not be touched and no data should be deposited there; as this buffer is not necessarily the same as the one for which the scan has been started, file systems may ignore these arguments.

dp_Arg4 would define which information would go into the buffer provided by dp_Arg2; the encoding of dp_Arg4 is given by table **??** in section **??**. However, as this packet should not fill any data into the supplied buffer, file systems may also ignore it.

dp_Arg5 is a pointer to an ExAllControl structure, and the pointer provided here is identical to the pointer provided to the ACTION_EXAMINE_ALL packet whose abortion is requested. Thus, file systems may use the pointer value and specifically the eac_DiskKey element in the structure it points to to identify and release any resources related to the scan.

Before replying this packet, the file system shall set dp_Res1 to a Boolean success code indicating whether it could abort the scan. On success, dp_Res1 shall be set to DOSTRUE, and dp_Res2 to 0.

If the file system does not implement this packet, it shall set dp_Res1 to DOSFALSE and dp_Res2 to ERROR_ACTION_NOT_KNOWN. The *dos.library* then (attempts to) emulates this packet by setting the match pattern in the ExAllControlExAllControl structure to a pattern that matches no entry, and then continues the scan with a ACTION_EXAMINE_ALL packet. This would then clearly reach the end of the directory without overrunning the buffer. However, due to a defect in the *dos.library* up to the latest version, the (non-matching) pattern is a literal pattern and not a pre-parsed pattern. It therefore could match

an entry in the directory. Thus, it is advisable to always implement ACTION_EXAMINE_ALL_END if ACTION_EXAMINE_ALL is implemented, even if it is just replied with success and no additional resources need to be released.

On any other error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.3.5 Examining from a File Handle

The ACTION_EXAMINE_FH packet examines an object identified by a file handle rather than a lock.

Table 14.25: ACTION_EXAMINE_FH

DosPacket Element	Value
dp_Type	ACTION_EXAMINE_FH (1034)
dp_Arg1	fh_Arg1 of the FileHandle
dp_Arg2	BPTR to a FileInfoBlock structure
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the ExamineFH() function of the *dos.library*, see section **??**, and retrieves information on an object identified by a file handle rather than a lock. This packet therefore allows to retrieve information on files from handlers that do not support locks. It was introduced in AmigaDOS version 36.

dp_Arg1 is a copy of the fh_Arg1 element of the FileHandle that is to be examined and thus serves to identify the object[6].

dp_Arg2 is a BPTR to a FileInfoBlock structure as documented in section **??**, and which shall be filled by the handler with the information on the object identified by dp_Arg1. As for the packet type ACTION_EXAMINE_OBJECT specified in section **??**, the handler shall fill fib_FileName and fib_Comment with BSTRs. They are converted to NUL-terminated C strings by the *dos.library*.

Before replying this packet, the file system shall set dp_Res1 to a Boolean result code. On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.4 Packets for Working with Links

The packets in this section interact with links and define the interface for working with links on the file system level. They mirror the functions of the *dos.library* listed in section **??** which also provides additional background information.

The packet interface consists of two packets: one for creating all types of links given the path of the link to create and the link target, and a second packet that is specific to soft links only: this packet finds the path of the link target given a path containing a soft link as one of its components. It is mostly used by the *dos.library* internally to resolve soft links transparently to applications.

Unlike soft links, hard links are resolved entirely within the handler and do not require interaction with the *dos.library*.

AmigaDOS version 47 introduced another type of link, the *external link*. When accessing the link, the file system containing the link interacts with the file system of the link target and copies the contents of the target into the link. Thus, external links implement a "copy on read" functionality. Such links are created by the same packet that also creates soft and hard links, and similar to hard links, they do not require an additional packet for resolving them.

[6]Note that the information in [**?**] on dp_Arg1 is incorrect.

14.4.1 Creating Links

The ACTION_MAKE_LINK packet creates soft, hard or external links in a file system.

Table 14.26: ACTION_MAKE_LINK

DosPacket Element	Value
dp_Type	ACTION_MAKE_LINK (1021)
dp_Arg1	BPTR to a FileLock
dp_Arg2	BPTR to BSTR of the path of the link to create
dp_Arg3	BPTR to FileLock or APTR to C-string
dp_Arg4	Type of the link, from table **??**
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the MakeLink() function of the *dos.library*, see section **??**, and as such creates soft, hard or external links within the file system. It was added in AmigaDOS version 36.

Arguments are populated as follows:

dp_Arg1 is a BPTR to a FileLock structure that identifies the directory within which the object is to be created. It is typically retrieved from the GetDeviceProc() function.

dp_Arg2 is a BPTR to a BSTR that provides the path of the link to be created. This path is logically appended to the path of the directory given by dp_Arg1, and the last component of this path is the file name of the link to be created.

dp_Arg3 identifies the target of the link, i.e. the object the link points to. If dp_Arg4 is LINK_HARD, then this argument is a BPTR to a FileLock structure identifying the target. The type of the link then depends on whether this target is on the same or a different file system.

If the FileLock identifies an object on the same file system as dp_Arg1, then a hard link is created. If the object locked by dp_Arg3 is a file, then the created link will represent a file, otherwise a directory.

Resolution of hard links is up to the file system. That is, if the link is accessed, it is up to the file system to locate the target of the link and access it instead of the link. Multiple implementation strategies exist for hard links: Either, each object includes a reference counter that is incremented for each directory entry pointing to it, that is for each link created; likewise, this counter is decremented each time a directory entry is removed. If the counter reaches 0, the object itself is deleted. This strategy is used for example by the family of ext file systems on Linux. Links cannot be distinguished from regular objects in this design as the link is just another directory entry pointing to the same file system object.

Alternatively, each link is pointing to the linked object, and the object itself contains a list of links that reference it. If a link is deleted, it is removed from its directory and from the list of links within the target object. If the object itself is deleted, one of the links becomes the object itself and the list of links is copied from the deleted object to the link. The FFS implements the latter design.

If dp_Arg4 is LINK_HARD and the FileLock pointed to by dp_Arg3 is on a different file system, then an external link shall be created. That is, the file system addressed by the packet shall create a file or directory within its own file system, and this file or directory shall mirror the contents of the link target given by dp_Arg4 whenever the link or an object within the link is accessed; the file system shall thus implement a “copy on read” for the linked object. If the link target is a directory, it shall also create copies of the objects within it if they are accessed.

Once the copy is made, the copied object is detached from its source; if the source is deleted or modified, the copied objects remains available unaltered, and can also be modified or deleted without affecting the object in the link target. If no copy has been created so far and the object in the link target has been deleted, an attempt to access the linked object fails with ERROR_OBJECT_NOT_FOUND.

Currently, only the RAM-Handler implements this type of link, and it is there used to realize the ENV assign containing all preferences settings. Section **??** contains more information on external links which were added in AmigaDOS version 47.

If dp_Arg4 is LINK_SOFT, then dp_Arg3 is a pointer to a NUL-terminated C string providing the target of the link. This string is not immediately interpreted by the file system and stored as-is; it is interpreted as a path name relative to the location of the link at the time the link is resolved[7].

Resolution of soft links is performed within the *dos.library* or within the client application as not all functions of the library implement soft link resolution. For a list of functions that *do* implement it, see table **??** in section **??**, see also section **??** for how to resolve soft links manually.

Upon accessing a soft link, or a path containing a soft link, the file system shall create an error code ERROR_IS_SOFT_LINK. If this error is received, the sender of a packet accessing the link — thus typically the *dos.library* itself — shall read the target of the link with the ReadLink() function specified in section **??**. This function then again contacts the file system containing the link with a ACTION_READ_LINK which shall construct from the link an updated path to the object. Section **??** provides more details on this packet.

From this follows that file systems are not aware whether the target of a soft link actually exists as they can only provide the path to the object, but not necessarily the object itself; in particular, soft links can cross file system boundaries. If the target of a soft link is deleted, the soft link itself remains and becomes a "dangling" link. When accessing such a link, its resolution fails with an error ERROR_OBJECT_NOT_FOUND.

Upon replying the ACTION_MAKE_LINK packet, dp_Res1 shall be set to DOSTRUE and dp_Res2 to 0 on success. On failure, dp_Res2 shall be set to DOSFALSE and dp_Res1 to an error code.

14.4.2 Resolving a Soft Link

The ACTION_READ_LINK packet constructs from a path containing a soft link an updated path by inserting the target of the link.

Table 14.27: ACTION_READ_LINK

DosPacket Element	Value
dp_Type	ACTION_READ_LINK (1024)
dp_Arg1	BPTR to a FileLock
dp_Arg2	APTR to C string
dp_Arg3	APTR to buffer
dp_Arg4	Buffer size
dp_Res1	Actual buffer size needed
dp_Res2	Error code

This packet implements the ReadLink() function of the *dos.library*, see section **??**, and is used there to resolve soft links and obtain the link target. This packet was added in AmigaDOS version 36.

Implementing this packet correctly requires, however, some care as the soft link may be in the middle of the path provided by dp_Arg2 and not just at its last component, and the soft link itself may be an absolute or relative path. This packet shall provide from the original path an updated path that points to the intended location relative to the location *of the soft link*, and that is not necessarily the same as a path relative to dp_Arg1.

Arguments of this packet are populated as follows:

dp_Arg1 is a lock to a directory in the file system relative to which the path in dp_Arg2 shall be interpreted. This *need not* to be directly the directory containing the link. The lock argument of ReadLink() will be placed here.

[7]This is quite unusual as AmigaDOS otherwise represents strings as BSTR, though that would limit the length of the link target.

dp_Arg2 is a pointer to a path name relative to dp_Arg1 which contains a soft link as one of its components. The soft link *need not to be* the last component of this path, it can also be one of the components in the middle of it. This argument is, unlike in many other packets, not a BPTR to a BSTR but a pointer to a regular NUL-terminated C string.

The file system shall now proceed with soft link resolution as follows: Starting from the directory given by dp_Arg1, it shall interpret the path in dp_Arg2 component by component as explained in chapter **??**, until it finds a soft link. That is, a colon (":") indicates that the scan shall continue at the root directory, an isolated slash ("/") shall enter the parent directory, and all other components terminated by a slash shall be interpreted as names of directories that shall be recursively entered. This process continues until either the end of the path is found, or — and this is the purpose of this packet — a soft link is detected.

If a soft link is detected as part of the path, the target of the link as stored in the file system shall be read. Note that the link target can not only consist of an isolated file or directory name, but can well be an absolute or relative path that shall be merged with the path provided in dp_Arg2.

In particular, if the path stored in the soft link contains a colon (":"), and hence is an absolute path, the components parsed off from dp_Arg2 up to the detection of the soft link shall be disregarded in order to start from the root directory. If no device or volume name is provided in the absolute link target, the file system shall insert the name of the current volume and a colon, in order to provide an unambiguous anchor of the newly constructed path. The remaining path stored in the soft link is then concatenated to the volume name. Finally, the rest of the components from dp_Arg2 behind the component name of the soft link are concatenated to this intermediate path, forming the final link target.

If the path stored in the soft link is a relative path, then it shall be concatenated to the path from dp_Arg2 up to the component name of the soft link at which iterating through dp_Arg2 was interrupted. The remaining components from dp_Arg2 behind the component name of the soft link shall then be attached to this intermediate path, forming the final result.

This procedure is tedious, but it ensures that the soft link is interpreted relative to the path from which the client application detected it, even if the soft link is the middle of the supplied path and not contained directly in the directory provided by dp_Arg1. Note that the above algorithm only resolves a single soft link in a path; this is intentional, the *dos.library* will send another ACTION_READ_LINK packet to the same or another file system if the remaining path contains additional soft links, or even in case the soft link points to another soft link. It is therefore important at the level of client application — or the *dos.library* — to detect endless loops of soft links pointing to each other. The *dos.library* aborts soft link resolution after the expansion of 15 links.

dp_Arg3 is the pointer (not a BPTR) to a target buffer into which the path resolving the soft link shall be filled as a NUL-terminated C string.

dp_Arg4 is the size of this target buffer in bytes.

On an error, dp_Res1 shall be set to -1, or to -2 in case the target buffer overflows and cannot take the full target path[8]. In case of error, dp_Res2 shall also be set to an error code. In particular, if the target buffer size in dp_Arg4 is too small to hold the resolved path, dp_Res2 shall be set to ERROR_LINE_TOO_LONG and dp_Res1 to -2.

If the source path in dp_Arg2 does actually not contain a soft link and the scanning algorithm aborted without finding one, this also constitutes an error that shall be reported by setting dp_Res1 to -1. This error is not handled consistently in AmigaDOS. The RAM-Handler reports ERROR_OBJECT_WRONG_TYPE, the FFS the error code ERROR_OBJECT_NOT_FOUND. While the former error looks more sensible, the latter is the right choice — it helps an algorithm resolving soft links to identify the case that it has probably not yet found the right directory of a multi-assign that contains the soft link. See also the code in section **??**.

On success, dp_Res1 shall be set to the length of the constructed path, i.e. to the length of the string in dp_Arg3 (not including the terminating NUL), and dp_Res2 to 0.

[8]Currently any negative value will do to indicate an error, and the *dos.library* does not distinguish between -1 and -2 as result code. Unfortunately, even the latest version of the FFS will *not* return a negative value but 0 in case of an error; this is a defect.

The following example is a skeleton code implementing the above algorithm:

```
/* Representation of a file system object in a file
** system... this is a prototype. Extend as needed.
*/
struct node {
  LONG type;  /* entry type, file, directory, link... */
  LONG size;  /* size of the soft link name           */
  ....
};

/* Find the object from lock and string. This is a file
** system internal function that need to be provided
** by the user.
** path_before is the offset of the last component
** of the path in the string that could be resolved
** successfully. If a softlink is found in the
** middle of the path, or some other error was found,
** NULL is returned.
** link_node is set to a pointer to a structure that
** represents the link.
** path_pos is set to the offset of the "/" behind
** the soft link if a soft link is in the middle.
** If the link is the last component of the path, then
** the node that represents the soft link is returned,
** Otherwise, the node is returned.
*/
extern struct node *locatenode(BPTR lock,const UBYTE *path,
                               LONG *path_before,
                               struct node **linknode,
                               LONG *path_pos);

/* copy the target of the soft link into the buffer behind
** the path of the link as zero-terminated C string.
** This function also need to be provided by the
** implementation of the file system. It returns
** the size of the soft link as result, or -1 on error.
*/
extern LONG readsoftlink(const struct node *,
                         UBYTE *buffer,LONG size);

/* Will be filled into dp_Res2 on return, i.e.
** what will be filled into IoErr()
*/
extern LONG res2;

/* Pointer to DosList of this handler, needed
** for the name of the volume.
*/
extern struct DosList *volumenode;

/* Skeleton of an implementation of the readlink function
```

```
** lock   is from dp_Arg1 and a BPTR to a filelock
** string is from dp_Arg2 and the path containing a link
** buffer is from dp_Arg3 and the target buffer
** size   is from dp_Arg4 and the size of the buffer.
*/
LONG readlink(BPTR lock, UBYTE *string,UBYTE *buffer, ULONG size)
{
  struct node *node,*linknode;
  UBYTE *src  = string;
  LONG is_dir = FALSE;
  LONG path_before;
  LONG path_pos;
  LONG res,len;
  /*
  ** Locate the first soft link in the path
  */
  node = locatenode(lock,string,&path_before,
                    &linknode,&path_pos);
  if (!node) {
    if (IoErr() == ERROR_IS_SOFT_LINK) {
      is_dir = TRUE;                /* remember... */
      node   = linknode;
    } else {
      /* Resolution did not work for some
      ** other error. Return an error.
      */
      return -1;
    }
  }

  /* Check whether the found object is a softlink */
  if (node->type != ST_SOFTLINK) {
    res2 = ERROR_OBJECT_WRONG_TYPE;
    return -1;
  }

  /* Check whether the link is in the middle or the end of
  ** the path.
  */
  if (is_dir) {
    /* must deal with '/' and ':' correctly in the soft link
    ** code below counts on these assigns!
    */
    string = string + path_pos;
    len    = strlen(string) + 1;       /* + 1 for slash */
  } else {
    len    = 0;        /* softlink is last part of string */
  }

  /* Check whether there is sufficient room in the
  ** target buffer.
```

```
*/
if (node->size + 1 + path_before + len >= size) {
  res2 = ERROR_LINE_TOO_LONG;
  return -2;
}

/* Copy the head of the link resolution path to the
** target buffer, ready to append the contents of
** the link.
*/
memcpy(buffer,src,path_before);

/* Find the contents of the link */
res = readsoftlink(node,buffer + path_before,size - path_before);
if (res < 0)
  return res; /* Deliver the error */

/* If the link is absolute, i.e.\ contains a ':', then ignore the
** start of the string. 'res' is the length of the link location.
*/
src = strchr(buffer + path_before,':');
if (src) {
  /* Yes, is absolute. First check whether it is ":", thus
  ** whether it refers to this volume.
  */
  if (src == buffer + path_before) {
    /* Is relative to our root, but not necessarily
    ** to the root of the caller, so fix up.
    */
    char *volname = (char *)BADDR(volumenode->dol_Name);
    LONG len      = *volname; /* It's a BSTR */
    /* Check whether there is enough buffer */
    if (len + res >= size) {
      SetIoErr(ERROR_LINE_TOO_LONG);
      return -2;
    } else {
      /* Move soft link behind the volume name,
      ** note that the softlink is ":" here and
      ** the ":" becomes part of the volume name.
      */
      memmove(buffer + len ,buffer + path_before,res + 1);
      /* Prepend the volume name */
      memcpy(buffer,volname + 1,len);
      res += len;
    }
  } else {
    /* Here the link is absolute to a given volume
    ** so move the soft link behind the volume name
    */
    memmove(buffer,buffer + path_before,res + 1);
  }
```

```
  }

  /* add on the rest of the path behind
  ** the soft link if it sits in the middle
  */
  if (is_dir) {
    if(!AddPart(buffer, string, size)) {
      SetIoErr(ERROR_LINE_TOO_LONG);
      return -2;
    }
  }

  /* The result is the length of the path created */
  return strlen(buffer);
}
```

14.5 Packets for Adjusting Metadata

Packets in this section change metadata associated with file system objects, such as name, comment, creation date or protection flags.

14.5.1 Renaming or Moving Objects

The `ACTION_RENAME_OBJECT` changes the name of an object such as a file, directory or a link, or relocates it within the directory tree of a volume.

Table 14.28: `ACTION_RENAME_OBJECT`

DosPacket Element	Value
`dp_Type`	`ACTION_RENAME_OBJECT (17)`
`dp_Arg1`	BPTR to source `FileLock`
`dp_Arg2`	BPTR to BSTR of the object path
`dp_Arg3`	BPTR to target `FileLock`
`dp_Arg4`	BPTR to BSTR of the target path
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet implements the `Rename()` function of the *dos.library*, see section **??**, and relocates and/or renames an object within the directory structure of a volume. This packet cannot relocate objects across volumes, i.e. `dp_Arg1` and `dp_Arg3` are locks to objects within the same volume, and `dp_Arg2` and `dp_Arg4` represent paths within this volume.

Arguments are populated as follows:

`dp_Arg1` represents the directory to which the source `dp_Arg2` is relative to. The `Rename()` function takes it from the output of the `GetDeviceProc()` function applied to the source path, and it typically corresponds to the current directory of the calling process or the lock of an assign in the source path.

`dp_Arg2` is the path of the original object. This path need not to consist of a single component, though the object to rename or relocate is the last component of this path. The file system shall walk the provided path, starting from `dp_Arg1`, to identify the object that is to be renamed or moved.

`dp_Arg3` is the directory to which the target path `dp_Arg4` is relative to, represented as a lock. `Rename()` also determines this lock through `GetDeviceProc()` and the target path.

dp_Arg4 is the target path of the object, relative to dp_Arg3. The location of the target is formed by logically appending the path from dp_Arg4 to the directory in dp_Arg3.

The file system shall find a suitable target directory into which the object is to be moved: for this, it shall walk the directory tree starting at dp_Arg3, resolving directories as given by the path in dp_Arg4. If any of the components in this path but the last component does not exist or is not a directory or not a hard link to a directory, the result shall be ERROR_OBJECT_NOT_FOUND or, if a soft link is in the path, ERROR_IS_SOFT_LINK.

If the source and the target object are identical, though possibly only differ by case (i.e. lower/upper case is possibly different), then original object shall be renamed, giving the original object the file name from the last component of the target path. This adjusts the case of the object name without changing the directory structure or the object location.

If the last component in the target path is an existing directory or hard link to an existing directory, and an object of the source name does not exist within it, the source object as identified by dp_Arg1 and dp_Arg2 shall be moved *into* this directory under its original name, i.e. the name as given by the last component of the path in dp_Arg2.

If the last component of the target path does not exist, it forms the new name of the target object, which will then be moved into the directory given by the target path up to the second to last component.

If the last component of the target path is a soft link, the attempt to rename into a soft link shall fail with the error code ERROR_IS_SOFT_LINK.

If the last component of the target path is an existing file or a hard link to an existing file, renaming shall fail with ERROR_OBJECT_EXISTS. That is, it is not permissible to rename into an existing file or replace an existing file.

Particular care needs to be taken if the object to be relocated is a directory. In such a case, the file system shall test whether an attempt is made to move the directory into itself or into one of the sub-directories of its own. As this would render the directory unreachable, the error ERROR_OBJECT_IN_USE shall be generated.

Before replying the packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On error, dp_Res1 shall be to DOSFALSE and dp_Res2 to an error code.

14.5.2 Deleting an Object

The ACTION_DELETE_OBJECT removes an object from a directory, releasing the storage it occupies.

Table 14.29: ACTION_DELETE_OBJECT

DosPacket Element	Value
dp_Type	ACTION_DELETE_OBJECT (16)
dp_Arg1	BPTR to FileLock
dp_Arg2	BPTR to BSTR of the object path
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the DeleteFile() function of the *dos.library*, see section **??**, which can delete files, directories and links.

dp_Arg1 is a lock to the directory to which the path in dp_Arg2 is relative. The DeleteFile() function fills it from GetDeviceProc().

dp_Arg2 is the path to the object to be deleted. The path can contain multiple components, in which case the file system shall walk the path starting at dp_Arg1 until reaching its last component which is the

object to be deleted. If the target is a directory, the file system shall check, in addition, if the directory is empty, and if not so, refuse to delete it with the error code ERROR_DIRECTORY_NOT_EMPTY.

If the target is a soft link or a hard link, the link shall be deleted, and not the object the link points to. In particular, in this special case resolution of links, in particular soft links, is not necessary. This particular case depends on the asymmetry between links and regular entries of the Amiga file systems: Deleting a link to a non-empty directory is possible, though deleting the directory itself not, even if the (or a) link could replace it and adopt its entries. Other file systems that cannot distinguish between hard links and their link targets may resolve the situation differently, and for example refuse to delete hard links that point to non-empty directories.

Before replying the packet, dp_Res1 shall be set to a Boolean success indicator. In case of success, dp_Res1 shall be set to DOSTRUE and dp_Res2 to 0. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.5.3 Changing the Protection Bits

The ACTION_SET_PROTECT changes the protection bits of a file system object.

Table 14.30: ACTION_SET_PROTECT

DosPacket Element	Value
dp_Type	ACTION_SET_PROTECT (21)
dp_Arg2	BPTR to FileLock
dp_Arg3	BPTR to BSTR of the object path
dp_Arg4	New protection bits
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the SetProtection() function of the *dos.library* introduced in section **??**; it changes the protection bits of an object on the file system to that given in dp_Arg4. For the definition of protection bits, see section **??**, table **??**. Some file systems may support only a subset of all protection bits, or may not be able to carry them on all objects, such as directories or links; in such a case, they may ignore the request, or may only implement a subset of all bits.

dp_Arg2 is a lock of a directory to which the path in dp_Arg3 is relative[9]. The SetProtection() function retrieves it from the GetDeviceProc() function.

dp_Arg3 is a path relative to dp_Arg2. This path need not to consist of a single component, though the object whose protection bits shall be changed is the last component of this path. The file system shall walk the provided path, starting from dp_Arg2 to find the object to modify.

dp_Arg4 are the new protection bits as specified in table **??** in section **??**. Note that the least significant four bits are shown inverted by most system tools such as the List command.

Links are a special case, and here file system authors need to decide whether the protection bits apply to the object itself or the links pointing to the object. The FFS keeps the protection bits of the object and links to the object always in sync. The RAM-Handler ignores requests to change the protection bits of hard links, but copies its bits always from the link target. Protection bits of soft links are not particularly meaningful. The FFS creates upon an attempt to set their protection bits the usual ERROR_IS_SOFT_LINK, whereas the RAM-Handler does not create this error if the last component of the path in dp_Arg3 is a soft link. Instead, it sets the protection bits of the link.

Before replying to this packet, the file system shall set dp_Res1 to a Boolean result code. On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 shall be set to 0. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

[9]This is not a typo, dp_Arg1 is really unused.

14.5.4 Setting the Comment to an Object

The ACTION_SET_COMMENT packet changes, adds or removes a comment of a file system object such as a directory, file or link.

Table 14.31: ACTION_SET_COMMENT

DosPacket Element	Value
dp_Type	ACTION_SET_COMMENT (28)
dp_Arg2	BPTR to FileLock
dp_Arg3	BPTR to BSTR of the object path
dp_Arg4	BPTR to BSTR of comment
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the SetComment() function of the *dos.library*, see section **??**, and as such changes, adds or removes a comment on a file, directory or link. Not all file systems will support this request, or may only support comments on a subset of objects.

dp_Arg2 represents a directory relative to which the path in dp_Arg3 shall be interpreted. The SetComment() function fills it from the GetDeviceProc() function.

dp_Arg3 is the path relative to dp_Arg2 of the object whose comment is to be changed.

dp_Arg4 is a BPTR to a BSTR of the new comment to be set. If this string is empty, the comment shall be removed. File systems shall also check the size of the comment and fail with the error code ERROR_COMMENT_TOO_BIG if the comment is longer than what the file system and the FileInfoBlock structure (see section **??**) supports. The latter limits the size of the comment to 79 characters. Note that the handling of overlong comments is different from handling overlong file names: The latter are truncated without error to the maximum size the file system supports.

Before replying to this packet, the file system shall set dp_Res1 to a Boolean result code. On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 shall be set to 0. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.5.5 Setting the Creation Date of an Object

The ACTION_SET_DATE packet sets the creation date of an object on a file system.

Table 14.32: ACTION_SET_DATE

DosPacket Element	Value
dp_Type	ACTION_SET_DATE (34)
dp_Arg2	BPTR to FileLock
dp_Arg3	BPTR to BSTR of the object path
dp_Arg4	BPTR to DateStamp structure
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the SetFileDate() function of the *dos.library* from section **??** and sets the creation date of an object on a file system. Not all file systems will support this packet, or may only support it with reduced precision of the date.

dp_Arg2 represents a directory relative to which the path in dp_Arg3 is interpreted[10]. The implementation of the SetFileDate() function retrieves it from the GetDeviceProc() function.

[10]The information in [**?**] is incorrect, and dp_Arg1 is, indeed, unused.

dp_Arg3 is the path of the object whose creation date is to be changed. This path shall be interpreted relative to dp_Arg2, and its last component is the object whose creation date shall be changed.

dp_Arg4 is a BPTR to a DateStamp structure as specified in section **??** and defines the target date to install in the metadata of the object.

The AmigaDOS file systems, the FFS and the RAM-Handler, always keep the file date of hard links and link targets in sync, and it is recommended that alternative file system implementations follow this strategy. The strategy for soft links is not that uniform. While both file systems support a separate creation date for soft links, this packet adjusts the date of the link for the RAM-Handler, whereas it creates an ERROR_IS_SOFT_LINK for the FFS, and by the help of the *dos.library*, adjusts then the date of the link target. Given that the date is probably used to test whether a particular file is up to date, e.g. for make utilities, the latter strategy is advisable.

Before replying to this packet, the file system shall set dp_Res1 to a Boolean result code. On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 shall be set to 0. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.5.6 Setting the Owner of an Object

The ACTION_SET_OWNER packet sets the owner and group ID of an object in a file system.

Table 14.33: ACTION_SET_OWNER

DosPacket Element	Value
dp_Type	ACTION_SET_OWNER (1036)
dp_Arg2	BPTR to FileLock
dp_Arg3	BPTR to BSTR of the object path
dp_Arg4	Owner information
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the SetOwner() function of the *dos.library* (see **??**) and is supposed to set a group and user ID of an object in a file system; the actual encoding of this information is not defined by the *dos.library* or the packet. dp_Arg4 is a literal copy of the second argument of SetOwner(). Since AmigaDOS is not a multi-user system, the value of this packet is questionable; the FFS supports this packet since version 43, the RAM-Handler does not implement it.

dp_Arg2 represents a directory relative to which the path in dp_Arg3 is interpreted. The implementation of the SetOwner() function determines it from its argument through GetDeviceProc() and fills it for example with the current directory of the calling process.

dp_Arg3 is the path of the object whose owner information is to be changed. This path shall be interpreted relative to dp_Arg2, and its last component is the object whose owner shall be changed.

dp_Arg4 is an opaque owner information; the *dos.library* does not define its meaning. This argument is a verbatim copy of the second argument of SetOwner() and the FFS copies it directly into the Owner field of the blocks of type ST.SHORT, see section **??** and following.

The FFS follows for hard and soft links the same strategy as for ACTION_SET_DATE: The owner of hard links and link targets are always kept in sync, and in case the owner of a soft link is supposed to be changed, the FFS signals the error ERROR_IS_SOFT_LINK. With the help of the *dos.library*, the owner of the link target is then changed — if the file system of the link target supports this packet. This is also the recommended strategy. The RAM-Handler does currently not implement ACTION_SET_OWNER.

Before replying to this packet, the file system shall set dp_Res1 to a Boolean result code. On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 shall be set to 0. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.6 Packets for Starting and Canceling Notification Requests

The packets in this section implement notification requests. Once a notification request has been registered, the file system informs a task through a signal or through a message send to a port when the observed object changes. This mechanism is for example used by the `IPrefs` program to load new preferences when they are modified through the preferences editors. The *dos.library* interface for notification requests is introduced in section **??**.

14.6.1 Registering a Notification Request

The `ACTION_ADD_NOTIFY` packet registers a notification request at a file system.

Table 14.34: `ACTION_ADD_NOTIFY`

DosPacket Element	Value
`dp_Type`	`ACTION_ADD_NOTIFY (4097)`
`dp_Arg1`	APTR to `NotifyRequest` structure
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet implements the `StartNotify()` function of the *dos.library*, see section **??**. It registers a request at a file system to get informed on changes on a selected file system object. Notification requests, and therefore this packet, have been introduced in AmigaDOS version 36.

`dp_Arg1` is a C pointer[11] to a `NotifyRequest` structure that defines the object to be monitored, and the task to be informed on a change of this object. This structure is specified in section **??**.

The `nr_FullName` element of the `NotifyRequest` structure is already filled by an absolute path to the object to be monitored which is assumed to exist already. This path is computed by the *dos.library* within the `StartNotify()` function from the `nr_Name` element and the current directory of the calling process, and the file system shall depend on this path and not on the contents of `nr_Name` which, from the perspective of the file system, is undefined and may not even be filled.

The file system shall store this request in its internal structures, and then monitor the object and hard links to the object identified by `nr_FullName` for changes. Notification requests on directories shall also monitor objects directly contained in the monitored directory for changes; the request does not extend recursively to the contents of sub-directories.

A notification event either sets a signal or sends a message to a `MsgPort`:

If `NRF_SEND_MESSAGE` is set in `nr_Flags`, then a `NotifyMessage` shall be send to the port indicated in `nr_Msg.nr_Port`. This structure is also specified in section **??**. Its `nm_Class` element shall be set to `NOTIFY_CLASS`, and its `nm_Code` element to `NOTIFY_CODE`, both are defined in `dos/notify.h`. In addition, `nm_NReq` shall point to the `NotifyRequest` structure through which the object is monitored. The elements `nm_DoNotTouch` and `nm_DoNotTouch2` may be used for internal administration of the file system.

If `NRF_SEND_SIGNAL` is set, then the file system shall set a signal bit of the target task if the monitored object changes by calling

```
Signal(nr->nr_Signal.nr_Task,1UL<<nr->nr_Signal.nr_SignalNum);
```

If the `NRF_NOTIFY_INITIAL` flag is set in `nr_Flags`, the file system shall trigger a notification immediately after having received this packet, regardless of whether or not the object has been modified already. If `NRF_WAIT_REPLY` is set, and a `NotifyMessage` has already been send out, it shall rather note

[11] In [**?**] this argument is documented to be a BPTR, though this information is incorrect.

the modification, though defer any further notification on the same object until the former NotifyMessage has been replied. This avoids queuing up too many notifications at the client port. The NRF_MAGIC bit in nr_Flags may be used by the handler to implement this functionality; it may mark those requests for which a message has been send, but no reply has been received yet.

NRF_WAIT_REPLY is, of course, non-functional if notifications are transmitted through a signal by NRF_SEND_SIGNAL.

Table **??** lists the packets which shall or may trigger a notification request on a monitored object. Some changes do not trigger a notification request immediately, but only after the modified file is closed. This avoids sending notifications on incomplete objects and avoids piling up too many notifications at once. Monitoring a directory implicitly also monitors the files within it for the changes listed in this table:

Table 14.35: Packets triggering a Notification

Packet	Notes
ACTION_WRITE	After closing the file with ACTION_END
ACTION_SET_FILE_SIZE	After closing the file with ACTION_END
ACTION_FINDOUTPUT	After closing the file with ACTION_END
ACTION_FINDUPDATE	Only if the file was created, after closing it.
ACTION_DELETE_OBJECT	Immediately
ACTION_RENAME_OBJECT	Immediately
ACTION_SET_DATE	Immediately
ACTION_CREATE_DIR	Immediately, when monitoring directories
ACTION_MAKE_LINK	Immediately, for links created in monitored directories
ACTION_SET_COMMENT	Optionally
ACTION_SET_PROTECT	Optionally
ACTION_SET_OWNER	Optionally

Volume changes, ACTION_INHIBIT, ACTION_RENAME_DISK and ACTION_SERIALIZE_DISK shall *never* trigger notification requests. However, if a volume is ejected, and re-inserted into another drive, the notification requests need to be carried over to the file system of the drive into which the volume has been reinserted, in the same way locks are carried over from one file system to the other, see **??**. This requires that the file system shall also update the nr_Handler element of the NotifyRequest structure to point to the MsgPort of the receiving file system. By updating nr_Handler, the correct file system is contacted for canceling the request.

Upon replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.6.2 Canceling a Notification Request

The ACTION_REMOVE_NOTIFY packet cancels a notification request and aborts monitoring the object recorded in the NotifyRequest structure.

Table 14.36: ACTION_REMOVE_NOTIFY

DosPacket Element	Value
dp_Type	ACTION_REMOVE_NOTIFY (4097)
dp_Arg1	APTR to NotifyRequest structure
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the EndNotify() function of the *dos.library*, see section **??**, and terminates monitoring the object in the nr_FullName element of the NotifyRequest structure. This packet was introduced along with notification requests in AmigaDOS version 36.

nr_Arg1 is a C-pointer to the structure that indicates which notification request shall be canceled. The NotifyRequest structure is specified in section **??**.

If any notifications are still pending, e.g. because the monitored object has already been changed through a file handle though this file handle has not yet been closed, such pending notifications shall be canceled as well. The file system shall be aware that even after this packet has been replied, some NotifyMessages send out to clients can be replied later because the clients are still working on them. In particular, the memory associated to such messages shall not be released upon receiving the ACTION_REMOVE_NOTIFY packet and only after the NotifyMessage has been replied and was received again by the file system. The *dos.library* replies all NotifyMessages it finds still pending in the port of the client application, but clearly cannot handle those messages that are still being processed.

If the notification request is on a volume that is currently not inserted, but the request has not been claimed by another file system, i.e. the nr_Handler points still to a port of the file system receiving the cancellation request, cancellation shall proceed as if it is still owned by this file system. This is similar to how locks on removed volumes are handled, see section **??**.

Upon replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.7 Packets Operating on Entire Volumes

The packets in this section impact the volume mounted on a file system in total and do not interact with individual objects, such as files, directories or links on it.

14.7.1 Determining the Currently Inserted Volume

The packet ACTION_CURRENT_VOLUME determines from a FileHandle a BPTR to the DosList structure representing the volume on which the file is located.

Table 14.37: ACTION_CURRENT_VOLUME

DosPacket Element	Value
dp_Type	ACTION_CURRENT_VOLUME (7)
dp_Arg1	fh_Arg1 of a FileHandle or ZERO
dp_Res1	BPTR to DosList structure
dp_Res2	0

This packet is not exposed by the *dos.library*, but it used by it when constructing an error requester through its ErrorReport() function documented in section **??**. If an error is generated by a function taking a FileHandle as argument, this packet is used to determine the DosList entry representing the volume. From there, the name is copied into the requester, for example to ask the user to insert it.

dp_Arg1 shall be filled with a copy of the fh_Arg1 element of a FileHandle structure, or shall be ZERO. If a non-ZERO handle is provided, the file system shall fill a BPTR to the DosList of the volume the handle is located on into dp_Res1. In such case, the packet is safe as the volume entry cannot vanish as long as the handle still uses it[12].

[12] In [**?**], dp_Arg1 is not mentioned, though providing it is important to avoid a race condition. The RAM-Handler is probably an exception as it only maintains a single volume that cannot vanish during its lifetime.

If dp_Arg1 is ZERO, a BPTR to the DosList of the currently inserted volume, or ZERO if no volume is inserted, shall be returned. Unfortunately, as the volume may be ejected any time, it is unclear whether the DosList entry returned in dp_Res1 is still valid at the time it is interpreted by the issuer of of this packet. In such a case, the issuer of the packet should lock the device list upfront, copy the required information, for example the name of the volume, and unlocked the list again as seen from the following code:

```
void NameOfVolume(struct MsgPort *port,UBYTE *name,size_t len)
{
  struct DosList *dl;

  LockDosList(LDF_VOLUMES|LDF_READ);
  dl = (struct DosList *)BADDR(DoPkt1(port,ACTION_CURRENT_VOLUME,ZERO));
  if (dl) {
    /* dol_Name is always a 0-terminated BSTR */
    strncpy(name,(UBYTE *)BADDR(dl->dol_Name) + 1,len);
    name[len-1] = 0;
  } else name[0] = 0;
  UnLockDosList(LDF_VOLUMES|LDF_READ);
}
```

The file system shall set dp_Res1 to a BPTR to the DosList entry representing a volume, i.e. an entry on the device list whose dol_Type is DLT_VOLUME and whose dol_Task points to a MsgPort of the file system having received the packet. In case no volume is inserted, dp_Res1 shall be set to ZERO. This packet shall not fail, and dp_Res2 is ignored by the *dos.library*. For practical purposes, dp_Res2 shall be set to 0 indicating that no error has been detected.

14.7.2 Retrieving Volume Information from a Lock

The ACTION_INFO packet retrieves information on a volume, given a lock of an object on the volume, and provides it in an InfoData structure.

Table 14.38: ACTION_INFO

DosPacket Element	Value
dp_Type	ACTION_INFO (26)
dp_Arg1	BPTR to FileLock
dp_Arg2	BPTR to InfoData
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the Info() function of the *dos.library*, see section **??**, and provides information on the volume on which the locked object is located, or on the currently inserted volume if the lock is ZERO.

dp_Arg1 is a BPTR to a lock, used to identify a volume from which information is requested. If this lock is non-ZERO, the file system shall first validate whether the lock is valid, and whether it is a lock it had created. The packet shall fail with ERROR_INVALID_LOCK if not so. Once the lock is validated, the file system shall check whether the volume of the locked object is currently inserted, and if not, shall report the error ERROR_DEVICE_NOT_MOUNTED.

If dp_Arg1 is ZERO, the packet shall provide information on the currently inserted volume; if no volume is inserted, it shall fail with ERROR_NO_DISK. If the disk structure of the inserted volume is corrupt, the error code ERROR_NOT_A_DOS_DISK shall be indicated.

dp_Arg2 is a BPTR to an InfoData structure as defined in section **??** which shall be filled with information on the currently inserted volume, such as the state of the volume, the number of errors detected on it, its capacity and the number of free blocks. Details on this structure and the information therein are found in section **??**.

Upon replying this packet, dp_Res1 shall be set to a Boolean success indicator. If information on the volume could be retrieved, dp_Res1 shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

This packet is *not identical* to ACTION_DISK_INFO introduced in section **??** even if dp_Arg1 is set to ZERO. ACTION_INFO checks in addition the state of the inserted volume and generates an error code if no disk is inserted or the inserted disk is not validated. The packet in section **??** does not fail under such conditions, but returns the disk state in the InfoData structure in any case.

14.7.3 Retrieving Information on the Currently Inserted Volume

The ACTION_DISK_INFO packet retrieves information on whether the drive a file system is operating on contains a medium, whether this medium contains a valid file structure and whether it is writable. It provides this information in an InfoData structure.

Table 14.39: ACTION_DISK_INFO

DosPacket Element	Value
dp_Type	ACTION_DISK_INFO (25)
dp_Arg1	BPTR to InfoData
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is currently not exposed by the *dos.library* and must be issued through the packet interface directly. It is almost identical to ACTION_INFO except that it does not take a lock as argument.

If a disk is inserted and validated, it returns the same information as ACTION_INFO. Otherwise, it *does not fail* but returns the state of the file system in the InfoData structure. In specific, it sets id_DiskType to ID_NO_DISK_PRESENT if no volume is inserted, or to ID_NOT_REALLY_DOS if the disk is readable, but its disk structure is corrupt. Unlike the packet from **??**, this packet does not create errors in such cases. For all other other id_DiskType values this packet can generate, see table **??** in section **??**.

This packet is also used to retrieve console specific information from the CON-Handler or interactive handlers in general, and is documented as such again in section **??**. The elements of the InfoData are then interpreted in an alternative way.

dp_Arg1 is a BPTR to an InfoData structure as defined in section **??** into which information is filled such as the state of the currently inserted volume, the number of errors detected on it, its capacity and the number of free blocks. Details on this structure are found in section **??**.

Upon replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

At the time the packet is received by its issuer, the information provided in the InfoData structure is not necessarily up do date as the user can change the volume any time. It also remains operational if the file system is inhibited, see **??**, but then delivers 'BUSY' as disk state. If information on a specific volume is needed, ACTION_INFO described in section**??** is a better choice as ACTION_DISK_INFO only provides a snapshot of the file system state at some time in the past. The Workbench uses this packet to determine whether or which icon to show on its screen.

14.7.4 Relabeling a Volume

The ACTION_RENAME_DISK packet changes the volume name of the inserted medium.

Table 14.40: ACTION_RENAME_DISK

DosPacket Element	Value
dp_Type	ACTION_RENAME_DISK (9)
dp_Arg1	BPTR to BSTR of new volume name
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the Relabel() function of the *dos.library* as specified in section **??** and changes the volume name of the inserted medium. As such, it shall also change the name of the DosList representing the volume in the device list, i.e. adjust the dol_Name element of the DosList structure.

dp_Arg1 is a BPTR to a BSTR of the new volume name. Relabeling shall be applied on the currently inserted volume.

Before replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

This packet does not provide means to identify the medium to be relabeled, it always affects the currently inserted medium. If the user changes the medium after the packet has been send, but before it has been received by the file system, the recently inserted medium will be relabeled.

14.7.5 Initializing a New File System

The ACTION_FORMAT packet initializes a new file system on a medium or partition, writing administration information representing a blank volume on it. It therefore deletes all information stored previously on the volume.

Table 14.41: ACTION_FORMAT

DosPacket Element	Value
dp_Type	ACTION_FORMAT (1020)
dp_Arg1	BPTR to BSTR of new volume name
dp_Arg2	Dos type or other private data
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the Format() function of the *dos.library* introduced in section **??**. It was added in AmigaDOS version 36. Before that release, the Format command of the Workbench included the code to initialize the OFS or FFS disk structure.

When the file system receives this packet, it performs a soft-initialization of the partition or medium currently inserted. Note, however, that it does not perform low-level formatting, such as creating sectors on a floppy disk or issuing a low-level SCSI format command. If this is intended, formatting must be performed by a client application upfront, e.g. by using the TD_FORMAT command of the underlying exec device. This packet only writes administration information on a volume that represents it in empty state.

dp_Arg1 is a BPTR to a BSTR of the volume name the medium or partition shall be given[13]. This packet shall be issued while the file system is inhibited, and thus the DosList structure representing the volume will be created or updated at the time the file system is uninhibited.

[13]The information in [**?**] that the new volume name is in dp_Arg2 and the DosType is in dp_Arg3 is incorrect.

dp_Arg2 contains file system specific information that may be used to define its flavor. For example, for the FFS dp_Arg2 carries the DosType, which shall be one of the values documented in table **??** in section **??**. The Format() function will take dp_Arg2 from its third argument.

Before replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.7.6 Make a Copied Disk Unique

The ACTION_SERIALIZE_DISK packet serializes a volume, that is, ensures that the volume is unique and distinguishable from other volumes available to the system.

Table 14.42: ACTION_SERIALIZE_DISK

DosPacket Element	Value
dp_Type	ACTION_SERIALIZE_DISK (4200)
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is not exposed by the *dos.library*. The packet interface e.g. DoPkt(), must be used to issue it. The purpose of this packet is to ensure that the volume inserted in a drive is unique and distinguishable from other volumes. Along with ACTION_FORMAT, this packet shall only be issued while the file system is inhibited, see sections **??** and **??**.

This packet does not take any arguments, it affects the currently inserted volume. The FFS implements this packet by setting the volume creation date to the system date; other file systems can include volume IDs or other means to uniquely label disks. The DiskCopy command of the Workbench uses this packet after copying the disk content to ensure that the copy is distinguishable from its original.

Before replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.7.7 Write Protecting a Volume

The ACTION_WRITE_PROTECT packet enables or disables a software write-protection on the currently inserted volume, thus disallowing any write access through the file system on it.

Table 14.43: ACTION_WRITE_PROTECT

DosPacket Element	Value
dp_Type	ACTION_WRITE_PROTECT (1023)
dp_Arg1	Write protection flag
dp_Arg2	Password hash
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is not exposed by any function of the *dos.library*, though the Lock command of the Workbench uses it to enable or disable write protection on a volume. Optionally, write protection may be secured by a password of which only an integer hash key is provided to the file system. To re-enable write access, the file system shall check whether the password hash supplied matches the one when setting the write protection, and shall refuse to unblock it if the password hash keys do not match. How a password hash is computed is irrelevant to the file system, it only checks the keys for enabling and disabling the write protection for equality.

dp_Arg1 is a Boolean indicator that, if non-zero, enables write protection, and if DOSFALSE releases it. Any attempt to write data to the protected medium or partition shall fail with the error code ERROR_DISK_WRITE_PROTECTED, including an attempt to set a write protection on an already write protected file system, regardless of the password key used for re-protection.

dp_Arg2 is an integer password hash key that is stored internally in the file system when the write protection is set, and compared against if it is to be released again. If this hash is 0, then any password releases the lock; otherwise, if the received password hash is not equal to the original hash, attempting to release the protection shall fail with ERROR_INVALID_COMPONENT_NAME.

Version 47 of the Lock command uses currently the following algorithm to compute a hash key[14]:

```
ULONG ComputeHash(const UBYTE *password) {
  ULONG arg2  = 0;
  while (*password)
    arg2  = 10 * arg2 + *password++;
  return arg2;
}
```

Before replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.8 Packets for Interactive Handlers

The packet types documented in this section are specific to interactive handlers, e.g. handlers that interact with the environment of the computer system. Examples for interactive handlers are the CON-Handler interacting with the user through a window, the AUX-Handler which makes a console available through the serial port, and the Port-Handler which reads and writes data through the serial or parallel port and also makes the printer accessible to AmigaDOS.

14.8.1 Waiting for Input Becoming Available

The packet ACTION_WAIT_CHAR waits for characters becoming available from an interactive handler.

Table 14.44: ACTION_WAIT_CHAR

DosPacket Element	Value
dp_Type	ACTION_WAIT_CHAR (20)
dp_Arg1	Timeout in microseconds
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the WaitForChar() function of the *dos.library*, see section **??**. As such, it checks whether (interactive) input is available to satisfy a potential Read(). Note that this packet does not receive an indicator of a file handle, thus even if input is available, it is not clear which ACTION_READ packet will retrieve the available data. It neither removes any data buffered in the handler.

dp_Arg1 provides the timeout in ticks, where a tick is 20ms long, see also chapter **??**.

This packet instructs the handler to wait at most dp_Arg1 microseconds. If no input becomes available within this time period, dp_Res1 shall be set to DOSFALSE and dp_Res2 to 0.

If at least a single character of input becomes available within the timeout period, dp_Res1 shall be set to DOSTRUE. If the handler implements line buffering such as the CON-Handler, then dp_Res2 shall

[14]The algorithm in [**?**] is no longer up to date, the algorithm shown here has not been changed since at least version 40.

be set to the number of input lines available in the buffer of the handler. This feature is, for example, used by the ARexx interpreter for implementing the LINES() function. If the handler does not implement line buffering, dp_Res2 shall be set to 0.

In case of an error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.8.2 Setting the Line Buffer Mode

The ACTION_SCREEN_MODE packet changes the buffer mode of an interactive console and disables or enables line buffering.

Table 14.45: ACTION_SCREEN_MODE

DosPacket Element	Value
dp_Type	ACTION_SCREEN_MODE (994)
dp_Arg1	Buffer mode
dp_Res1	Boolean result code
dp_Res2	Error code

This packet implements the SetMode() function of the *dos.library* defined in section **??**. dp_Arg1 is a copy of the second argument of this function.

The purpose of this packet is to adjust the buffer mode of a console. The CON-Handler, responsible for all types of consoles, serves both the CON, the RAW and (through an indirection layer) the AUX devices, all of which are incarnations of the same handler configured differently; the CON-Handler is described in section **??** in more detail.

In particular, the devices CON and RAW are just two modes of the in total three modes of the graphical console. With this packet, a CON: window can be converted into a RAW: window and vice-versa, and into a third type of window for which no distinct device name is created.

The same type of mode switch can also be performed for the serial terminal represented by the AUX device, which also exists in three modes in total, though usually only one of them is exposed to the user as AUX:. This packet makes two additional modes available.

dp_Arg1 defines the mode into which the console shall be switched. In total three modes are available, regardless whether the console is a graphical console in a window or a serial console on an external terminal. The modes are defined in table **??** in section **??** and explained in more detail in section **??**.

All other values are reserved for future use; some third party implementations also support a mode 3 that is not documented here. For backwards compatibility, interactive handlers should interpret the value -1 (thus, DOSTRUE) identical to 1, i.e. switch to the unbuffered raw mode.

Before replying this packet, the handler shall set dp_Res1 to DOSTRUE if the mode switch could be performed, and dp_Res2 to 1 if the console is attached to an open intuition window, or to 0 if the window is currently closed or the console operates on top of some other device. Otherwise, if the mode switch is not possible, the handler shall set dp_Res1 to DOSFALSE and dp_Res2 to an error code.

14.8.3 Retrieving IORequest and Window Pointer from the Console

The ACTION_DISK_INFO packet retrieves from a console pointers to its underlying resources.

Table 14.46: ACTION_DISK_INFO

DosPacket Element	Value
dp_Type	ACTION_DISK_INFO (25)
dp_Arg1	BPTR to InfoData
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is not exposed by the *dos.library*; it is the same packet as the one introduced in section **??** and takes the same parameters. However, when applied to consoles, it returns different information and is thus discussed here again.

The `id_DiskType` element of the `InfoData` structure (see section **??**) pointed to by `dp_Arg1` provides information on the mode the console is in:

Table 14.47: `id_DiskType`

Value of id_DiskType	Console Mode
`'CON\0'`	Cooked or medium mode
`'RAW\0'`	Raw mode

Section **??** provides more information on console modes; they describe whether the console buffers entire lines and which key presses are reported how.

The `id_VolumeNode` element of the `InfoData` structure provided in `dp_Arg1` is filled with a pointer to the intuition window the console runs in. For a serial console, or a console running on any other device, this element remains `NULL`.

The `id_InUse` element is filled with a pointer to the `IORequest` structure (see `exec/io.h`) which is used to communicate with the exec device the handler operates on. For a graphical console, this is a `IORequest` to the *console.device*; for the `AUX:` console, it is an `IORequest` to the corresponding device the console runs on, e.g. of the *serial.device*. Other devices are also possible.

Even though this packet provides useful information to the caller, it has several drawbacks and its usage is discouraged. First, as it potentially provides information on the intuition window, a window opened in `AUTO` mode cannot be closed anymore, and neither can it be iconified. This is because both operations would invalidate the window pointer provided by this packet.

Applications should inform the console by sending an `ACTION_UNDISK_INFO` packet at the time the window pointer or the `IORequest` is no longer required. Console windows then regain the `AUTO` and iconification capabilities. `ACTION_UNDISK_INFO` is specified in section **??**.

Second, `ACTION_DISK_INFO` will not provide a window if the console is not running in an intuition window, but remotely over a serial line or any other device. The `IORequest` pointer then corresponds to the target device through which the console communicates with the user, and not to the *console.device*.

Often, applications (mis-)use this packet to retrieve the current cursor position or the dimensions of the window in character positions, assuming that the `IORequest` pointer in `id_InUse` is, actually, corresponding to a *console.device* and as such `io_Unit` of the request is a pointer to a `ConUnit` structure. However, this assumption may not be true, and console dimensions and the cursor position cannot be obtained in general in this way.

The following algorithm provides an alternative by switching the console to raw mode, and requesting the required information through CSI sequences. These sequences operate independently of the device and only require that the local or remote console implements a VT-100 compatible interface.

```
/* Retrieve the window dimensions in characters
** from a console connected to a file handle "file"
*/
void WindowSize(BPTR file,LONG *width,LONG *height)
{
  if (!ParsePosition(file,'r',width,height)) {
    /* Provide a standard console size in
    ** case of failure.
    */
    if (width)
```

```
      *width = 80;
    if (height)
      *height = 24;
  }
}

/* Retrieve the cursor position from a console
** connected to a file handle "file"
*/
void CursorPosition(BPTR file,LONG *x,LONG *y)
{
  ParsePosition(file,'R',x,y);
}

/*
** Maximum time to wait for the console
** to respond in microseconds. May require
** adjustment for slow connections.
*/
#define MAX_DELAY       200000

/* Generic CSI sequence parser for a VT-xxx
** console. Returns TRUE if the sequence could
** be parsed.
*/
BOOL ParsePosition(BPTR file,char answer,LONG *width,LONG *height)
{
BOOL success;
BOOL incsi,innum,negative,inesc;
LONG counter;
LONG args[5];
UBYTE in;

  memset(args,0,sizeof(args));
  SetMode(file,1);

  success  = TRUE;
  incsi    = FALSE;
  inesc    = FALSE;
  innum    = FALSE;
  negative = FALSE;
  counter  = 0;

  /* Now send a window borders or
  ** cursor status request to the stream
  */
  if (answer == 'R') {
    Write(file,"\033[6n",4);
  } else {
    Write(file,"\033[0 q",5);
  }
```

```
/* Parse the incoming string */
for(;;) {
  if (WaitForChar(file,MAX_DELAY) == FALSE) {
    success = FALSE;
    break;
  }

  if (Read(file,&in,1) != 1) {
    success = FALSE;
    break;
  }

  /*
  ** State machine for interpreting CSI or ESC
  ** sequences.
  */
  if (incsi) {
    if ((in<' ') || (in>'~')) { /* Invalid sequence? */
      incsi = FALSE;
    } else if ((in>='0') && (in<='9')) {
      /* Valid number? */
      if (innum == FALSE) {
        innum = TRUE;
        args[counter] = 0;
      }
      args[counter] = args[counter]*10+in-'0';
    } else {
      /* Abort parsing the number. Install its sign */
      if (innum) {
        if (negative)
          args[counter] = -args[counter];
        innum    = FALSE;
        negative = FALSE;
      }
      if ((in>='@') && (in<='~')) { /* End of sequence? */
        /* Is it a bounds report? */
        if ((in=='r') && (answer=='r') && (counter==3)) {
          if (height)
            *height = args[2]-args[0]+1;
          if (width)
            *width  = args[3]-args[1]+1;
          break;
        }
        /* Is it a cursor report? */
        if ((in=='R') && (answer=='R') && (counter==1)) {
          if (height)
            *height = args[0];
          if (width)
            *width  = args[1];
          break;
```

```
          }
          incsi = FALSE; /* Abort sequence */
        } else if (in==';') { /* Argument separator? */
          counter++;
          /* Do not parse more than 5 arguments,
          ** throw everything else away
          */
          if (counter>4) counter=4;
          innum    = FALSE;
          negative = FALSE;
        } else if (in=='-') {
          if (innum)
            incsi = FALSE;  /* minus sign in the middle is invalid */
          negative = ~negative;
        } else if (in==' ') {
          /* Ignore SPC prefix */
        } else {
          /* Abort the sequence */
          incsi = FALSE;
        }
      }
    } else if (inesc) {
      if (in == '[') {
        inesc    = FALSE;
        incsi    = TRUE;   /* found a CSI sequence */
        innum    = FALSE;  /* but not yet a valid number */
        negative = FALSE;
        counter  = 0;
        args[0]  = args[1]=args[2]=args[3]=args[4] = 1;
      } else if ((in >= ' ') && (in <= '/')) {
        /* ignore the ESC sequence contents */
      } else {
        inesc    = FALSE;  /* terminate the ESC sequence */
      }
    } else if (in == 0x9B) {
      incsi    = TRUE;   /* found a CSI sequence */
      innum    = FALSE;  /* but not yet a valid number */
      negative = FALSE;
      counter  = 0;
      args[0]  = args[1]=args[2]=args[3]=args[4] = 1;
    } else if (in == 0x1B) {
      inesc    = TRUE;  /* found an ESC sequence */
    } /* Everything else is thrown away */
  }

  SetMode(file,0);
  return success;
}
```

The above algorithm supports both 7-bit and 8-bit consoles and is aware of the 7-bit two-character equivalent of CSI. It thus cannot only be used for communication with an Amiga console, will work with any VT-100 compatible terminal.

Upon replying ACTION_DISK_INFO, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.8.4 Releasing Console Resources

The ACTION_UNDISK_INFO packet releases any resources obtained by ACTION_DISK_INFO.

Table 14.48: ACTION_UNDISK_INFO

DosPacket Element	Value
dp_Type	ACTION_UNDISK_INFO (513)
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is not exposed by the *dos.library*, the packet must be issued through the DoPkt() function. It was added in AmigaDOS version 45, but was already implemented in some third party handlers before.

The purpose of this packet is to let the console know that the pointers to the window and the IORequest provided by ACTION_DISK_INFO are no longer needed and the console may close the window or release the IORequest if needed. This has the the practical consequence that AUTO windows can be closed again, and the console can also be iconified again.

To implement this packet, the console should keep a counter to track the number of times its resources have been provided to clients. An ACTION_DISK_INFO increments it, and ACTION_UNDISK_INFO decrements it. As long as the counter is non-zero, the window shall remain open and the connection to the device implementing the console functionality shall be established. This includes that the window needs to be forced open on the first ACTION_DISK_INFO if it was closed, either because it is not yet open, is iconified or because it is an AUTO window that has been closed by the user.

This packet does not take any arguments. Even though there is practically no reason why this packet could fail, the console handler shall set dp_Res1 to DOSTRUE and dp_Res2 to 0 on success. On an error, if such an error should be possible, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.8.5 Stack a Line at the Top of the Output Buffer

The ACTION_STACK packet injects a line at the start of the output buffer of the console.

Table 14.49: ACTION_STACK

DosPacket Element	Value
dp_Type	ACTION_STACK (2002)
dp_Arg1	fh_Arg1 of a FileHandle
dp_Arg2	APTR to characters
dp_Arg3	Size of the buffer in bytes
dp_Res1	Number of characters stacked
dp_Res2	Error code

This packet is not exposed by the *dos.library*, it rather must be issued through the packet interface, e.g. DoPkt(). This packet injects a line at the top of the output buffer of the console. This buffer keeps all lines entered by the user, and lines injected by ACTION_STACK and ACTION_QUEUE. The number of lines in this output buffer can be obtained through ACTION_WAIT_CHAR which delivers the line count in dp_Res2. While this packet does not show up in [**?**], it is present since AmigaDOS version 36 and the integration of ARexx.

Lines from this buffer are provided to clients on ACTION_READ requests that empty this buffer line by line. Thus, a line provided through this packet will be delivered to the next reading client, before any other buffered lines, but after all lines entered by the user have been delivered.

Lines injected into the console output buffer are not echoed on the screen. Arexx uses this packet to implement the PUSH instruction which employs the console as "external stack". This packet places a line at the top of this stack.

dp_Arg1 is a copy of the fh_Arg1 element of a FileHandle structure interfacing to the console.

dp_Arg2 is a C pointer (not a BPTR) to an array of characters to be injected into the output buffer of the console.

dp_Arg3 is the size of the buffer in characters, i.e. the number of characters add at the head of the output buffer.

Upon replying this packet, dp_Res1 shall be set to the number of characters that could be stacked in the output buffer of the console, or to -1 in case of failure. On success, dp_Res2 shall be set to 0, or to an error code on failure.

14.8.6 Queue a Line at the End of the Output Buffer

The ACTION_QUEUE packet injects a line at the end of the output buffer of the console.

Table 14.50: ACTION_QUEUE

DosPacket Element	Value
dp_Type	ACTION_QUEUE (2003)
dp_Arg1	fh_Arg1 of a FileHandle
dp_Arg2	APTR to characters
dp_Arg3	Size of the buffer in bytes
dp_Res1	Number of characters stacked
dp_Res2	Error code

This packet is not exposed by the *dos.library*, it rather must be issued through the packet interface, e.g. DoPkt(). This packet injects a line at the end of the output buffer of the console. This buffer keeps all lines entered by the user, and lines injected by ACTION_STACK and ACTION_QUEUE. The number of lines in this output buffer can be obtained through ACTION_WAIT_CHAR which delivers the line count in dp_Res2. While this packet does not show up in [**?**], it is present since AmigaDOS version 36 and the integration of ARexx.

Lines from this buffer are provided to clients on ACTION_READ requests that empty this buffer line by line. Thus, a line provided through this packet will be delivered to a reading client after all other buffered lines have been read from the console, including lines entered by the user.

Lines injected into the console output buffer are not echoed on the screen. Arexx uses this packet to implement the QUEUE instruction which employs the console as "external stack". This packet places a line at the end of this stack, i.e. queues it.

dp_Arg1 is a copy of the fh_Arg1 element of a FileHandle structure interfacing to the console.

dp_Arg2 is a C pointer (not a BPTR) to an array of characters to be injected into the output buffer of the console.

dp_Arg3 is the size of the buffer in characters, i.e. the number of characters to append at the end of the output buffer.

Upon replying this packet, dp_Res1 shall be set to the number of characters that could be queued in the output buffer of the console, or to -1 in case of failure. On success, dp_Res2 shall be set to 0, or to an error code on failure.

14.8.7 Force Characters into the Input Buffer

The ACTION_FORCE packet injects characters into the keyboard buffer of the console, as if the user typed them.

Table 14.51: ACTION_FORCE

DosPacket Element	Value
dp_Type	ACTION_QUEUE (2001)
dp_Arg1	fh_Arg1 of a FileHandle
dp_Arg2	APTR to characters
dp_Arg3	Size of the buffer in bytes
dp_Res1	Number of characters stacked
dp_Res2	Error code

This packet is not exposed by the *dos.library*, it rather must be issued through the packet interface, e.g. DoPkt(). This packet injects characters into the keyboard buffer of the console, at the same place keystrokes are recorded. Such characters are echoed on the console, or executed in case of control sequences. Therefore, this packet also allows to move the cursor left or right, or erase the current line by emulating the corresponding keystrokes. While this packet does not show up in [**?**], it is present since AmigaDOS version 36, though did not work as intended before AmigaDOS version 47.

Note that the keyboard input buffer is different from the line output buffer; lines entering the console through ACTION_FORCE qualify as keyboard input[15]. The main user of this packet is the Shell through which it injects TAB-expanded file names into the console. The ConClip command also uses this packet to insert the paths of icons dropped on the console window.

dp_Arg1 is a copy of the fh_Arg1 element of a FileHandle structure interfacing to the console.

dp_Arg2 is a C pointer (not a BPTR) to an array of characters to be injected into the output buffer of the console. It may contain control sequences.

dp_Arg3 is the size of the buffer in characters, i.e. the number of characters to inject into the keyboard buffer.

Upon replying this packet, dp_Res1 shall be set to the number of characters that could be injected into the keyboard buffer of the console, or to -1 in case of failure. On success, dp_Res2 shall be set to 0, or to an error code on failure.

14.8.8 Drop all Stacked and Queued Lines in the Output Buffer

The ACTION_DROP packet disposes all lines that have been injected into the output buffer and thus reverts any ACTION_STACK and ACTION_QUEUE packets.

Table 14.52: ACTION_DROP

DosPacket Element	Value
dp_Type	ACTION_DROP (2004)
dp_Arg1	fh_Arg1 of a FileHandle
dp_Arg2	0
dp_Res1	Boolean result code
dp_Res2	Error code

[15]Even though the V36 CON-Handler already supported this packet, it is there implemented incorrectly and does not impact the keyboard buffer but the keyboard buffer. This was fixed in V47.

This packet is not exposed by the *dos.library*, it rather must be issued through the packet interface, e.g. `DoPkt()`. It removes any lines injected by `ACTION_STACK` and `ACTION_QUEUE` from the output buffer of the console and thus reverts their effects. Keyboard inputs and the keyboard buffer remain unaffected. Thus, this packet resets the line stack in the console. This packet is designed for compatibility with ARexx, it is, however, not used at the time of writing. While this packet does not show up in [**?**], it is present since AmigaDOS version 36 and the integration of ARexx.

`dp_Arg1` is a copy of the `fh_Arg1` element of a `FileHandle` structure interfacing to the console.

`dp_Arg2` shall be 0. This element is reserved for future use as a priority and shall be zero-initialized for forwards compatibility.

Upon replying this packet, `dp_Res1` shall be set to `DOSTRUE` and `dp_Res2` to 0 on success. On error, `dp_Res1` shall be set to `DOSFALSE` and `dp_Res2` to an error code.

14.8.9 Bring the Console Window to the Foreground

The `ACTION_SHOWWINDOW` activates the console window, if it is open, and brings it to the foreground.

Table 14.53: `ACTION_SHOWWINDOW`

DosPacket Element	Value
`dp_Type`	`ACTION_SHOWWINDOW (506)`
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet is not exposed by the *dos.library*. The packet interface, e.g. `DoPkt()`, is required to issue it. It brings the window associated to the console, should such a window exist, to foreground and activates it. If the console is a serial console or the console window is closed or iconified, no activity is performed and no error is reported. Unlike `ACTION_DISK_INFO`, it will not force the window open if the window is currently closed. This packet was introduced in AmigaDOS version 47.

The primary user of this packet is the `ConClip` tool which sends this packet to make the window visible whenever an icon is dropped on it. The path of the icon is injected into the keyboard input buffer with `ACTION_FORCE`.

Upon replying this packet, `dp_Res1` shall be set to `DOSTRUE` and `dp_Res2` to 0 on success. On error, `dp_Res1` shall be set to `DOSFALSE` and `dp_Res2` to an error code.

14.8.10 Change the Target Port to Receive Break Signals

The `ACTION_CHANGE_SIGNAL` packet sets a `MsgPort` to whose task the console sends break signals generated by the `Ctrl-C` to `Ctrl-F` key combinations.

Table 14.54: `ACTION_CHANGE_SIGNAL`

DosPacket Element	Value
`dp_Type`	`ACTION_CHANGE_SIGNAL (995)`
`dp_Arg1`	`fh_Arg1` of a `FileHandle`
`dp_Arg2`	APTR to `MsgPort` structure
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet is not exposed by the *dos.library*. The packet interface, e.g. `DoPkt()`, is required to issue it. It defines one of the two ports to which break signals as generated by the `Ctrl-C` to `Ctrl-F` key combinations are send. Actually, the port itself is not used as such, but only its `mp_Task` element providing the task to which signals are send. This packet was introduced in AmigaDOS version 36.

The port set by this packet is implicitly overridden and replaced by the port of the next process reading from the console by ACTION_READ. That is reading from the console implicitly requests receiving break signals. Despite to the reading port, the console also sends break signals to the last process that issued an ACTION_WRITE request, provided this is not a shell process running in the background.

The console takes a couple of precautions to avoid trashing memory by attempting to send signals to processes that no longer exist: the console implicitly disables the port set by ACTION_CHANGE_SIGNAL and ACTION_READ or the port set by ACTION_WRITE whenever an ACTION_END is received from a process to which the corresponding port belongs. That is, processes that are closing a stream to the console are removed as candidates for receiving break signals. In addition, before sending a signal, the console tests the validity of the port by testing whether its mp_Task field is known to the exec scheduler. For that, it searches the list of waiting and ready tasks within ExecBase for mp_Task and refuses to send a signal if the task is not found in one of the two lists.

This packet is, for example, used when a shell is started in a console window, or a NewShell command creates a shell in an already open console. In such a case, the System() function as part of the *dos.library* sends a ACTION_CHANGE_SIGNAL to the console to ensure that break signals are received by the new shell just started, and not by the shell which is then running in the background[16]. This packet is *not* send for executables started by the Run command in the background. Even though Run also goes through System(), it uses different parameters that suppresses its generation.

Arguments are populated as follows:

dp_Arg1 is a copy of the fh_Arg1 element of a FileHandle structure that is opened to the console, even though the system CON-Handler does currently not use this argument. Third party handlers may depend on it, however.

dp_Arg2 is a pointer to a MsgPort to whose task break signals will be send. If this argument is NULL, the currently configured port is not changed. As the console shall always return the previously configured port in dp_Res2, this allows clients to retrieve the currently configured break port without changing it.

Before replying this packet, dp_Res1 shall be set to DOSTRUE on success, and dp_Res2 to the port that was previously registered for receiving signals, or NULL if no such port was configured. On error, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

14.9 Packets Controlling the Handler in Total

The packets in this section impacts file systems and handlers at a global level. They do not apply to a particular volume or file system object.

14.9.1 Adjusting the File System Cache

The ACTION_MORE_CACHE packet increases or decreases the number of file system buffers.

Table 14.55: ACTION_MORE_CACHE

DosPacket Element	Value
dp_Type	ACTION_MORE_CACHE (18)
dp_Arg1	Buffer increment
dp_Res1	Total number of buffers or 0
dp_Res2	Error code

This packet implements the AddBuffers() function of the *dos.library* introduced in section **??**. It increases or decreases the number of buffers the file system may use to cache data. How exactly a file

[16] Unlike what [**?**] claims, this packet is not send by CreateNewProc().

system uses these buffers and how large these buffers are is specific to the file system implementation and its configuration.

The FFS uses these buffers to temporary store administration data such as blocks describing the directory structure or information which blocks a particular file occupies on disk. Providing additional buffers (within limits) can thus help to improve the performance of a file system by reducing the number of times the underlying device needs to be contacted. The FFS also uses these buffers to store payload data if the source or destination buffer of the client is not reachable by the exec device driver. The `de_Mask` element of the environment vector is used to determine such incompatibilities, see section **??**.

`dp_Arg1` is the increment (or decrement, if negative) of the number of buffers be added (or removed) from the pool of cache buffers of the file system. File systems may clamp this value to guarantee that a minimum number of buffers are available, or limit the buffer count to a useful value

The FFS versions 34 and below did not support reducing the number of buffers and neither accepted a value of 0 for the buffer increment. This was fixed in AmigaDOS version 36.

If this packet succeeds, the file system shall set `dp_Res1` to the number of buffers currently allocated, and `dp_Res2`[17] to 0. Thus, the current buffer count can obtained by sending this packet with `dp_Arg1` set to 0. On an error, `dp_Res1` shall be set to `DOSFALSE` and `dp_Res2` to an error code.

14.9.2 Inhibiting the File System

The `ACTION_INHIBIT` packet disables or enables access of the file system to the underlying device; once file system access is disabled, application programs such as `Format` may access the underlying device directly.

Table 14.56: `ACTION_INHIBIT`

DosPacket Element	Value
`dp_Type`	`ACTION_INHIBIT (31)`
`dp_Arg1`	Inhibit flag
`dp_Res1`	Boolean result code
`dp_Res2`	Error code

This packet implements the `Inhibit()` function of the *dos.library*, see **??**. An inhibited file system is blocked from accessing its underlying medium for read and write access, with the exception of the `ACTION_FORMAT`, `ACTION_SERIALIZE_DISK` and `ACTION_DISK_INFO` packets which remain operational. The former two even require the file system to be inhibited.

When requesting the disk state with `ACTION_DISK_INFO`, an inhibited file system changes its disk type `id_DiskType` in the `InfoData` structure to 'BUSY', see table **??** in section **??**.

Once the file system is uninhibited again, it shall perform a validation of the volume as if it has been re-inserted. This is necessary because an application bypassing the file system to access the volume directly could have changed the file system structure, the volume name or the date, or may have even written a completely new file system structure on it. This check thus implies verifying the flavor of the file system, e.g. the `DosType` for the FFS, and potentially creating and inserting a `DosList` entry into the device list representing the volume. Thus, first inhibiting and uninhibiting a file system is equivalent to simulating a medium change, which is also how the `DiskChange` command operates.

`id_Arg1` is a Boolean indicator that defines whether the file system shall be inhibited or uninhibited. If `dp_Arg1` is `DOSFALSE`, the file system is uninhibited and a file structure check shall be performed.

[17] The *dos.library* autodocs and [**?**, **?**] document that the file system shall return the buffer count in `dp_Res2` instead. However, the current version of the FFS returns it in `dp_Res1`, and the `AddBuffers` command depends on this — possibly erroneous — implementation. However, as programs seem to depend on this behavior, it is recommended to accept this deviation from official sources as specification change.

In all other cases, the file system shall be inhibited and, with the exception of ACTION_FORMAT and ACTION_SERIALIZE_DISK, should refrain from accessing the medium or partition.

Before replying this packet, dp_Res1 shall be set to a Boolean success indicator. On success, it shall be set to DOSTRUE and dp_Res2 shall be set to 0. On failure, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code. A reason for failing to inhibit a volume is for example if any files are open for writing, and thus the bitmap is inconsistent. Inhibiting in such a case could damage the file system structure.

14.9.3 Check if a Handler is a File System

The ACTION_IS_FILESYSTEM tests whether a handler is a file system.

Table 14.57: ACTION_IS_FILESYSTEM

DosPacket Element	Value
dp_Type	ACTION_IS_FILESYSTEM (1027)
dp_Res1	Boolean result code
dp_Res2	Buffer count or error code

This packet implements the IsFileSystem() function of the *dos.library* as introduced in section **??** and tests whether a particular handler provides sufficient services to operate as a file system. This packet was introduced in AmigaDOS version 36.

A file system shall be able to access multiple separate files, shall be able to support locks and shall also be able to examine directories such that commands like List are able to show directory contents. A file system may be either a flat or a hierarchical file system, i.e. file systems are not required to support multiple directories per volume.

This packet does not take any arguments. It shall set dp_Res1 to DOSTRUE and dp_Res2 to 0 in case the handler qualifies as a file system. In case the handler supports this packet but does not implement a file system, it shall set dp_Res1 to DOSFALSE and dp_Res2 to 0.

If the handler sets dp_Res1 to DOSFALSE and dp_Res2 to ERROR_ACTION_NOT_KNOWN, then this is an indication that the handler cannot interpret this particular packet. If this secondary result code is received by the IsFileSystem() function, it falls back to a heuristic for determining whether a handler is a file system: it uses Lock(":",SHARED_ACCESS), i.e. ACTION_LOCATE_OBJECT to obtain a lock on the file system root. If successful, then the *dos.library* assumes that the handler is actually a file system. An approximation of this algorithm is found in section **??**

14.9.4 Write out all Pending Modifications

The ACTION_FLUSH packet instructs the file system to write all pending or cached modifications out to the medium[18].

Table 14.58: ACTION_FLUSH

DosPacket Element	Value
dp_Type	ACTION_FLUSH (27)
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is not exposed by the *dos.library* and thus can only be send manually through the packet interface, e.g. DoPkt(). File systems may cache writes in order to improve their performance, and may defer write operations an arbitrary amount of time. This packet instructs the file system to write out all modifications immediately.

[18]Users of Unix like systems may consider this as the AmigaDOS equivalent of sync.

This packet shall not be replied before all changes have been written back, the buffers at exec device level have been flushed to the physical layer, e.g. by a CMD_UPDATE, and if applicable, the drive motor has been turned off. However, FFS versions of 40 and before replied ACTION_FLUSH immediately and thus this packet only established a write barrier. This was fixed in AmigaDOS version 43.

Even though it is unclear how clients could make use of the result code, file systems should set dp_Res1 to DOSTRUE and dp_Res2 to 0 on success[19]. In case of error, file systems should set dp_Res1 to DOSFALSE and dp_Res2 to an error code.

14.9.5 Shutdown a Handler

The packet ACTION_DIE requests a file system unmount its volumes and terminate.

Table 14.59: ACTION_DIE

DosPacket Element	Value
dp_Type	ACTION_DIE (5)
dp_Res1	Boolean result code
dp_Res2	Error code

This packet is not exposed through the *dos.library*; rather, the packet must be send through the packet interface, i.e. through DoPkt(). It requests the file system to release all resources, set the dol_Task element of the DosList structures that point to (one of its) ports(s) in the device list to NULL, and then terminate the process.

A file system shall not exit as long as resources cache copies of its MsgPort(s); this is typically the pr_MsgPort of the handler process which is released by AmigaDOS as soon as the process exits. This port would become unusable on exit, and an attempt to contact the file system through it would crash the system. Resources that contain a pointer to the (or a) file system port are file handles, locks or NotifyRequest. If any of them are still active, the file system or handler cannot exit, and ACTION_DIE shall fail.

If none of these resources exist, as next step, references to the file system port(s) shall be removed from the device list. To gain access to the device list, the file system should use AttemptLockDosList() as a blocking semaphore may cause a deadlock. If the list is not accessible at this time, the file system cannot safely shut down and ACTION_DIE shall also fail. If the device list is accessible, the file system shall set the dol_Task elements of the volumes it administrates and the device node it is represented by to NULL. This will request AmigaDOS to launch a new file system from the dol_SegList segment as soon as a client attempts to access it through its device name.

Unfortunately, even these precautions are not fully sufficient to safely unmount a device because client programs can request a handler message port via GetDeviceProc() or DeviceProc() any time for direct communication and buffer the returned port. A file system has no direct control of where or whether cached copies of its port(s) are held, and thus whether it can safely terminate. This limits the utility of this packet to system debugging tools.

In case the file system can foresee that it cannot safely exit at the time the request is made, this packet shall fail by setting dp_Res1 to DOSFALSE and dp_Res2 to an error code, e.g. ERROR_OBJECT_IN_USE. If termination is seemingly possible, this packet shall be replied by setting dp_Res1 to DOSTRUE and dp_Res2 to 0.

After unmounting the volume and replying to this packet, the MsgPort(s) of this handler or file system used for packet communication can still contain packets that were submitted after ACTION_DIE has been received. In order to avoid a deadlock or a crash, the file system shall still reply these packets, for example with the default result codes from table **??** in section **??**; additional details on how to terminate a handler are found in section **??**.

[19]The official reference [**?**] only states that dp_Res1 shall be set to DOSTRUE. This is probably what most AmigaDOS file systems implement, though the value of unconditionally indicating success is questionable.

Some handlers do not initialize the dol_Task element of its DosList entrie(s) with an address of their port(s); interactive handlers such as the CON-Handler are typical examples. Such handlers shall terminate themselves without requiring an ACTION_DIE when the last resource opened to them had been shut down, i.e. all file handles have been closed, all locks (if they are supported at all) have been unlocked and all NotifyRequests (if supported) had been canceled.

14.9.6 Do Nothing

The ACTION_NIL packet performs no activity.

Table 14.60: ACTION_NIL

DosPacket Element	Value
dp_Type	ACTION_NIL (0)
dp_Res1	DOSTRUE
dp_Res2	0

This packet is not exposed by the *dos.library*. It does not perform any activity. A handler or file system implementation should indicate success when replying this packet. Another reason why a seemingly ACTION_NIL packet can appear at a handler MsgPort is because the startup package has its dp_Type set to 0, too, and thus identifies itself also as ACTION_NIL. While ACTION_NIL does not provide any arguments, the startup package does, see section **??**.

There is no reason why this packet should actually fail, unless a handler implements startup handling through the packet type ACTION_NIL. On success, dp_Res1 shall be set to DOSTRUE and dp_Res2 to 0. On an error dp_Res2 shall be set to DOSFALSE and dp_Res2 to an error code.

14.10 Handler Internal Packets

The packets in this section shall never be sent to a handler or file system as part of an application request. Instead, these packets are only used internally within the handler, and are rather an implementation trick to uniform the handler event processing. While regular DosPackets are carried by an exec message, the packets listed here are linked to an IORequest (see exec/io.h). This allows using a shared port, typically the pr_MsgPort of the handler process, for receiving replied IORequests and incoming packets as both appear as DosPackets.

For such packets, the io_Message.mn_Node.ln_Name points to the DosPacket structure, and its dp_Link elements points back to the IORequest. Handlers dispatch these IORequest to exec devices asynchronously with SendIO(), and continue processing requests while the exec device is busy with the requested input or output command. Due to its linkage to a DosPacket, a completed IORequest is received by the packet interface of the handler as one of the packet types listed here.

14.10.1 Receive a Returning Read

The ACTION_READ_RETURN packet is part of a returning CMD_READ or similar input IORequest sent to an exec device and indicates that reading data completed.

Table 14.61: ACTION_READ_RETURN

DosPacket Element	Value
dp_Link	APTR to IORequest
dp_Type	ACTION_READ_RETURN (1001)

This packet does *not* constitute a request to the handler, it rather indicates that the IORequest pointed to by dp_Link completed and the requested data is now available. As such, this packet shall never be replied.

14.10.2 Receive a Returning Write

The ACTION_WRITE_RETURN packet is part of a returning CMD_WRITE or similar output IORequest sent to an exec device and indicates that writing data completed.

Table 14.62: ACTION_WRITE_RETURN

DosPacket Element	Value
dp_Link	APTR to IORequest
dp_Type	ACTION_WRITE_RETURN (1002)

This packet does *not* constitute a request to the handler, it rather indicates that the IORequest pointed to by dp_Link completed and data had been written out. As such, this packet shall never be replied.

14.10.3 Receive a Returning Timer Request

The ACTION_TIMER packet is part of a returning TR_ADDREQUEST, a *timer.device* request to wait for a certain time span.

Table 14.63: ACTION_TIMER

DosPacket Element	Value
dp_Link	APTR to IORequest
dp_Type	ACTION_TIMER (30)

This packet does *not* constitute a request to the handler, it rather indicates that the timerequest pointed to by dp_Link completed and the requested time span passed. As such, this packet shall never be replied. The FFS uses this packet to turn off the drive motor after the last read or write operation, and interactive handlers such as the console use it to implement ACTION_WAIT_CHAR.

14.11 Obsolete and Third-Party Packets

The following packets are not implemented nor defined by AmigaDOS but are either obsolete or currently in use by some third-party handlers. Their functionality is beyond the scope of this work. This list does not claim to be complete:

Table 14.64: Some Third Party Packets

dp_Type	Purpose
ACTION_GET_BLOCK(2)	obsolete, used by the BCPL file system
ACTION_SET_MAP(4)	obsolete, used by the BCPL file system
ACTION_EVENT(6)	obsolete, used by the BCPL file system
ACTION_DISK_TYPE(32)	obsolete, used by the BCPL file system
ACTION_DISK_CHANGE(33)	not used, not related to C:DiskChange
ACTION_VDU (992)	Tripos legacy virtual terminal support
ACTION_SETVDU (993)	Tripos legacy virtual terminal support
ACTION_SC_WRITE (996)	Tripos legacy terminal support
ACTION_SC_MSG (997)	Tripos legacy terminal support
ACTION_DUMMY (1000)	Tripos legacy, not used in AmigaDOS

ACTION_FINDREADONLY(1009)	open a file for exclusive read access
ACTION_FINDONEWRITER(1010)	open a file for exclusive write access
ACTION_WREIG (1011)	Tripos legacy streamer support
ACTION_REWIND (1012)	Tripos legacy streamer support
ACTION_DIRECT_READ(1900)	used by the CDTV
ACTION_DOUBLE(2000)	create two file handles for a pipe
ACTION_PEEK(2005)	retrieve the current console input line
ACTION_REPLACE(2006)	replace the current console input line
ACTION_GET_HISTORY(2007)	retrieve the history of the console
ACTION_HANDLER_DEBUG(2010)	install a debug hook into a handler
ACTION_SET_TRANS_TYPE(2011)	configure LF/CR translation
ACTION_NETWORK_HELLO(2012)	network handler support package
ACTION_JUMP_SCREEN(2020)	move console to another screen
ACTION_SET_HISTORY(2021)	update or set console history
ACTION_GET_DISK_FSSM(4201)	get FileSysStartupMsg structure
ACTION_FFS_INTERNAL1(7654)	was reserved for the v37 FFS
ACTION_FFS_INTERNAL2(8765)	was reserved for the v37 FFS
ACTION_TOGGLE_INTL(331122)	reserved to toggle the international mode

Chapter 15

The AmigaDOS Shell

The Shell is the command line interpreter of AmigaDOS and implements a simple language. User applications can use services of the shell by requesting it to interpret a shell script or launching a new shell in a user-provided console window, then interpreting commands entered by the user. The latter is called an *interactive* shell, the former is non-interactive.

The Shell is built into the Kickstart ROM, even though AmigaDOS is flexible enough to allow custom shells and make them available to user applications. The Shell is also responsible for booting up the system by executing the `Startup-Sequence` script in the `S` assign.

15.1 The Shell Syntax

The Shell reads *commands* along with its *arguments* line by line from the console or a script. Each line consists of at least one command and its arguments, all separated by spaces or tabulators. Each argument is either a value, e.g. a file name, a string or a number — or a pair of a keyword and a value. The keyword may be either separated by blank spaces or tabulators from its value, or by an equals sign ("="), without any spaces or tabulators in between. Many shell commands use the argument parser from section **??** and thus use the syntax implied by the `ReadArgs()` function described above, even though commands can use other means for parsing their arguments.

The *command* refers to an alias, or a script or executable file within the *path*. The path is a list of directories the Shell searches for scripts or executables; it always contains the current directory and the `C` assign, though additional directories can be added or removed with the `Path` command any time. Aliases (see **??**) are simple Shell macros that expand to other commands and perform simple argument injection and reordering.

The following sections describe the Shell syntax in detail. Even though it stems from the Tripos system, AmigaDOS added over the years features such as variable substitution, compound commands including pipes, back-tick expansion and additional redirection operators.

15.1.1 Input/Output Redirection

Commands receive from the Shell an input and output stream, and a console handler to which commands can refer by the special file name "$\star$". Optionally, commands can also be provided with an error stream through which diagnostic messages are printed. The Shell allows redirecting or creating these streams by operators on the command line that shall also be separated by the spaces or tabulators from the arguments and the command. In the following, the term "*path*" is an absolute or relative path to a file or device in the sense of chapter **??**, that is, a path name that can be opened to provide alternative input, output or error streams.

The Shell supports the following redirection operators:

>*path* redirects the output of a command to a particular file, creating the file if it does not yet exist, or overwriting it if does.

>>*path* appends the output of a command to a file if the file already exists, or creates the output file if it does not yet exist.

<*path* redirects the input of a command from an already existing file which must exist; otherwise, an error is reported.

<<*ind* uses the current console or the shell script from which the command is run as input, line by line, until a line starting with the indicator `ind` is found. The indicator can be chosen arbitrarily. A typical choice is `EOF`. This avoids creating temporary files for providing longer inputs, the input is instead taken from the script containing the command. This operator was added in AmigaDOS version 45.

<>*path* redirects both the standard input and the standard output of the command to a file. The file must exist already, otherwise an error is reported. This can be used, for example, to detach a command completely from the console and redirect input and output to `NIL:`.

*>*path* redirects the error output of the command to the given file. If the path identifies an interactive stream (see section **??**), such as a console, then the console process of the command is also set to the handler of the path. Changing the console process will also redirect the output of commands that print errors to the * file representing the console. This redirection operator was also added to AmigaDOS version 45. The syntax of this redirection operator is a reminder that the console, i.e. the file "*", is redirected.

*>>*path* appends the error output of a command to a file, or creates a new file for error output if it does not yet exist. If the target file is interactive, the console process of the command is also updated. This redirection operator was added to AmigaDOS version 45.

*<> creates a standard error stream and redirects it to the standard output, that is, errors go into the same file into which command output is redirected. This operator does not take a path as argument and stands alone. If none of the three above redirection operators are present, then (unlike in other operating systems) the command *does not* receive a standard error stream. What happens in such a case with error output is specific to the implementation of the command. The system error handling functions will still redirect error output to the standard output as if an implicit *<> would be present. This redirection operator was introduced in AmigaDOS version 45, too.

Where the redirection operator is placed within a command line does typically not matter, e.g. whether it is directly following the command or placed at the end of the command line. There are, however, three exceptions the Shell makes when parsing the command line:

For the `Alias` and `Run` command, redirections of the command itself shall be placed directly following the command, i.e. directly behind `Run` or `Alias`. The same applies to all other commands if the shell variable `oldredirect` is set to `on`. This enables the legacy AmigaDOS 33 redirection syntax.

In cases described above, redirections placed at a later point of the command line will become part of the command line arguments. That is, for `Run`, the command run in background will redirect its streams rather than the output of the `Run` command, and for `Alias`, the redirection will become part of the alias and will become active when the alias is expanded. Enabling the `oldredirect` variable will do likewise for all other commands which then need to parse redirection operators manually as they appear as part of their command line arguments. Most executables are not prepared to interpret redirection operators, however, and it is thus recommended *not to set* the `oldredirect` variable.

15.1.2 Compound Commands and Binary Operators

Starting with AmigaDOS version 45, the Shell allows combining multiple commands on the command line, then forming *compound* commands. The following operators go *between* commands such that the last argument of the first command is directly to the left of such an operator, and the next command follows to the right of the operator. As for arguments, these operators need to be separated by blank spaces or tabulators from the surrounding commands or arguments:

`|` The vertical bar creates a pipe such that the standard output of the command to the left becomes the standard input of the command to its right. As some commands require an explicit file name they read data from, or they write data to, the pipe itself can be explicitly addressed through the `PIPE:` file name, both for the reading and the writing end. Pipes require the Queue-Handler (see section **??**), which is mounted by the System-Startup module described in section **??** during bootstrapping the system[1].

`||` Two vertical bars concatenate the output of two commands together into one common stream which can then redirected to a common output[2].

`&&` Sequences two commands, first executes the one to the left, and if its result code is below the `FailAt` value, continues to execute the command to its right.

The brackets "`(`" and "`)`" group commands and execute the grouped command in a separate process in a sub-shell. Similar to all other operators listed in this section, both the opening and closing bracket shall be separated by spaces or tabulators from commands and arguments as otherwise the shell interprets them as part of the command or arguments. Brackets provide logical grouping of commands into compound commands.

Thus, for example, the command line

```
List | More
```

will pipe the output of the `list` command through the pager, and the command line

```
echo "Current Directory Content:" || List | More
```

will print an additional title on top which will also be run through the pager.

The command line

```
( List && Dir ) >RAM:Listing
```

will run both the `List` and `Dir` command on the current directory one after another, and the joined output of both commands, as run in a sub-shell, will be redirected into a file in `RAM:`. Note that spaces are placed between the brackets and the commands as otherwise they would be parsed as part of the command name.

15.1.3 Unary Shell Operators

In addition, the Shell recognizes two unary operators that do not stand between commands, but only at their end or at the end of the command line.

`&` The run-back operator works similar to the `Run` command, it starts the command line within which it appears in background. The command will receive a new logical console, and its output and error streams will be redirected to this console, unless additional operators redirect the streams to other files explicitly. Unfortunately, at this time, the ROM-based CON-Handler does not take advantage of this information and will simply merge the output of the process run back with that of the regular shell, though third party handlers use this information for job control and hold the output of the background process until moved explicitly to the foreground. This operator was introduced in AmigaDOS version 45.

[1] With some tweaks and limitations, pipes could already be made somewhat working in earlier AmigaDOS versions. Full pipe support was introduced in version 45.

[2] This syntax is different from the `bash` operator that looks similar.

+ The plus operator shall only appear at the end of a command line. If it is present, it injects a line-feed and the following line of the current input of the Shell into the argument line of the command. It thus forms an argument string that consists of multiple lines. Only very few commands can actually process a line-feed as part of their argument line, most will ignore all characters behind the first line feed. One particular command that supports such argument lines is `Run`. It feeds its argument line unprocessed into a new non-interactive Shell as command stream, building it from a string using a technique similar to that explained in section **??**. This sub-shell will thus receive a shell script as input which consists of multiple lines, and will execute the commands one after another. In combination with the `Run` command, the plus ("+") operator therefore indicates that the following line forms an additional command line of a shell script that is to be executed in background.

; The semicolon ends a command line and starts a comment. All characters beyond the semicolon are ignored.

Thus, for example, the following two lines entered on the console

```
Run List +
Dir
```

run the `List` command in background, which is followed by the `Dir` command, also run in the same background shell after the `List` command finished. The same can be accomplished, though in a more compact form, by the single line

```
( List && Dir ) &
```

15.1.4 Quoting and Escaping

To allow spaces, tabulators, equals-signs and the above operators within command names and arguments, e.g. to handle file names containing such characters, the Shell offers quoting. Blank spaces, tabulators, semicolons, the above operators and equals-signs become part of the command or argument and are not interpreted by the Shell if they are enclosed in double quotes ("`"`"). A quote only starts a quoted argument or command and is thus a functional element of the Shell if it is at the start of the line or preceded by a blank space or a tabulator; quotes *within* an otherwise unquoted argument are literals and stand for themselves. A quote, however, does terminate a quoted string even if it is *not* followed by a blank space or tabulator.

The Shell recognizes the asterisk "`*`" as an escape character, and depending on the escape sequence, it is either active in any place, or only recognized within double quotes. The escape sequences supported by the Shell can be roughly broken up into two classes:

The first class of escape sequences is only substituted within double quotes. These legacy escape sequences are recognized by the Shell, but the actual substitution of the escape sequence by the escaped character is left to the executing program and thus happen *outside* of the Shell.

`*N` The newline character, ASCII `0x0a`.

`*E` The ESC character, ASCII `0x1b`. There is no escape sequence for the CSI control character `0x9b`, but there is a 7-bit equivalent sequence all terminals support, namely `*E[`. The Shell does not establish this equivalence, but the console does.

`*"` The double quote; a double quote escaped as such does not terminate the quoted argument, but represents the quote itself as part of the argument.

`**` The asterisk itself.

As the Shell does not perform substitutions for the above sequences, they appear within the command line as received by the executed program, which is then responsible to perform the above substitutions correctly itself. Thus, the command

```
Echo "Hello*NThere"
```

prints the word `Hello` on one line, and `There` on a new line below, though the `Echo` command receives the argument line `"Hello*NThere"` including the quotes and the escape sequence. Replacement of "`*N`" with the line feed happens inside of the `Echo` command, or more precisely, within the `ReadArgs()` function called by it.

The above escape sequences are *not* substituted outside of double quotes, and therefore, are literals there. This allows, for example, to use "`*`" as file name — without quotes — representing the current code. The command

```
Echo Hello*NThere
```

therefore prints `Hello*NThere` on the console, without any line feed, unchanged.

Proper interpretation and escape sequence substitution is ensured if the command uses the `ReadArgs()` or `ReadItem()` functions of the *dos.library*, see section **??** and following. This distribution of responsibilities has the inconvenient side effect that some third-party argument parsers, e.g. the ones provided by some C compilers, do not fully support the (admittedly unorthogonal) quotation and escaping rules of the Shell and thus do not deliver the expected results.

The second class of escape sequences is transparently substituted by the Shell, and thus no additional burden arises for the executed program to interpret them[3]. Unlike the above, the following sequences are *also* interpreted outside of quotes, everywhere on the command line:

`*$` The literal dollar sign. The non-escaped dollar sign is a syntax element that indicates variable expansion, see section **??**.

`*`` The literal back-tick. The non-escaped back-tick is a syntactical element for command output substitution, see section **??**.

`*[` The literal opening square bracket. Square brackets are part of the alias syntax, see section **??**, and this sequence inserts a bracket itself if it is part of an alias. This sequence can only be used within aliases, but has no special meaning outside of them.

`*]` The literal closing square bracket. Similar to the above, this sequence is only recognized during alias expansion and inserts a literal square bracket into the expanded alias.

The following examples illustrate the consequences of the above syntax. While

```
Echo *$
```

prints a `$` sign, even the asterisk is outside of quotes, the same asterisk upfront an opening square bracket

```
Echo *[
```

prints `*[` because the command is not part of an alias substitution. However, if the sequence is part of an alias definition such as

```
Alias PrintBracket Echo *[
```

then this creates an alias that prints `[` because *now* the bracket is escaped by an asterisk and not interpreted as a syntactical element of the alias definition.

[3]This inconsistency can probably be only understood in historical context as variable expansion and back-tick substitution were later Amiga specific extensions to the original Tripos Shell.

15.1.5 Variables and Variable Expansion

The AmigaDOS Shell supports variables, both local to the Shell and system-global, and expands variables as part of the regular command line parsing. A variable is indicated by a string starting with a dollar-sign ("$") and followed by alpha-numerical characters, i.e. 0-9, A-Z and a-z. Variable names are case-insensitive. The last character that is outside this range terminates the variable name.

An equivalent but more flexible way of referring to a variable is by enclosing the variable name after the "$" sign in braces, i.e. "${*name*}" within which all other characters except the closing brace are allowed as components of the name as well. In particular, variables can be grouped in a directory-like hierarchy where components are separated by forward-slashes ("/") as in paths. Section **??** provides more information on how to access shell variables from program code without requiring variable expansion of the Shell.

Local variables are represented as entries in the `pr_LocalVars` list that is part of the process structure that executes the Shell, see chapter **??** and section **??**. When expanding variables, the Shell first checks there for a variable of a matching name. Only if that fails, the Shell tests for global variables. They are represented as files in the `ENV` assign, which is typically an external link within the `RAM` device copying elements from the `ENVARC` assign. That is, local variables take priority over global variables.

Variables are substituted by the Shell *before* the resulting argument line is provided to the command, i.e. it is not necessary to expand variables within commands. The Shell does not attempt to generate pairs of double quotes to ensure that an expanded variable corresponds to a single argument. In fact, if a variable contains spaces, tabulators or equals-signs and it is not included in double quotes, it will appear in the expanded command line as multiple arguments. However, if a variable contains asterisks or double quotes, and the variable appears within double quotes, such asterisks or double quotes are escaped by the Shell with an (additional) asterisk as escape character to ensure the resulting argument is represented appropriately.

If an attempt is made to expand a variable that is not defined, then the variable is not expanded at all. In such a case, the entire expression, including the "$" sign and optional braces stand for themselves and become the argument.

If the variable name is started with "$?" instead of "$", then the Shell will check whether the variable exists or not, and instead of inserting its value, it will substitute the name by a 1 if the variable exists, or by a 0 if it does not exist, i.e. is undefined. If a variable reference starts with "$??", the Shell checks whether the variable exists as *global* variable, and if so, expands the entire reference to 1, otherwise to 0. Finally, if the variable reference starts with "$!", the variable may contain control characters such as line feeds that are included in the expansion. This binary expansion is only applicable to global variables, and as potential control characters are injected into the Shell tokenizer, it is necessary to include such binary expanded variables in double quotes. This ensures that the Shell escapes the control characters properly and makes them accessible to the command and its argument parser.

15.1.6 Pre-defined Shell Variables, Configuring the Shell

Some variables are built into the Shell and either allow to configure features of the Shell, or are filled by the Shell with status information on the recently executed command. They are listed in the following *without* a leading `$` sign required to expand them to their contents:

`process` is filled with the CLI task number. Each Shell is assigned a unique small integer to which `process` is expanded. This is the same number found in `pr_TaskNum` of the process structure of the Shell (see chapter **??**) and the process number shown by the `Status` command. It can be used, for example, to generate unique file names for temporary files.

`RC` is the return code of the previously run command. It can be used to check whether the last command failed or not. The Shell compares this result with the value set by the `FailAt` command and if it is greater or equal than the threshold, a Shell script is aborted. This result code is also found in the `cli_ReturnCode` element of the CLI structure, see section **??**.

`Result2` is the error code the last executed command returned in case of failure. It is a code from the list given in section **??**. The `Why` command prints a textual description of this error code. This variable corresponds to the `cli_Result2` element of the CLI structure, more details are given in section **??**.

The following Shell variables are not set by the Shell, but only read by it. They configure the Shell and some of its features. Some of these variables take Boolean values. In such a case, the strings "`on`", "`1`", "`yes`" or "`TRUE`" indicate an enabled feature, all other values disable the corresponding feature. The comparison is not case-sensitive.

`VIEWER` is the path of a program that is used to show non-executable files. If this viewer is set to the path of the `MultiView` program, then just typing the name of a known datatype shows its contents. If this variable is left undefined, attempting to execute a non-executable file as command results in an error. Section **??** provides more details.

`echo` is a Boolean variable that, if set, echos the currently executed command on the console. This is possibly helpful for debugging Shell scripts, though `interactive` is probably an even more useful tool, see below.

`debug` is a Boolean variable that enables writing all executed commands over the serial port, at 9600 baud, 8 bits, no parity, 1 stop bit. Multiple debugging tools are available to redirect this debug output into a file. If logging is enabled in the boot menu, the `syslog` ROM module will capture these outputs and redirect them to `RAM:syslog`. Section **??** provides more background on this feature.

`oldredirect` changes the syntax of the redirection operators introduced in section **??**. If this Boolean variable is enabled, then redirection operators are only recognized if they follow the command immediately. If they are somewhere else on the command line, they become part of the regular arguments of a command. Thus, for example, "`List >Out C:`" will continue to redirect the output of the `List` command, though "`List C: >Out`" will not. Instead, it will attempt to list the "`C:`" assign along with a file or directory named "`>Out`" where the "`>`" character is part of the file name. This emulates the AmigaDOS 33 BCPL CLI syntax. Note that enclosing a redirection operator in double quotes also disables its function regardless of this variable, and makes it a literal symbol that becomes part of a command line argument.

`interactive` enables interactive tracing of Shell scripts. This variable can also be set by the boot menu to enable debugging of the `Startup-Sequence`, see also section **??**. If this variable is set, the Shell prints each command prior execution, and waits for a keyboard command. The return key, or "`y`" then executes the command, the delete key or "`n`" skips over the command, and the escape key or "`q`" disables tracing and deletes this variable.

`simpleshell` disables TAB expansion through the console medium mode, see section **??**. Instead, the Shell will only set the console to cooked mode and read entire lines. The history and file name expansion by the TAB key are then not available anymore, and the Shell falls back to pre-AmigaDOS 45 console usage. As of AmigaDOS 47, the history is no longer part of the console, but was migrated into the Shell, and thus is then neither available anymore.

`histsize` sets the size of the history in lines. This variable is only used if the Shell does actually keep a history, and thus requires the `simpleshell` variable to be either undefined or off.

`histskipdups` is a Boolean variable that, if set, instructs the Shell not to keep duplicate commands in its history. If a command is executed a second time, this repeated command does not become an identical second entry on the history. This variable is only recognized if the Shell is actually keeping a history, which it does not if `simpleshell` is enabled.

15.1.7 Backtick Expansion

After variable expansion, the Shell checks the command line for back-ticks, ("`` ` ``"). The characters within two terminating back-ticks form a command itself that is executed *before* interpretation of the containing command line continues, and the standard output of the enclosed commands is substituted for the command

line within the back-ticks. As this output may include line feeds, these are furthermore substituted by blank spaces.

Back-tick expansion was already available in AmigaDOS version 36, but was restricted to only a single pair of backticks per command line. This restriction was lifted in AmigaDOS version 45.

Note that the result of the backtick expansion may even contain functional syntax elements of the Shell, e.g. angle brackets in the output of a command in back-ticks will become Shell stream redirection operators which are then further interpreted by the Shell. Needless to say, this can cause bad surprises.

The back-ticked sequence can itself be enclosed in double quotes. While this avoids the above surprise, it will necessarily only generate a single logical argument. Only in that case, namely a back-ticked sequence contained in another pair of double quotes, the Shell performs backwards escaping of asterisks and double quotes — as for variable expansion. Each double quote or asterisk in the output of the quoted back-ticked command is escaped with (another) asterisk. This ensures that the resulting output string within double quotes retains its original value once interpreted through `ReadItem()` or `ReadArgs()`. Note that backwards-escaping does not take place without the additional layer of double quotes surrounding the back-ticked command, same as in variable substitution.

15.1.8 Alias Substitution

The next step in Shell processing is alias expansion. Aliases are simple macros that replace the alias with another command with the option of rearranging arguments. Before the Shell searches for a command in the path, it checks whether it constitutes an alias and then expands it.

Aliases are always local to the executing Shell, AmigaDOS keeps them in the same list as local shell variables. If the command name matches an alias, its name is removed, and replaced by the contents of the alias. If this contents contains a pair of opening and closing square brackets, that is the sequence "`[]`", then the brackets are replaced by all remaining arguments of the command. The result of alias expansion is another command that undergoes alias expansion again, except that the same alias cannot be used twice in the same command line to avoid endless recursion.

For example, given the following alias definition

```
Alias XType Type [] hex
```

a new alias `XType` is defined such that the command

```
XType S:Startup-Sequence
```

is expanded into

```
Type S:Startup-Sequence hex
```

which prints a hex-dump of its argument on the console. Thus, aliases are useful to create simple one-line command scripts. Complex command sequences also requiring argument parsing are, however, the domain of shell scripts. Section **??** provides more details on the `Execute` command through which such scripts are processed.

15.1.9 Command Location and Execution

The purpose of the next step is to identify from a command name a segment containing machine code which is executed to implement the function of the command.

If the command name is not quoted, the Shell first attempts to locate the name on the list of resident segment, see section **??**. If a match is found, its use count `seg_UC` is incremented and the segment `seg_Seg` contains then the code that will be executed. If no match is found there, or the command name is quoted, then the Shell keeps looking in the current directory for a file system object matching the command name —

at this stage, both files and directories are accepted. Note that unlike under Unix like systems, the current directory is always scanned first, and there is no option to disable searching it.

If the command could not be located there and does not include a directory name, the Shell keeps scanning all directories in the *path*, though from this point on, only files match. The path is a list of directories containing commands, and is adjusted by the `Path` command, see also section **??** how it is represented internally.

If nothing is found in the path, the Shell finally checks the `C` assign. The directory — or directories, in case it is a multi-assign — the assign points to is also always implicitly contained in the path and, similar to the current directory, cannot be removed from it.

If neither a resident segment nor a file or directory could be found, the Shell indicates failure due to an unknown command.

If, however, a file system object matching the command name was successfully located, the Shell checks whether it is a file or a directory. In the latter case, the Shell inserts a `CD` command upfront, and restarts the process of locating the command. Thus, directories found as matches are executed as arguments of the `CD` command which exists as Shell built-in on the list of resident segments.

If the result is a file, the Shell continues to check the protection bits of it, see section **??** for their value. If the `s` protection bit is set, the Shell assumes that command is actually a script. If this script starts with the magic character sequence "`/*`", it is assumed to be a Rexx script and an implicit `RX` command is prepended to the file name, and command location starts again with the updated line. If the script starts with the two-character sequence "`#!`" or "`;!`", then the remaining first line of the script file contains the path of the command interpreter which is then also injected as command name, again restarting command location. If none of the two conditions hold, the Shell assumes this file is a Shell script, prepends an `Execute` command upfront the script name and again restarts command location. Thus, with the `s` bit set, either the Rexx interpreter, a custom interpreter or the Shell itself — through the `Execute` command — runs the script. More on the latter command is in section **??**. Implicit execution of ARexx scripts was added in AmigaDOS version 37, the option of having custom interpreters was added in version 45.

If the `e` protection bit is set indicating an executable file, the Shell attempts to load the file through `LoadSeg()` returning a segment for execution. If this succeeds and, in addition, the `p` and `h` bits are set, the segment just loaded is also added to the list of resident segments, thus making it resident implicitly. This function of the `h` bit was introduced in AmigaDOS 39, then removed due to restricted ROM size in AmigaDOS 40, and was reintroduced in AmigaDOS 45.

If the `e` protection bit is *not* set, but the shell variable `VIEWER` is set, then the Shell attempts to check whether the system `datatypes.library` is able to identify the contents of the file. If a suitable datatype exists, the Shell prepends the contents of this variable to the file name and then restarts command location. By setting `VIEWER` to the system program `MultiView`, the Shell can therefore display any system-known datatype by typing its name. This feature was added in AmigaDOS version 45.

If the above algorithm was able to come up with a segment, the Shell establishes its input, output and error stream, and potentially also updates its console process if the error output is interactive. The found segment is then — depending on the context — either started as a background process, or overlays the Shell executable with the `RunCommand()`, see section **??**. A background process is started for the command left of a pipe operator, i.e. to the left of the "`|`" or "`||`" operators, or if the "`&`" operator is present at the right of the command. Otherwise, the command overlays the Shell process.

Finally, if the Shell found a file, but could not load a segment for executing, interpreting or viewing it, it indicates that the file is not executable.

15.1.10 The Execute Command

Unlike what its name suggests, the `Execute` command does not actually interpret Shell scripts. Instead, it performs argument substitution within an existing Shell script through a simple pattern matching process,

and then leaves the execution of the resulting script to the Shell by adjusting its command input stream. Thus, `Execute` does not actually execute anything, it rather prepares a script for execution through the Shell.

Argument substitution through `Execute` is controlled through additional syntax elements that are only implemented within `Execute` and that are unrelated, and even partially conflicting with the syntax elements of the Shell[4].

Argument substitution and `Execute` syntax elements are controlled through "pseudo"-commands that are only interpreted though `Execute` but removed before the resulting script is fed back into the Shell. All these pseudo-command start by default with a dot ("`.`"), though even this character can be changed through another pseudo-command. Such pseudo-commands controlling the `Execute` syntax elements shall be placed at the top of scripts as the `Execute` one-pass interpreter needs to see them first. If the first line of the script does not contain a pseudo-command, `Execute` bypasses argument substitution completely and feeds the script unaltered into the Shell by adjusting its command input stream to the script.

The following pseudo-commands are supported by the `Execute`:

`.dot` takes a single argument and changes the character by which all (following) pseudo-command start. This is by default a dot ("`.`"), and for the sake of simplicity, the following pseudo-commands are all shown with this default.

`.key` or `.k` defines which arguments a Shell script takes. The argument is a `ReadArgs()` type template, defining the names of the formal parameters along with their types. Each formal name in this template, enclosed in angle brackets – or rather the characters defined by `.bra` and `.ket` – is substituted by a matching argument provided to the script. Clearly, only a single `.key` pseudo-command shall be present within a script. Such command templates and their syntax are discussed in section **??** along with the `ReadArgs()` function which is also used by `Execute`.

`.default` or `.def` defines defaults for arguments that are not present on the command line that invoked the script. It takes two arguments, first the key — the formal name of the parameter for which a default is to be provided — and the default value itself. Key and default value may be either separated by blank spaces, tabulators, or an equals-sign "(=)". If no default value is provided, an empty string is used. A single `.default` pseudo-command can be provided per formal parameter. Another mechanism to provide default parameters is through the "`$`" character, see below.

`.bra` defines the character that marks the beginning of a formal parameter that is to be substituted. A logical choice for such formal parameters would be to place them in local variables, and let the Shell perform the substitution[5]. However, `Execute` uses another syntax by which formal parameters are enclosed in pairs of characters, one starting the parameter, and another ending it. The initial character is by default the "<" sign, but the `.bra` pseudo-command can change it. As "<" also redirects the standard input of commands, this default is probably not a very wise choice, and it should be changed by `.bra`. Suggested alternatives are curly or square brackets.

`.ket` or `.k` defines the character that marks the end of a formal parameter that is to be substituted. This is by default the ">" sign, i.e. formal parameters are enclosed in pairs of angle brackets. Unfortunately, this default is not a very wise choice either as it makes output redirections in scripts impossible. To override this default, `.ket` should be used. A suggested alternative is a closing curly or square bracket.

`.dollar` or `.dol` defines the character that defines an alternative mechanism for providing defaults for formal parameters. Without this pseudo-command, the character for providing defaults is the dollar sign ("`$`"). Thus, for example, with all defaults active, the script

```
.key FILE

echo <FILE$Help>
```

[4]This is probably another historical accident from Tripos legacy

[5]This is another historical accident, likely.

echos the contents of the format parameter "FILE" defined through the .key command, though if not present, prints "Help".

Thus, an optional "$" sign, or its replacement defined through this pseudo-command, separates the formal parameter from its default value during substitution. The formal name, the "$" sign, and its default are all enclosed in this order in the angle brackets, as seen in the example above, or in whatever the angle brackets were replaced with by the .bra and .ket commands.

Two "$" signs in angle brackets expand to the shell number and may be used as unique identifier, e.g. to generate a file name of a temporary file. The following script

```
.dot . ;ensure that argument substitution is on
echo "The shell number is <$$>"
```

prints the current shell number on the console. The topmost pseudo-command of this script is only required to ensure that Execute performs argument substitution — without it, the "<$$>" token would be printed literally.

Even though the "$" sign as syntax element for default value separation is only active within .bra and .ket, it is still a bad choice as it conflicts with the Shell syntax which uses the same character to indicate Shell variables.

Formal argument substitution otherwise follows the same lines as variable substitution, and Execute attempts to preserve the original command line arguments as good as possible. That is, if a formal parameter is enclosed within double quotes in the script, asterisks and double quotes are escaped properly. If the formal parameter included spaces or tabulators and thus was quoted on the command line, Execute generates a pair of double quotes when substituting the formal parameter with its value, unless the formal parameter is already enclosed in double quotes. If a formal parameter takes multiple arguments, as indicated by a /M modifier in its template (see section **??**), then Execute also expands it as multiple parameter in scripts.

The generated script after substitution of formal parameters is placed in the T: directory, its name is filled into cli_CommandFile and the command input of the invoking Shell, namely cli_CurrentInput (see section **??**), is redirected to a file handle opened from this temporary script. Thus, the Shell continues execution of commands, but rather takes its input from the temporary file rather than its current input, e.g. the console. As this construction would forbid the recursive execution of another Shell script within a Shell script, the Shell detects Execute as a special case for which it keeps a stack of script files and active redirections. Through that, it provides each recursive Execute a clean environment that allows redirection of the command input through another nested instance of Execute.

This is a major difference compared to AmigaDOS version 40 and below which instead resolved this situation by concatenating recursively executed scripts to each other. This construction had a series of bad side effects, one being that a script could not skip backwards over an Execute command.

Once cli_CurrentInput exhausted, the Shell terminates execution there, closes this file handle, deletes the temporary file whose name is stored in cli_CommandFile and, depending on how it was initiated and configured, continues execution from cli_StandardInput or terminates. More on Shell processing of scripts and its configuration through System() is found in sections **??** and **??**.

15.2 Creating and Controlling the Shell

The functions in this section create shell processes and run commands within them, or overlay a shell process with a command. AmigaDOS also keeps a process table of all active shell processes and each running shell is uniquely identified by its task number.

15.2.1 Create New Shells and Execute Scripts

The `System()` function creates a new shell, and potentially executes a shell script within it. Depending on parameters, it waits for the script to complete, or launches an interactive Shell in a console or a non-interactive shell in the background. The same *dos.library* entry point exists under three names that only differ in how the code generator of a compiler provides parameters for it.

The `SystemTagList()` function is equivalent to the `System()` function and does not differ in arguments and call syntax. It is only present to harmonize function naming across all library entry points. It takes a pointer to a string containing a shell script, and additional parameters encoded in a `TagItem` array as defined in `utility/tagitem.h`.

The `SystemTags()` function also receives this string, but expects the `TagItem`s explicitly as one or multiple extra arguments to the function. Compilers typically build then the `TagItem` array on the stack and pass the pointer to the first item on the stack to the *dos.library* entry point.

```
error = SystemTagList(command, tags) /* since V36 */
D0                    D1      D2

LONG SystemTagList(STRPTR, struct TagItem *)

error = System(command, tags) /* since V36 */
D0             D1      D2

LONG System(STRPTR, struct TagItem *)

error = SystemTags(command, Tag1, ...) /* since V36 */

LONG SystemTags(STRPTR, ULONG, ...)
```

This function is the generic shell execution function that creates a shell in one or multiple modes, and executes the commands provided as first argument in that shell. Hence, the first argument establishes a shell script, encoded as a string, that is interpreted by the shell. If the string depletes, the shell terminates without continuing to pull commands from its input stream; this is in contrast to the `Execute()` function in section **??**. This string may contain multiple (compound or simple) commands separated by newline characters, they will be executed one after another. Argument substitutions as by the `Execute` command (see section **??**) are *not* performed, but otherwise the entire shell syntax is available, including shell variables, compound commands, back-tick substitution, redirection and pipes. If argument substitution is necessary, `Execute` should be called as command in the first argument, receiving its input from a temporary file or a pipe.

Depending on its arguments, this function is synchronous, i.e. waits for the completion of the called commands, or asynchronous and then detaches from the caller. By proper usage of arguments, this function can emulate (or is actually even used by) the `Run` and `NewShell` commands, and is also used by the system startup module to create the initial boot shell, see section **??**.

By default, the newly created Shell receives the input and output file handles of the caller, i.e. `pr_CIS` and `pr_COS` are copied from the calling process, see chapter **??** for the documentation of the process structure. However, with suitable tags, the caller can provide alternative input and output streams. Whether these file handles are closed upon termination of the shell depends on further arguments, but the default is not to close them when run synchronously, and to close them otherwise, even if they were the streams of the caller, so beware!

The input and output file handles provided to the shell *shall be different*, i.e. it is not permissible to provide the same file handle as input and as output stream. If the input and output handle should go to an interactive stream such as the console, then only provide an input stream and set the output stream to `ZERO`

by SYS_Output, see the list of tags below. The *dos.library* will in that case create an output file handle by opening another stream from the input handle to the console through the "*" file name. This is similar to the CloneHandle() function introduced in section **??**.

The newly creates shell will receive a copy of the path, the local shell variables, the prompt, the current directory and the stack size of the shell of the caller, if such a shell exists, i.e. pr_CLI of the calling process is non-ZERO. Otherwise, a default path containing only the C: directory and the current directory will be created, no local variables will be defined, the prompt will be set to "%N> " (including a trailing space) and the current directory to "SYS:".

By default, the executing shell will be the system Shell whose segment is available as a system segment of the name "BootShell" in the list of resident modules, see section **??**. Other shells can be provided by specifying their names which are then located on the list of resident segments. Section **??** provides more information on how to implement a custom shell.

The tags this function tags are documented in dos/dostags.h and consist of the SYS_ tags and a subset of the NP_ tags also used to create new processes, defined in section **??**.

SYS_Input This tag takes a BPTR to a file handle as argument which becomes the input stream of the new shell. If this tag is not provided, the input stream of the calling process (pr_CIS) will be used. If command is NULL, this stream will also be closed when the shell terminates, thus the caller needs to open a separate file handle as if SYS_Asynch is set.

SYS_InName This tag takes a string as argument. This string will be used as argument to Open() to create a stream that will be used as input stream to the newly created shell. This stream will always be closed when the created shell terminates, regardless of other tags. This tag is mutually exclusive to SYS_Input, it was introduced in AmigaDOS version 47.

SYS_Output This tag defines a BPTR to a file handle which will be used as output stream of the new shell. This handle *shall be different* from the handle provided by SYS_Input. If this tag is not present, the output stream of the calling process (pr_COS) will be used for shell output. If the command is NULL and SYS_Asynch is not set, then SYS_Output or pr_COS is cloned by the mechanism of section **??** if it is an interactive stream, otherwise the console of the caller is used as an output stream. That is, even though System() is in this case implicitly asynchronous, SYS_Output will *not* be closed and remains available to the caller. If SYS_Output is explicitly set to ZERO, then AmigaDOS will attempt to create an output stream itself: If an input stream is present and interactive, and command is non-NULL or SYS_Asynch is non-zero, then the input stream is cloned through the "*" file name, see **??**. Otherwise, the handler provided through the NP_ConsoleTask or the console task of the caller, if the former tag is not present, is used to open "*". If opening the console fails, AmigaDOS will instead provide a handle to NIL: as output stream, thus disregarding any output. Any stream implicitly provided by the *dos.library* by the above mechanism rather than explicitly through a non-ZERO argument to SYS_Output will also be closed transparently upon termination of the shell.

SYS_OutName provides an output file name that will be opened by the *dos.library* and used as output stream for the newly created shell. This file will always be closed when the shell terminates. This tag is mutually exclusive to SYS_Output, it was introduced in AmigaDOS version 47.

SYS_CmdStream provides a BPTR to a file handle from which commands are read, i.e. a shell script is supplied as stream, and not as a string. This tag is only used if the command argument is NULL and then provides an alternative (stream-based) source for the script to be executed. This stream is *always* closed on exit, and closure cannot be prevented by any other tag. If SYS_Asynch is set, then the shell first reads commands from SYS_CmdStream, and once this stream reaches an EOF, it is closed and the shell continues reading from SYS_Input until this stream also reaches an EOF, or an EndCLI command terminates the Shell. Thus, the configuration of providing a SYS_CmdStream and setting SYS_Asynch is equivalent to the NewShell command. This tag was added in AmigaDOS version 47.

SYS_CmdName provides a file name of a shell script whose contents is interpreted. This tag is mutually exclusive to SYS_CmdStream and only used if the command argument is NULL. This stream will always be closed on exit. This tag was added in AmigaDOS version 47.

SYS_Asynch If the Boolean argument to this tag is non-zero, the shell is detached from the calling process and executes concurrently to the caller; setting this tag also implies that the streams provided by SYS_Input and SYS_Output are closed when the shell terminates. It is thus necessary to explicitly provide SYS_Input or SYS_InName and SYS_Output or SYS_OutName, as otherwise the input or output streams of the calling process will be closed. For legacy reasons, setting the command argument to NULL also enforces asynchronous execution, independent of the value of this tag, and thus the same precautions in preparing input and output streams should be taken[6] for a NULL command as well.

SYS_UserShell If this Boolean tag is set to non-zero, then the "user shell" is launched. This corresponds to the segment "Shell" on the resident list of AmigaDOS. The default of this tag is DOSFALSE, indicating that the Boot Shell is to be used which corresponds to the "BootShell" segment on the resident list. Upon system startup, and in all typical configurations of AmigaDOS, both correspond to the AmigaDOS Shell. Users can, however, replace both shells with custom segments, see section **??**.

SYS_CustomShell This tag provides a string of the name of a custom shell to be used instead. The AmigaDOS list of resident segments is scanned for a fitting name, and the resident segment found is then used as shell.

The following tags defined for CreateNewProc(), see section **??**, are also recognized:

NP_StackSize defines the stack size in bytes for the shell to be created and also the stack size the shell will allocate for its clients. The default is 4096 bytes for the shell itself; for the shell clients, the stack size is taken by default from the shell of the calling process, or 4096 bytes if the caller is not a shell process.

NP_Name is the name of the shell process to be created. The default is "Background CLI". AmigaDOS version 45 and before did not honor this tag.

NP_Priority is the priority of the shell process to be created. The default priority is the priority of the calling process.

NP_ConsoleTask provides a pointer to the MsgPort of the console handler that is used for opening the "*" and paths relative to "CONSOLE:". This port will be copied to pr_ConsoleTask, but is potentially overridden by the process MsgPort of the input or output file handle if they are interactive. If command is NULL, then an interactive output handle will provide the console. Otherwise, an interactive input handle will override NP_ConsoleTask, and will also request with ACTION_CHANGE_SIGNAL to deliver break signals to the created shell.

NP_CopyVars is a Boolean tag that specifies whether local shell variables are copied into the new shell. The default is to copy the variables.

NP_Path contains BPTR to a linked list of directory locks that establish the path of the newly created shell. The structure of this path is specified in section **??**. The path provided by this tag *is not* copied for the new shell, but directly used *and released* by it when it exits. If this tag is not present, a copy of the path of the caller is made if the caller is associated to a shell. Otherwise, a minimal path will be created that contains only the current directory and the C assign.

NP_ExitCode and NP_ExitData define a function that is called when the shell process exits. This mechanism is described in more detail in section **??**.

The return code of the System() function is -1 in case creating the shell failed. In such a case IoErr() delivers an error code providing more details on the cause of the failure. Otherwise, if the commands were executed synchronously, the return code is the result code of the last command in the shell executed, and IoErr() is set to the error code set by the last command executed. If an asynchronous shell was created, the result code is 0 on success, and IoErr() will be set to 0.

System() is used to implement a couple of essential system functionalities, and all Shell commands and system functions that create a shell go through this function. This includes the NewShell and Run shell

[6]This inconsistency does not allow synchronous execution in combination with a SYS_CmdStream; this is probably a defect.

commands and the Shell icon on the Workbench. The initial CLI interpreting S:Startup-Sequence is also created through the System() function from AmigaDOS version 47 onward.

The Run command creates a new shell process by the following:

```
SystemTags(NULL,SYS_Input,instream,SYS_Output,Output(),
           SYS_UserShell,TRUE,TAG_DONE);
```

where the input stream instream is a string stream (see section **??**) containing the commands to be executed; they are taken from the command line of Run, or to be more precise, from the input buffer of its input file handle.

Even though SYS_Asynch is not set, the System() function detaches the created shell because its first argument is NULL, see above. Surprisingly, it is the System() call and not the Run command that prints the CLI number of the created shell on the console in this particular case.

The NewShell command and also the Shell program on the Workbench uses System() as follows:

```
SystemTags(NULL,SYS_InName,window_arg,
           SYS_CmdStream,Open(from_arg,MODE_OLDFILE),
           SYS_Output,ZERO,
           SYS_Asynch,TRUE,SYS_UserShell,TRUE,
           NP_Name,"Shell Process",TAG_DONE);
```

where the window argument and the from argument are coming from the command line arguments of the NewShell command. That is, System() receives a console as new input file handle, and a command stream which is by default "S:Shell-Startup". Setting SYS_Asynch to TRUE ensures that the shell continues to read commands from its input file handle as soon as the command file depletes. AmigaDOS version 45 and below used a custom mechanism for NewShell that was undocumented and is now deprecated. The Boot Shell is created likewise except that NP_Name is set to "Initial CLI".

The Execute() function described in section **??** is approximately equivalent to

```
SystemTags(NULL,SYS_CmdStream,cmd,
           SYS_Input,in,SYS_Output,out,SYS_UserShell,FALSE,
           NP_Priority,0,TAG_DONE)?DOSTRUE:DOSFALSE;
```

where cmd is a string stream (see section **??**) containing the commands to be executed, constructed from the first argument of Execute(), and in and out are its second and third argument. The difference of System() and Execute() is the returned result, and that the above function returns immediately while Execute() with a non-NULL command string blocks until the Shell terminates.

Under AmigaDOS version 45 and before, this function closed the stream provided by SYS_Input in case the new shell could not be created, even if synchronous execution is requested. This was fixed in AmigaDOS version 47.

> *Your Resources, My Resources* Beware how the System() function manages resources you pass in. In general, for synchronous operations, resources such as streams are used, but not released. For asynchronous operations, the streams you provide are in the responsibility of the created shell and released when the shell terminates, even if they were the resources of your process. The System() function also creates an asynchronous shell if the command argument is NULL, regardless of the SYS_Asynch tag. For details, please study this section carefully.

15.2.2 Execute Shell Scripts (Legacy)

The `Execute()` function creates a new AmigaDOS shell, which then executes commands from a string, and once the string depletes, continues executing commands from an input stream. This function is obsolete, and should be replaced by `System()` which is more flexible.

```
success = Execute( commandString, input, output )
D0                 D1             D2      D3

BOOL Execute(STRPTR, BPTR, BPTR)
```

This function is a deprecated function to create AmigaDOS shells and interpret shell scripts that has been superseded by `System()`, see section **??**.

The first argument is a string containing a shell script with lines separated by line feed characters. If this argument is `NULL` and thus no command string is present, a new Shell is created and run asynchronously in the background, i.e. `Execute()` returns then immediately. The new Shell receives its input, and thus its commands from the file handle provided as second argument. This stream is closed when the background process terminates. The output goes to a clone of the output stream `output`, obtained through an algorithm similar to the one in section **??** if it is interactive. Otherwise, the console of the caller is used as output stream, and if that stream does not exist, output will be disregarded by providing a `NIL:` handle to the shell created. In particular, `output` will *not* be closed, but remains available to the caller as only a clone of this handle will form the output of the Shell.

If the first argument is non-`NULL`, the `Execute()` command is synchronous and will not return until the Shell completed its job. It first reads commands from the the command string, and once this depletes, checks whether the `input` handle is `ZERO`. If so, the Shell terminates at this point. Otherwise, it switches over reading commands from the stream provided by `input` until this stream reaches its EOF, or an `EndCLI` command terminates the Shell explicitly. The output of the Shell goes through the `output` stream provided as third argument. If the third argument is `ZERO`, and the input stream is interactive, AmigaDOS opens "`*`" from the input stream handler, otherwise attempts to open the console from the console process of the caller, or if that is not possible, disregards all output by providing a `NIL:` handle as output. In this mode, neither `input` nor `output` will be closed when the function returns.

Unlike `System()`, this function returns `DOSTRUE` on success and `DOSFALSE` in case creating the Shell failed. This implies that the result code of the last command executed is not available to the caller — instead, if the Shell could be launched, `DOSTRUE` is returned as success indicator. It also always executes commands through the Boot Shell, and not through a custom or user shell.

Except for the similarity in name, the `Execute()` function has nothing in common with the `Execute` command described in section **??**. One does not depend on the other, i.e. `C:Execute` is not necessary for `Execute()`, and `Execute` does not run into `Execute()` either, but rather modifies the current input stream of the Shell. Under AmigaDOS version 34 and below, `Execute()` was based on the `Run` command, but this dependency has been removed long since. Since AmigaDOS 36, `Run` depends on the `System()` function.

Under AmigaDOS version 36 and up to version 45, this function closed the `input` stream in case it could not create a new Shell.

15.2.3 Run a Command Overloading the Calling Process

The `RunCommand()` runs a shell command from a process, and overloads the process with the command.

```
rc = RunCommand(seglist, stacksize, argptr, argsize) /* since V36 */
D0                D1         D2       D3      D4

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

This function runs a command from the calling process, and provides for this command its own stack and its own arguments. It does not create a new process nor a new shell, but executes the command as part of the caller. The AmigaDOS Shell uses this function to execute loaded commands within its context.

The `seglist` argument is a BPTR to the chained list of segments of the executable to start, for example as returned by `LoadSeg()` for disk-based commands, or as the `seg_Seg` element of resident commands, see section **??**.

The `stacksize` argument is the size of the stack in bytes to be provided to the command; this has to be provided by the caller as this function does not attempt to identify the minimum stack size necessary. The Shell takes the stack size from `cli_DefaultStack`, see section **??**, and multiplies it by 4. It additionally searches the `seglist` for the stack cookie, see section **??**, and from that potentially increases the size. The `Task` structure of the caller, in particular its elements `tc_Upper` and `tc_Lower` are also updated to reflect the new position and size of the stack. Thus, programs can depend upon that their stack size is given by `task->tc_Upper - task->tc_Lower`, regardless whether they were started from the shell or the Workbench.

The `argptr` and `argsize` arguments provide command line arguments that are passed to the command. They are provided through several mechanisms: First, the CPU register `a0` is loaded with `argptr` and register `d0` is filled with `argsize`. Second, the `pr_Arguments` element of the process is temporarily replaced by `argptr`, see also chapter **??**, and thus the arguments are made available to `GetArgStr()`. Third, a buffer for the input file handle `pr_CIS` of the caller is allocated and filled with a copy of the argument line; more on the file handle structure is in section **??**. Thus, buffered read operations as those listed in section **??** will retrieve the arguments. This step is necessary to make the arguments available to `ReadArgs()`, see section **??**, the AmigaDOS argument parser.

The `ReadArgs()` function, and probably many other parsers require that the argument is terminated by a newline, hex `0x0a`. While it is possible to provide an argument string containing more than one newline character, `ReadArgs()` will ignore all arguments behind the first newline. However, some commands such as `Run` will make use of the entire string.

All changes performed by `RunCommand()` on the caller and its resources, namely the stack, the modifications of the input file handle and storage of the arguments and the modified stack are reverted when the called command returns.

As the program name *is not* part of the command line arguments, it is advisable to set the path and file name of the loaded program by `SetProgramName()` upfront, more on this function in section **??**. The startup code of many C compilers will construct `argv[0]` from it.

This function returns the result code of the called command, or -1 if it failed to allocate resources such as the input file handle buffer or the stack. `IoErr()` remains its value from the called command, or is set to `ERROR_NO_FREE_STORE` if resources could not be allocated.

15.2.4 Checking for Signals

The `CheckSignal()` function tests whether particular signals, as for example those to break process or a shell script, are set in the process of the caller.

```
signals = CheckSignal(mask) /* since V36 */
D0                    D1

ULONG CheckSignal(ULONG)
```

This function tests those signals that are set in the `mask` and returns their state. Typical signals to check for are defined in `dos/dos.h` and include those bits that are set by the `Ctrl-C` through `Ctrl-F` console key combinations. The function returns all signals set in the process *and* the mask, and clears the signals in the mask. This is unlike the exec function `SetSignal()` which returns the complete signal mask of the process, and not only those requested in the argument. A typical example is

```
if (CheckSignal(SIGBREAKF_CTRL_C))
  break; /* abort the command */
```

which checks whether the user pressed Ctrl-C, and also clears the corresponding signal. This function does not alter IoErr().

15.2.5 Request a Function of the Shell

The DoShellMethod function requests from the shell of the calling process a specific function. The DoShellMethodTagList() function belongs to the same entry point of the *dos.library*, though uses a different calling convention that receives the tag list as an explicit pointer instead of a variably sized argument list.

```
ptr = DoShellMethodTagList(method, tags) /* since V47 */
D0                          D0     A0

APTR DoShellMethodTagList(ULONG, struct TagItem *)

ptr = DoShellMethod(method, Tag1, ...)

APTR DoShellMethod(ULONG, ULONG, ...)
```

This function requests the from the shell of the calling process the execution of a function identified by method. The arguments to the shell are provided in the tag list, or by a list of tags terminated by TAG_DONE. The return value of the shell is provided in ptr, a secondary return code is provided in IoErr().

The AmigaDOS Shell supports the following methods defined in dos/shell.h:

SHELL_METH_METHODS returns a const array of methods the shell supports. This is an array of ULONGs, each representing a method ID. The array is is terminated by a 0UL. Every shell shall support this method.

SHELL_METH_GETHIST is a method of the AmigaDOS Shell that provides read-only access to the history. It does not take any arguments. The returned pointer is a MinList structure, see exec/lists.h. Each node in this list consists of a HistoryNode structure, defined in dos/shell.c:

```
struct HistoryNode {
  struct MinNode  hn_Node;
  UBYTE          *hn_Line;
};
```

The hn_Line is a pointer to a command in the history of the Shell. This command is *not* terminated by a newline, but only by a NUL. The caller *shall not* alter this list.

SHELL_METH_CLRHIST erases the entire history of the AmigaDOS Shell. This method does not take any arguments.

SHELL_METH_ADDHIST adds an entry to the tail of the AmigaDOS Shell history and make it accessible to the user. This method takes a single tag, namely SHELL_ADDH_LINE. The argument of this tag is a const UBYTE * to a NUL-terminated string, and defines the entry to be added to the history. The Shell copies the provided string, and can strip initial or trailing spaces. It can also limit the size of the history buffer by releasing the oldest entry or entries from the head of the list. This method returns a non-zero result on success, or NULL on error.

SHELL_METH_FGETS retrieves a single line from the console the Shell runs in, while offering access to TAB expansion and access to the history. The difference between this method and the FGets() function is that while the latter provides elementary line editing functions of the console, it does not have access to Shell internal states such as the path of the caller required for TAB expansion[7]. If this Shell method is used,

[7]Readers beware: TAB expansion and the history are *not* console features, but Shell features.

TAB expansion is available based on the path and configuration of the Shell.

This method takes a single optional Boolean tag, SHELL_FGETS_FULL, defined in dos/shell.h. If this tag is DOSFALSE, which is the default, then the Shell is instructed to read a string from the console that corresponds to a path relative to the current directory of the caller. TAB expansion will only scan the current directory for matches. This mode is useful for requesting a file or multiple files from the user, and it is for example used by the AmigaDOS argument parser, ReadArgs(), when the input is a question mark ("?") and more input is required from the user.

If SHELL_FGETS_FULL is non-zero, then a full command line is requested from the Shell. This implies that the first argument of the input requested from the console is a command, and is thus located somewhere in the path of the caller. Thus, when performing TAB expansion within the first argument, the entire path is scanned, and not just the current directory.

This method returns a pointer to a NUL-terminated string as result; as this string is kept within Shell internal buffers, the caller shall make a copy of the string before calling another method of the Shell.

If DoShellMethod() is called from a process that is not part of a shell, it returns NULL and sets IoErr() to ERROR_OBJECT_WRONG_TYPE.

15.2.6 Find a Shell Process by Task Number

The FindCliProc() function finds a process by its task number.

```
proc = FindCliProc(num) /* since V36 */
D0                 D1

struct Process *FindCliProc(ULONG)
```

This function returns a process given its task number. Note, however, that AmigaDOS only assigns task numbers to shells and commands run within or from shells and not to other processes, such as those started from the Workbench, handlers, file systems or device drivers. Thus, this function is not quite as useful as it seems.

The num is the task number that identifies the process to locate. It is stored in the pr_TaskNum element of the process structure, see chapter **??**.

This function returns a pointer to the process structure whose task number is num. If no such process exists, this function returns NULL. However, even if it returns a non-NULL result code, it is possible that the process has already exited and no longer exists when the function returns. Furthermore, the *dos.library* does not attempt to protect its process table within this function, and there is no semaphore protecting it. Therefore, this function *shall only be called* while task switching is disabled with Forbid() and its return value is only valid as long as task switching is disabled.

This function does not alter IoErr(), even if no process is found.

15.2.7 Retrieve the Size of the Process Table

The MaxCli() function returns the size of the process table.

```
number = MaxCli() /* since V36 */
D0

LONG MaxCli(void)
```

This function returns the number of entries the process table can hold. However, this information is of limited value for various reasons: First, this information does *not* correspond to the number of running

processes, but only describes the number of entries the process table is able to hold. Second, only shells and commands run within or from shells are recorded in the process table, and other processes as those started from the Workbench, or handlers, file systems or device drivers do not enter this table. Third, the size of this table changes dynamically depending on how many shell processes are active, even under the feet of the caller.

As the *dos.library* does not protect the process table from modifications, this function *shall only be called* while task switching is disabled with `Forbid()`, and its return value is only valid as long as task switching remains disabled.

15.3 The CLI Structure

The `CommandLineInterface` structure is the public interface of a shell, see section **??** for its definition. Every command started from a shell has access to this structure through the `pr_CLI` BPTR in its process structure, the latter is documented in chapter **??**. As commands run by the shell only overlay the shell process by the `RunCommand()` function specified in section **??**, the process structure of a shell command is actually the process of the shell.

The functions listed in this section provide accessor functions to the `CommandLineInterface` structure. Altering or requesting properties of the shell through these functions should be preferred to modifying the CLI structure directly.

15.3.1 Obtaining the Name of the Current Directory

The `GetCurrentDirName()` function copies the current directory of the shell into the provided buffer if such a shell exists, or retrieves the current directory of the calling process otherwise.

```
success = GetCurrentDirName(buf, len) /* since V36 */
D0                          D1   D2

BOOL GetCurrentDirName(STRPTR, LONG)
```

This function checks whether the caller is a shell command. If so, it copies the string the shell prints through the `%S` token of the prompt as current directory into the supplied buffer, see also section **??**. That is, the function copies `cli_SetName` of the CLI structure (see section **??**), into `buf`. If the current directory path fits into `len` bytes including a terminating `NUL` byte, then this function returns `DOSTRUE` and sets `IoErr()` to 0. Otherwise, it truncates the name, sets `IoErr()` to `ERROR_LINE_TOO_LONG`, but still returns `DOSTRUE`. The Shell does not update `cli_SetName` itself, i.e. the element this function depends upon. Rather, the `CD`, `SwapCD`, `PushCD` and `PopCD` commands keep it consistent when changing the current directory. However, if a command changes `pr_CurrentDir` without updating `cli_SetName`, or calls `SetCurrentDirName()` without updating the current directory of its processes, then the string supplied by this function may not correspond to the current directory the shell actually uses.

If the caller is not a shell command, then the function uses the lock representing the current directory of the calling process, namely `pr_CurrentDir`, and converts it to a string by `NameFromLock()`. This function, see section **??**, also truncates its result to `len` bytes and, if truncation was performed, sets `IoErr()` to `ERROR_LINE_TOO_LONG` and returns `DOSFALSE` if the directory name does not fit into `len` bytes. On success, the function returns a non-zero result code, but it does not set `IoErr()` consistently.

This function had a defect in AmigaDOS version 36 by not handling zero-sized buffers correctly. This was fixed in version 37. Even the latest versions return `DOSTRUE` if the caller is run from a shell, though the supplied buffer is too short and the directory name was truncated.

15.3.2 Set the Current Directory Name

The SetCurrentDirName() updates the buffer within which the shell keeps the string representing the current directory. It does not update the lock representing the current directory.

```
success = SetCurrentDirName(name) /* since V36 */
D0                          D1

BOOL SetCurrentDirName(STRPTR)
```

This function updates the string printed as current directory by the %S token of the shell prompt, see also section **??**; this string represents the path of current directory the shell assumes to operate in. If the caller is executing from a shell, it copies the supplied string into the cli_SetName element of the CLI structure of the shell, and potentially truncates it to the size of this buffer. Even if the string is truncated, the function returns DOSTRUE. It does not update IoErr() in either case. For legacy reasons, the size of this buffer is limited, and thus the buffer may not reflect the full directory name supplied. While in principle the size of the cli_SetName buffer can be changed when creating a CLI structure with AllocDosObject() introduced in section **??**, AmigaDOS ignores the buffer size supplied there.

If the caller is not a shell command, this function returns DOSFALSE without setting an error code.

This function does not attempt to synchronize the lock pr_CurrentDir of the calling process with the supplied path. If the two are not consistent, the path the shell could print as part of the prompt would be incorrect. Thus, any attempt to change cli_SetName through this function *shall* also update the current directory by calling CurrentDir().

15.3.3 Obtaining the Current Program Name

The GetProgramName() function copies the name of the currently executed program into a buffer.

```
success = GetProgramName(buf, len) /* since V36 */
D0                       D1   D2

BOOL GetProgramName(STRPTR, LONG)
```

This function fills buf with the what the shell assumes to be the file name of the currently executed command. Unlike what the function name suggests, the program name can also be a path including directory components if this is what was entered as command. The file name is taken from cli_CommandName of the CLI structure, see section **??**, where the shell deposits it before executing a program. If the program name including NUL termination requires more than len bytes, this function first truncates it and sets IoErr() to ERROR_LINE_TOO_LONG. If the name including a terminating NUL byte fits into len bytes, the name is copied and the function sets IoErr() to 0. In either case, even if the program name is truncated, the function returns DOSTRUE.

If this function is not called from a shell command, the function installs an empty string in buf if len is at least 1, returns DOSFALSE and sets IoErr() to ERROR_OBJECT_WRONG_TYPE.

The startup code of many C compilers use this function, or the equivalent element of the CLI structure, to fill argv[0], the name of the running program.

The shell buffer that keeps the current command name is unfortunately due to legacy reasons limited in size. Even though the shell uses internally a longer buffer and is thus not limited in the path length of the commands it executes, the ability of the shell to communicate long file names to the caller is restricted, and thus the command name retrieved from this function may not reflect the correct file name.

15.3.4 Set the Current Program Name

The SetProgramName() sets the assumed name of the currently executed program.

```
success = SetProgramName(name) /* since V36 */
D0                         D1

BOOL SetProgramName(STRPTR)
```

If this function is called from a shell command, it installs the supplied string as program name into cli_CommandName, see section **??**, and returns DOSTRUE. If the supplied string does not fit into the shell internal buffer, it is truncated without this function indicating failure. Unlike what the function name suggests, the program name is a path and can therefore also contain directory components.

This function is mainly intended to be used by the shell to communicate the name of the executing program to the program startup code, for example to supply argv[0] of C programs.

If this function is not called from a shell command, it returns DOSFALSE. This function does not change IoErr() in any case.

15.3.5 Obtaining the Shell Prompt

The GetPrompt() function copies the prompt format string with all formatting instructions into a caller supplied buffer.

```
success = GetPrompt(buf, len) /* since V36 */
D0                  D1   D2

BOOL GetPrompt(STRPTR, LONG)
```

If this function is called from a shell command, it copies the prompt format string including a terminating NUL into buf if it fits into len bytes, potentially truncating it if it does not. If truncation was performed, IoErr() is set to ERROR_LINE_TOO_LONG, otherwise to 0. In either case, even if the string was truncated, the function returns DOSTRUE.

If this function is not called from a shell command, an empty string is copied into buf if len is at least 1, and IoErr() is set to ERROR_OBJECT_WRONG_TYPE and the function returns DOSFALSE.

The shell prompt provided by this function is the un-expanded prompt including format strings as it is provided by the Prompt command, and not the expanded prompt currently printed by the shell. The AmigaDOS Shell recognizes the following strings:

- %S is replaced by the path of the current directory as returned by GetCurrentDirName().
- %N is substituted with the CLI number, which is the closest analog of a process ID AmigaDOS has to offer. This is taken from pr_TaskNum of the process running the shell, see also chapter **??**.
- %R represents the return code of the last executed command as contained in cli_ReturnCode.
- %% is the percent ("%") sign itself.

The AmigaDOS Shell also expands variables as described in section **??** in the prompt, executes commands in backticks, see section **??**, and injects its output into the printed prompt. Any "%" sign included in expanded variables or backticks is *not* a formatting command but stands for itself.

15.3.6 Setting the Shell Prompt

The `SetPrompt()` sets the shell prompt format string.

```
success = SetPrompt(name) /* since V36 */
D0                  D1

BOOL SetPrompt(STRPTR)
```

If called from a shell command, this function updates the shell prompt format string to `name`, potentially truncating it to the size of the shell internal buffer. It returns `DOSTRUE` even if the prompt is truncated.

If this function is not called from a shell command, it returns `DOSFALSE`. In does not change `IoErr()` in any case.

The shell prompt provided to this function may contain all format strings described in **??**, such as `%S` for the current directory, or shell variables and back-ticks to construct a prompt dynamically. The shell-internal buffer size for the prompt is unfortunately limited. If longer prompts are required, they could be placed in a local shell variable which is expanded when the prompt is printed on the console.

15.3.7 Retrieving the CLI Structure

The `Cli()` function returns a pointer to the `CommandLineInterface` structure describing properties of the shell within which the calling process is executing, or `NULL` in case the process is not run from a shell.

```
cli_ptr = Cli() /* since V36 */
D0

struct CommandLineInterface *Cli(void)
```

This function returns a pointer to the `CommandLineInterface` structure that describes properties of the shell the calling process runs in. The function returns `NULL` if the caller is not part of a shell process. This structure, defined in `dos/dos.h`, looks as follows:

```
struct CommandLineInterface {
    LONG   cli_Result2;
    BSTR   cli_SetName;
    BPTR   cli_CommandDir;
    LONG   cli_ReturnCode;
    BSTR   cli_CommandName;
    LONG   cli_FailLevel;
    BSTR   cli_Prompt;
    BPTR   cli_StandardInput;
    BPTR   cli_CurrentInput;
    BSTR   cli_CommandFile;
    LONG   cli_Interactive;
    LONG   cli_Background;
    BPTR   cli_CurrentOutput;
    LONG   cli_DefaultStack;
    BPTR   cli_StandardOutput;
    BPTR   cli_Module;
};
```

Even though this function was introduced in AmigaDOS 36, a BPTR to the CommandLineInterface structure is also found in the pr_CLI element of the process structure, see chapter **??**.

Some elements of this structure point to string buffers of unspecified size and should therefore not be altered manually as an attempt to copy an over-sized string into them could overwrite other system structures. Instead, the accessor functions from sections **??** to **??** should be preferred for reading, and also for updating them.

The elements of this structure are as follows:

cli_Result2 is the IoErr() the last executed command of the shell left, or the shell created itself when failing to interpret or execute a command line. This element is for example used by the Why command to print a textual description of the error. The Shell also copies it into the $Result2 shell variable.

cli_SetName is a BPTR to a BSTR containing the path of the current directory. This string is used to generate a shell prompt; the AmigaDOS Shell substitutes the "%S" format directive of the prompt by the string stored here. The CD command and its PushCD, PopCD and SwapCD variants update this element. This element should be accessed through GetCurrentDirName() and SetCurrentDirName(), see sections **??** and **??**.

cli_CommandDir contains a linked list of directories that are scanned for commands and scripts. It is a BPTR to the following (undocumented, but trivial) structure:

```
struct PathComponent {
    BPTR pc_Next;
    BPTR pc_Lock;
};
```

where pc_Next is the BPTR to the next directory in the path or ZERO for the end of the list, and pc_Lock is a lock of the directory that will be scanned for a matching command file.

The current directory is always the first component of the path and not included in the above list. It is thus checked first for matching files, even if cli_CommandDir is ZERO. The C: directory is always the last component of a path and neither explicitly included in the above list.

The Path command is used to print and adjust the path stored in this list.

cli_ReturnCode is the return code of the last executed command, i.e. the value the command left in the d0 CPU register when existing to the shell. The Shell also copies this value to the $RC shell variable.

cli_CommandName is a BPTR to a BSTR containing the file name of the currently executing command. The shell places here the unaltered string provided by the user as command, and therefore the string is either the name of a resident segment, a file name in the path, or can include directory components and then is a path relative to the current directory of the shell. It is typically used by the startup code of C compilers to fill the argv[0] argument. This element should be accessed through GetProgramName() and SetProgramName(), see sections **??** and **??**.

cli_FailLevel contains the threshold at which executed commands will cause an abortion of their containing shell script. The value is deposited here by the FailAt command. If a command exits with return code larger or equal than the cli_FailLevel, this will cause termination of the currently executing script.

cli_Prompt contains a BPTR to BSTR that is used by the shell to generate the command prompt of interactive shells. The formatting directives the AmigaDOS Shell recognizes are listed in section **??**. This element should be accessed by GetPrompt() and SetPrompt(), see section **??** and **??**.

cli_StandardInput is a BPTR to a file handle that is the primary source from which the shell reads command lines and also the default input handle the shell provides to its clients in the absence of input redirection. This file handle typically corresponds to a console window within which the shell is executed. The Run command will deposit here a string stream (see section **??**) containing the command or commands

to run in background. The file handle provided through the SYS_Input or SYS_InName tags of the System() function will be placed here. Once this stream is exhausted, the shell terminates.

cli_CurrentInput is a BPTR to a file handle from which the shell is currently reading command lines. This stream is either coming from the SYS_CmdStream or SYS_CmdName tags of the System() function (see **??**), or a string stream constructed from its first argument. Also, the Execute command, see section **??**, places here the file handle of the original or processed shell script that is to be executed. Once this file is exhausted, the shell will close it. This happens, for example, if a script reaches the EOF or is aborted by the Exit command. Depending on how the shell was created and configured, see section **??**, the shell then sets cli_CurrentInput to cli_StandardInput if the two are different and continues execution from there, or in other configurations, terminates.

cli_CommandFile is a BPTR to the BSTR of a temporary shell script that is currently being executed. The only reason why its path is stored here is to allow the shell to clean up such temporary scripts. Whenever the Execute command requires processing a script for argument substitution, it creates a temporary script in T: whose name is stored in cli_CommandFile. Once its execution completes, the shell ensures that this temporary script is deleted again. For details on how Execute works see section **??**.

cli_Interactive is a Boolean flag that indicates whether the shell is interactive, i.e. requesting data from the console. It is computed and updated by the shell itself. If this Boolean is non-zero and cli_Background is DOSFALSE, a prompt is printed before attempting to read a command. If the shell is executing a script, this element is DOSFALSE and the shell then checks for Ctrl-D to potentially abort a running script.

cli_Background is a Boolean flag that indicates whether the shell runs in background. This flag is set by System() and Execute() for shells that are started asynchronously or are equipped with a non-interactive (non-console) output stream. A synchronous System() call with a non-NULL command argument, however, will always create a foreground shell. If this flag is cleared, and the shell is interactive, it prints prompts for requesting commands from the user. A shell not running in background also prints a message when its input stream reaches the EOF and it terminates[8].

cli_CurrentOutput is currently not used by the shell. It is initialized with the same file handle as cli_StandardOutput.

cli_DefaultStack is the minimum stack in long words the shell allocates for commands before executing them. The Shell also checks commands for stack cookies, see section **??**, that may enlarge the stack further. This element is set by the Stack command. Note that this element is *not* the stack size of the shell process itself, but a lower limit of the stack size of its clients.

cli_StandardOutput is the file handle the shell provides as default output handle to its clients, and to which it prints prompts and other messages. This handle is copied by the System() function from the SYS_Output tag or obtained by opening the file name provided by the SYS_OutName tag. If none of the two are available, System() will derive an suitable output stream from the input stream or the console handler of its caller, see **??**.

cli_Module is a BPTR to the singly linked list of segments of the command currently executed. The shell also uses this BPTR to release loaded, non-resident commands. It is either filled with the chained segment list returned by LoadSeg() (see section **??**), or an element of the list of resident segments, see section **??**. A common technique for implementing load and stay resident commands is to set this BPTR to ZERO to prevent the shell from releasing the memory of the loaded program.

Even though the CommandLineInterface structure is typically constructed by AmigaDOS, e.g. through the System() function, it is sometimes necessary to create it manually. This structure shall always be constructed through AllocDosObject() as it contains some internal elements required by extended shell features, see section **??** for details. If allocated by other means, the AmigaDOS shell will lack some

[8]While [**?**] documents that this flag determines whether the shell terminates when its current input reaches an EOF, this is not correct since AmigaDOS version 36. Instead, the startup flags from table **??**, section **??** do.

of its features. Access to TAB expansion and the shell history is through the DoShellMethod() function specified in section **??**, and not through the CLI structure.

The Cli() function does not change IoErr().

15.4 Accessing Shell Variables

The functions in this section provide access to local and global shell variables, get and set them, or check whether a particular variable is defined.

Local variables and aliases are represented as a LocalVar structure defined in section **??** which is linked into the pr_LocalVars list of the Process structure specified in chapter **??**.

Global variables are files in a directory hierarchy of the ENV assign, which exists typically as an external link in the RAM-Handler. The functions in this section also allow to make changes to global variables permanent such that they survive a reboot. For that, the file representing the variable is updated in ENV: and ENVARC:, which is usually the link target of the former. Section **??** introduces these assigns.

15.4.1 Reading a Shell Variable

The GetVar() function reads the contents of a global or local shell variable or alias and copies its contents to a buffer.

```
len = GetVar( name, buffer, size, flags ) /* since V36 */
D0             D1     D2     D3    D4

LONG GetVar( STRPTR, STRPTR, LONG, ULONG )
```

The name argument is the name of the variable to retrieve; this name may contain one or multiple forward slashes ("/") which structure variables hierarchically similar to directories in file systems. It is not case-sensitive. For global variables, this hierarchy is mapped to directories within the ENV assign. There variables are represented as files that may also be accessed through the file system functions of the *dos.library*. Unless specified otherwise, this function first checks for local shell variables first, but if no matching variable is found, it checks for a global variable of the provided name.

The buffer and size arguments specify a buffer and its length into which the contents of the variable or alias is copied. It is advisable to store only printable characters in variables, in which case a NUL or line feed character truncates the contents of the variable. However, variables may also contain binary data; such binary data is, however, only copied if it is requested explicitly. If a variable does not fit into the buffer, it is truncated without setting an error code, though non-binary variables are always NUL-terminated. The size argument includes the byte necessary for termination.

The flags argument determines the type of variable to access, and whether binary contents of the variable is made accessible. The following flags are defined in dos/var.h:

The lowest 8 bits of flags identify the nature of the variable. If set to LV_VAR, this function reads variables, if set to LV_ALIAS, it reads aliases. Aliases are always local to the shell and cannot be global.

If the GVF_GLOBAL_ONLY flag is set, then this function only returns contents of global variables, and local variables are ignored. Aliases are never global.

If the GVF_LOCAL_ONLY flag is set, then this function only returns contents of local variables and ignores any global variables.

If the GVF_BINARY_VAR flag is set, then copying the contents of variables into the supplied buffer does not stop at newline (0x0a) or NUL characters, but attempts to transfer the entire variable into the buffer, potentially truncating it if necessary.

If the GVF_DONT_NULL_TERM flag is set along with GVF_BINARY_VAR, then this function does not attempt to NUL-terminate the supplied buffer. Rather the entire variable, if possible, is copied into buffer. Otherwise, even binary content is NUL-terminated.

On success, this function returns the size of the supplied variable after truncation is applied if necessary. The size does not include the terminating NUL, if one is included. On success, this function sets IoErr() to the entire size of the variable in the database without truncation, including any bytes beyond a terminating newline or NUL.

On error, −1 is returned and IoErr() is set to an error code; it is set to ERROR_BAD_NUMBER if len is 0, or ERROR_OBJECT_NOT_FOUND if the variable could not be found. Any other error code resulting from reading from ENV: is also forwarded to the caller through IoErr().

AmigaDOS version 36 returned the full size of the variable, not the number of characters that could be read. This was changed in version 37 which also sets IoErr(). The flag GVF_DONT_NULL_TERM only worked for local variables up to version 37. This was fixed in version 39.

15.4.2 Setting a Shell Variable

The SetVar() assigns a value to a local or global shell variable or an alias, potentially creating a new variable, or potentially deleting it.

```
success = SetVar( name, buffer, size, flags ) /* since V36 */
D0                D1     D2     D3    D4

BOOL SetVar(STRPTR, STRPTR, LONG, ULONG )
```

This function assigns the value in buffer to the variable or alias named name, possibly creating it if it does not exist, or deleting it if buffer is NULL.

The name argument is the name of the variable to create, update or delete. It is not case-sensitive. The name may contain one or multiple slashes ("/") which corresponds to a hierarchy of variables that work similar to directories and paths on regular file systems; this hierarchy is represented by directories in ENV: for global variables, but is also available for local variables. This function potentially creates levels in the hierarchy, i.e. sub-directories, if necessary.

The buffer and size arguments are the contents of the variable and the size of the contents. It is generally advisable to only include printable characters in variables, even though they may include also binary data which can be retrieved by setting the GVF_BINARY_VAR for GetVar(). If size is −1, then the buffer contains a NUL-terminated string and this function determines the string size itself. The terminating NUL of such a string *does not* become part of the variable value.

If buffer is NULL, then a matching variable is deleted. It is then equivalent to DeleteVar().

The flags argument determines the type of the variable to be set or created. The following flags are defined in dos/var.h:

LV_VAR sets, creates or deletes a regular shell variable, local or global.

LV_ALIAS sets, creates or deletes an alias. Aliases can only be local. This is mutually exclusive to the above.

If GVF_GLOBAL_ONLY is set, then a global variable is created, deleted or updated. This flag *shall not* be combined with LV_ALIAS. If GVF_GLOBAL_ONLY is *not* set, and buffer is not NULL, this function only updates or creates local variables; it even creates a local variable if a global variable of the same name already exists. If GVF_GLOBAL_ONLY is *not* set, and buffer is NULL, it first attempts to delete a local variable, and if none is found, attempts to delete a global variable of the matching name.

If GVF_LOCAL_ONLY is set, then only a local variable is created, deleted or updated. This flag only makes a difference if buffer is NULL as all other operations default to local variables anyhow.

If GVF_SAVE_VAR is set, and a global variable is set or deleted, the change will be made permanent by mirroring it in the ENVARC: directory[9].

This function returns a non-zero return value in case of success. IoErr() is then not set consistently. On error, this function returns DOSFALSE and IoErr() is set to an error code. If a variable is to be deleted by setting buffer to NULL, and this variable does not exist, this function will set IoErr() to ERROR_OBJECT_NOT_FOUND.

Releases prior to AmigaDOS version 47 did not clear the e protection bit of the file(s) created in ENV: or ENVARC:. If a variable name contained two slashes such as "//", releases prior to AmigaDOS version 47 crashed.

Furthermore, SetVar() depends on the file system responsible to the directory the ENV assign points to to support both MODE_READWRITE as mode of the Open() function specified in section **??**, as well as the SetFileSize() function, see section **??**. While both the FFS and the RAM-Handler implement these functions and ENV usually points to a directory in the latter, some third-party file systems may not. In such a case, SetVar() can fail silently when setting a global variable.

Additionally, SetVar() has up to and including AmigaDOS version 47 a potential race condition if two processes attempt to modify or read the same global variable concurrently. While the file system implementing ENV should provide sufficient protection for writing data concurrently into a file, SetVar() uses a two-step process in which the data is written first, and then the variable is truncated to its target size which may happen concurrently with a second process either writing to, or reading from the same variable.

15.4.3 Finding a Shell Variable

The FindVar() locates a local shell variable or alias of the calling process.

```
var = FindVar( name, type ) /* since V36 */
D0              D1    D2

struct LocalVar * FindVar(STRPTR, ULONG )
```

This function finds the administration structure corresponding to a local shell variable or alias of the calling process. It does not provide access to global variables which are represented as files in the ENV assign.

The name argument specifies the name of the local variable or alias to locate. It is not case-sensitive. This name may contain forward slashes ("/") to structure variables in hierarchies.

The type argument defines whether a local shell variable or alias is to be found. The following types from dos/var.h are available:

LV_VAR requests the function to locate a local shell variable.

LV_ALIAS request the function to find an alias. This is mutually exclusive to the above.

Only the least significant bits of type are relevant, all other bits, in particular GVF_GLOBAL_ONLY, are ignored.

If a matching variable or alias is found, a pointer to a LocalVar structure, defined in dos/var.h is returned:

```
struct LocalVar {
  struct Node lv_Node;
  UWORD       lv_Flags;
  UBYTE      *lv_Value;
  ULONG       lv_Len;
};
```

[9] [**?**] does not document this flag, even though it is present and working.

This structure shall never be allocated nor modified by the caller as it may grow in future releases. Instead, local variables shall only be created through SetVar().

The elements of this structure are as follows:

lv_Node is a node structure as defined in exec/nodes.h. It chains all local variables and aliases of a process in the pr_LocalVars element of the process structure, see chapter **??**. The lv_Node.ln_Name element is the full path of the variable, including all directory elements and its name, separated by forward slashes ("/"). The type of the node, namely lv_Node.ln_Type identifies whether this structure describes a variable or an alias. It can be either LV_VAR or LV_ALIAS.

lv_Flags contains flags. Currently, only a single flag is used here, and that is GVF_BINARY_VAR. If this flag is set, then a non-printable text was set as contents of the variable. However, this flag is currently only set, but never used by AmigaDOS; instead, the flags argument of GetVar() determines whether the variable contents is interpreted as text or as binary value.

lv_Value is a pointer to the value of the variable. This value is *not* a NUL-terminated string, but rather an array of bytes whose size is found in lv_Len.

On success, this function does not alter IoErr(). If no matching variable or alias is found, this function returns NULL and sets IoErr() to ERROR_OBJECT_NOT_FOUND.

15.4.4 Deleting a Shell Variable

The DeleteVar() deletes a global or local shell variable or alias.

```
success = DeleteVar( name, flags ) /* since V36 */
D0                   D1    D2

BOOL DeleteVar(STRPTR, ULONG )
```

This function removes a shell variable or alias from the local or global pool of shell variables. It is equivalent to SetVar(name,NULL,0,flags) introduced in section **??**.

The name argument is the name of the variable or alias to remove. It is not case-sensitive. This name may contain slashes ("/") that work similar to path separators. This allows variables to form a hierarchy similar to files in a file system. Without additional flags, this function first attempts to find a local variable and then deletes it if it exists. If no such variable exists, the function attempts to find a global variable of the same name and attempts to delete it. It does not delete directories of variables.

The flags argument provides information on which type of variables or alias are to be deleted. The following flags are honored:

LV_VAR deletes a regular shell variable.

LV_ALIAS deletes an alias. Aliases can only be local. This is mutually exclusive to the above.

If GVF_GLOBAL_ONLY is set, then only a global variables is to be deleted; the function then does not attempt to find a local variable of the given name.

If GVF_LOCAL_ONLY is set, then only a local variable is to be deleted.

If GVF_SAVE_VAR is set, and a global variable is deleted, the change will be made permanent by also deleting the variable in the ENVARC assign.

This function returns a non-zero return value in case of success. IoErr() is then not set consistently. On error, this function returns DOSFALSE and IoErr() is set to an error code. This function will set IoErr() to ERROR_OBJECT_NOT_FOUND if an attempt is made to delete a non-existing variable or alias.

15.5 Command Line Argument Parsing

The functions in this section support applications in parsing arguments from the command line. Because the entire command line except the command name is delivered literally to programs run from the shell, it is possible to use custom algorithms to extract arguments from it, e.g. in order to deliver them in the form of an `argv[]` array to the `main()` function of a C program.

The functions in this section have the advantage that they are aware of the delicate syntax of the AmigaDOS Shell and its quoting and escaping rules as laid out in section **??**; they also integrate a simple help system for command line tools. Because argument parsing is provided as a service of AmigaDOS, command line tools do often not need to be linked with startup code provided by C compilers, allowing such tools to remain small and memory efficient.

15.5.1 Parsing Command Line Arguments

The `ReadArgs()` function parses one or multiple arguments from the command line of the caller using a template, and writes them into a user-supplied array.

```
result = ReadArgs(template, array, rdargs) /* since V36 */
D0                 D1       D2      D3

struct RDArgs *ReadArgs(STRPTR, LONG *, struct RDArgs *)
```

This function is the generic argument parser of AmigaDOS, and highly recommended for all command line tools as it provides a consistent interface to shell commands. The arguments a command expects is encoded in a human-readable template supplied as first parameter, and parsed arguments are placed into `array`. If `rdargs` is `NULL`, then `ReadArgs()` retrieves its input from the buffer of the standard input of the calling process. Other sources are possible, and the function is not restricted to the input stream. This function requires that the command line is terminated by a newline character (`0x0a`).

The `template` parameter specifies the arguments the parser shall retrieve from its input. It describes their names by keywords, their type, whether they are optional or mandatory, and whether the keyword is required on the command line to fill the argument, or whether an argument can also be matched by position.

Each command line argument to be filled is represented by a keyword, and an optional abbreviation that is separated from the full keyword by an equals-sign ("="), e.g. "`QUICK=Q`". Additional options defining the type and parsing rule of the argument may follow the keyword or, if present, the abbreviation, and are separated from them by a forwards slash ("`/`"). Most options can be combined, and are then also separated from each other by slashes. Such options describe whether a particular argument is mandatory, optional, numerical, whether the keyword is required to match an argument or whether arguments are filled by position. The complete list of options is found below. For example, `QUICK=Q/S` indicates a Boolean switch matching the keyword `QUICK` or `Q` on the command line, and the option the `/S` indicates that it is a switch. The keywords are separated by commas ("`,`") from each other.

`ReadArgs()` retrieves keywords and arguments one by one through the `ReadItem()` function introduced in section **??**, and through that function implements the quoting and escaping rules from section **??**. It then matches keywords on the command line to keywords in the template; a keyword either stands for itself for Boolean flags, or is followed by its value, either separated by spaces, tabulators or an equals-sign ("="), i.e. `FROM=org` or `FROM org` on the command line assigns to the keyword `FROM` in the template the value `org`. If no keyword is present, then remaining arguments that do not explicitly require keywords are filled left to right from the command line.

Upon return, each argument is placed in one of the elements of the `array` argument of `ReadArgs()`, the first argument into the first element, the second into the second, in the order they appear in the template. While this is formally an array of `LONG`s, the actual type of an element depends on the nature of the argument as specified through the options following the keyword, and can be a `UBYTE *`, a `UBYTE **` or a

LONG * casted to LONG instead. The array shall be created and zero-initialized upfront by the caller of ReadArgs(), and it shall be large enough to hold one entry per keyword in the template.

The following options are supported:

/S The argument is a Boolean switch. The corresponding entry in array will be set to 0 if the keyword is not present on the command line, and set to a non-zero value if it is present at least once.

/T The argument is a Boolean switch which expects an explicit argument to set its state[10]. If the argument is On or Y, the entry in array will be set to DOSTRUE. If the argument is Off or N, the entry in array will be set to DOSFALSE. The acceptable values for this argument *are not* localized and rather hard-coded.

/K This argument is only filled in if its keyword appears on the command line, arguments without /K and without /S are also matched by position. For example, if the template is FROM/K, then its element in array is not filled in unless FROM appears on the command line. Remaining command line arguments are filled to non-/K keyword slots left to right.

/A This argument is required. If it is not present on the command line, ReadArgs() returns an error.

/N The argument is a number. If the argument is present on the command line, then the corresponding element in array is filled with a *pointer* to a LONG. If the argument is not present, the element in array remains NULL. This allows the caller to identify numeric arguments that have not been provided.

/M This keyword matches multiple strings. Any command line argument that could not be matched to the template will be associated to this keyword. Clearly, at most one /M keyword may be specified in a template. The corresponding entry in array consists of a pointer to an array of UBYTE *s, each of them contains one of the matching arguments of the command line. The last entry in this array is NULL, indicating its end. For example, if the template is FROM/M,TO, then a command line such as a b c TO d will fill the second element of array with "d", but the first 3 will match FROM. Thus, the first element of array will be a pointer to an array of 4 pointers, the first of which will point to the NUL-terminated string "a", the second to "b", the third to "c" and the forth pointer will be NULL.

If there are also /A keywords in the template, i.e. arguments that are required, and some of them are not yet matched to anything on the command line, then ReadArgs() will fill them from the end of the M keywords. This allows templates such as FROM/A/M,TO/A where the final, last argument on the command line fills the TO keyword even without the keyword explicitly appearing on the command line. The Copy command is a typical application of such a template as it requires one destination directory, but also one or many source files to copy.

/F The entire rest of the command line is matched to this argument, even if keywords appear in it. The corresponding entry in array is filled with a single pointer to a string. Such templates are used, for example, by the Alias command as the rest of the command line can form a command by itself and can contain keywords.

Without any option, the keyword is a non-required string argument that is matched from the command line either by the keyword or by position. The corresponding entry in array is in that case a UBYTE *, that is a pointer to a NUL-terminated string that is filled in if the argument is present, or remains NULL if the argument is not found on the command line.

The rdargs parameter customizes ReadArgs() and provides an alternative source of the command line through the RDArgs structure defined below. The same structure is also used for internal resource

[10] **[?]** and also the autodocs state that /T defines a Boolean toggle that changes its value with each appearance, but this information is not correct.

management of ReadArgs(). If NULL is passed in here, this function will allocate and initialize a RDArgs structure itself, and the command line to be parsed will be taken from the buffer of the input file handle of the calling process, i.e. pr_CIS (see section **??** and chapter **??**).

To customize command line parsing, a RDArgs structure as documented in dos/rdargs.h shall be allocated through AllocDosObject(), see section **??**, and shall be initialized as specified below. This structure reads as follows:

```
struct RDArgs {
  struct  CSource RDA_Source;     /* Select input source */
  LONG    RDA_DAList;             /* PRIVATE. */
  UBYTE  *RDA_Buffer;             /* Optional buffer. */
  LONG    RDA_BufSiz;             /* Size of RDA_Buffer (0..n) */
  UBYTE  *RDA_ExtHelp;            /* Optional extended help */
  LONG    RDA_Flags;              /* Flags for any required control */
};
```

Its most relevant part is the RDA_Source structure, which allows to provide an alternative source for the string to be parsed. It is also documented in dos/rdargs.h and is defined as follows:

```
struct CSource {
  UBYTE   *CS_Buffer;
  LONG     CS_Length;
  LONG     CS_CurChr;
};
```

In this structure, CS_Buffer points to the string representing the command line to be parsed, and CS_Length is the size of this string in characters. The CS_CurChr element is the index of the first character in the string to be considered for parsing. This element is typically set to 0 upfront, but if ReadArgs() is used for parsing a longer string buffer in multiple calls, this element may be carried over from a previous parse. If CS_Buffer is NULL, ReadArgs() will retrieve the command line from the file handle buffer of pr_CIS of the calling process, as if the rdargs argument of ReadArgs() was set to NULL.

The RDA_DAList element of the RDArgs structure is private to ReadArgs() and shall not be read or written to. This element is initialized to NULL by AllocDosObject().

The RDA_Buffer element contains a buffer which will be used by ReadArgs() to store parsed data. This buffer space will be used to store parsed arguments, pointers to parsed numbers for the /N option, or arrays of pointers to the parsed arguments for the /M option. If RDA_Buffer is NULL, then ReadArgs() will supply one instead. The size of this buffer in bytes is in RDA_BufSiz. ReadArgs() will allocate more buffers itself if the initially supplied buffer is not large enough, or none is provided. Typically, callers would leave the allocation of the buffer to ReadArgs() unless memory allocation should be avoided altogether and the size of the supplied command line is known to be limited.

The RDA_ExtHelp string is an optional extended help string. If the command line consists of a single question mark, then first ReadArgs() prints its template argument to the standard output of the caller, and requests arguments again by reading from the standard input. If yet another question mark is provided as input and RDA_ExtHelp is not NULL, then this extended help is printed and a third attempt is made to read command line arguments. This feature allows users to request help on the called command, and enter the command line arguments again after reading the template and the extended help.

The RDA_Flags element allows callers to further refine the functionality of ReadArgs(). The available flags are defined in dos/readargs.h and are as follows:

RDAF_STDIN is defined, but not in use.

RDAF_NOALLOC If this flag is set, it instructs ReadArgs() to never allocate or extend its buffer. Only the buffer supplied through RDA_Buffer will be used. If this buffer overflows, parsing aborts and IoErr() is set to ERROR_NO_FREE_STORE.

RDAF_NOPROMPT If this flag is set, the single question mark as request for help is not honored but taken as a literal argument on the command line, that is, command line help through the supplied template parameter or RDA_ExtHelp is not provided.

On success, ReadArgs() returns a RDArgs structure, either the same as passed in through rdargs, or one that was created by this function; as this structure administrates the resources acquired during argument parsing, it shall be released by FreeArgs(), see section **??**, once all arguments have been interpreted and worked on. FreeArgs() shall be called regardless whether the RDArgs structure was implicitly allocated by the ReadArgs() call, or upfront through AllocDosObject(). Note that the strings and string arrays pointed to by the array are part of the resources released by FreeArgs().

On failure, this function returns NULL. All resources such as buffers allocated through ReadArgs() will be released. If a custom rdargs structure was provided by the caller through the rdargs parameter, then this structure is *not* released; instead FreeDosObject() (see **??**) shall be called to dispose a user provided structure.

Beside errors related to input and output operations, the following additional error codes can be returned in IoErr() on failure:

ERROR_NO_FREE_STORE indicates that either the function failed to allocate storage for its buffers, or extending its buffer was explicitly disabled through the RDAF_NOALLOC flag and the supplied buffer space was not large enough.

ERROR_KEY_NEEDS_ARG indicates that the template required a particular argument to be present, namely through the /A option, though the supplied command line did not define a value for this argument.

ERROR_LINE_TOO_LONG indicates that the supplied command line was too long and could not be processed. This happens for example if a /M (multi-argument) keyword is present in the template and too many arguments are supplied.

15.5.2 Releasing Argument Parser Resources

The FreeArgs() function releases all resources allocated by ReadArgs().

```
FreeArgs(rdargs) /* since V36 */
           D1

void FreeArgs(struct RDArgs *)
```

This function releases all resources acquired by the ReadArgs() argument parser. The rdargs argument shall be the return value of ReadArgs(). FreeArgs() releases all temporary buffers associated to the rdargs structure, and in case it was allocated by ReadArgs(), also the structure itself. If a custom allocated RDArgs structure was passed into ReadArgs(), then first FreeArgs() shall be called to release the system-allocated resources, and then the structure itself shall be disposed by FreeDosObject() introduced in **??** at a later point.

As FreeArgs() potentially releases all elements of the array argument of ReadArgs() as part of its resources, the contents of this array are no longer usable afterwards. If array is supposed to be reused for another ReadArgs() call, all its elements shall be set to zero again.

From AmigaDOS version 39 onward, this function clears the RDA_Buffer element of rdargs upfront, which can point to one of the resources that are about to be released. While this makes a rdargs structure allocated through AllocDosObject() safe to reuse for another ReadArgs() call, it also makes a

custom allocated buffer installed there unavailable. Thus, if such a custom buffer is supposed to be reused, RDA_Buffer must be reinitialized[11].

15.5.3 Reading a Single Argument from the Command Line

The ReadItem() reads a single item from the command line or from a provided CSource structure.

```
value = ReadItem(buffer, maxchars, input) /* since V36 */
D0                D1        D2      D3

LONG ReadItem(STRPTR, LONG, struct CSource *)
```

This function reads a single argument from the command line, i.e. the buffer of the input file handle of the calling process, or the supplied CSource structure. This structure is defined in dos/rdargs.h and specified in section **??**.

An argument is delimited by spaces, tabulators, semicolons (";") or equals-signs ("="), unless they are enclosed in double quotes. A newline or EOF always terminates an argument, within or outside of quotes. Section **??** provides more details on the Shell syntax.

If an argument is quoted, the first class of escape sequences listed in section **??** are applied, and the quotes are removed from the argument before it is copied into the buffer. This includes the sequences *", *E, *N and **. Such escape sequences are not relevant *outside* of quotes, and the asterisk there stands for itself.

Once an argument delimiter is found, ReadItem() terminates; while a line feed, semicolon or EOF is moved back into the source, spaces, tabulators or equal-signs are removed from the input buffer before this function returns[12].

The buffer argument provides a user-supplied buffer into which the parsed off argument is copied. This buffer has a capacity of maxchars; if this buffer overflows, ReadItem() aborts parsing and returns ITEM_ERROR. It does not attempt to allocate an extension buffer, unlike ReadArgs().

The input argument provides an alternative source for the command line to be parsed; this source consists of a buffer pointer, a character count and an offset in the form of a CSource structure defined in section **??**.

This function returns the following result codes defined in dos/dos.h:

ITEM_EQUAL an equals-sign ("=") was found as delimiter. This may be used as an indication that the parsed string establishes a keyword, and not an argument.

ITEM_ERROR the supplied user buffer is too small or the source run out of data or run into a newline before a matching closing double quote was found for an opening double quote.

ITEM_NOTHING no argument could be retrieved because the source run into the end of the command line, i.e. either a newline, semicolon or EOF.

ITEM_UNQUOTED an unquoted item was found and retrieved from the command line.

ITEM_QUOTED a quoted item was found and retrieved.

ReadItem() does not set IoErr(), even if ITEM_ERROR is returned.

15.5.4 Find an Argument in a Template

The FindArg() function finds the index of a keyword in a ReadArgs() compatible template.

[11] [**?**] recommends to zero RDA_Buffer manually. This is no longer necessary, though re-installing a custom buffer is.
[12] Unlike what is stated in [**?**] only a subset of the delimiters is unread.

```
index = FindArg(template, keyword) /* since V36 */
D0                D1        D2

LONG FindArg(STRPTR, STRPTR)
```

This function scans a template as provided to `ReadArgs()` for a specific keyword and returns the (zero-based) count of the keyword within the template. See section **??** for how templates look like.

The `template` argument a pointer to a `NUL`-terminated C string defining the template; the syntax of the template is specified in section **??**.

The `keyword` argument provides a keyword that is to be found within the template. It is provided as a pointer to a `NUL`-terminated C string. Matching the keyword to the template is case-insensitive.

This function returns a zero-based index which argument in the template matches the supplied keyword matches, i.e. 0 for the first keyword, 1 for the second and so on. It returns -1 if no matching argument is found in the template. `IoErr()` remains unmodified, even if no matching keyword is found in the template.

15.6 Resident Segments

To ease access to often used program code, AmigaDOS administrates a list of resident segments. This list includes resident commands, and thus allows the Shell to execute them quickly without requiring to load them from disk; but also ROM-resident system segments such as the Shell are placed there during the initialization of AmigaDOS.

The list of resident segments should not be confused with the exec list of resident modules kept in `SysBase->ResModules`, even though its purpose is quite similar[13].

Segments enter this list in multiple ways: First, commands whose `p` protection bit (see section **??**) is set may be added to this list explicitly through the `Resident` command. This bit indicates that the command is "pure", which means that it does not alter its code or data, and its code can be executed from several processes at once. Unfortunately, AmigaDOS makes no attempt to verify these requirements and thus depends on the compiler or user to indicate such pure commands correctly.

Second, if a command has the `p` and `h` protection bits set, it is automatically added to the list of resident segments at the first time such a command is executed. This allows users to indicate often used commands that then stay resident automatically. The `h` stands for "hold", meaning that a command will be held resident once used.

Third, the AmigaDOS Shell adds its own built-in commands during initialization to this list. This includes elementary commands such as `CD` or `Execute` that are contained in the Kickstart and do not need to be loaded from disk.

Fourth, some non-command segments are added to the list during initialization of AmigaDOS through System-Startup, which is discussed in more detail section **??**. These include the CON-Handler, the RAM-Handler, the FFS which is represented by a resident segment named "`FileHandler`", and — finally – the Shell.

The latter is even added three times to this list. First, as "`shell`", which is the the shell that is used by default by the `Shell` icon and the `System()` function. A user shell may replace this entry and thus become accessible by the `Shell` icon on the Workbench. Second, as "`BootShell`", which shall remain unchanged. It always refers to the shell that booted the system, and is also the shell that is used by the `Execute()` function discussed in section **??**. Third, as "`CLI`", which is only present for legacy reasons and not used by the system at all, unless requested explicitly by the `SYS_CustomShell` tag of the `System()` function, see section **??**. Unlike the above two entries of the resident list, it uses BCPL binding and as introduced in section **??**.

[13] This duplication of structures is again a historic accident due to the origin of AmigaDOS.

A resident segment is described by the Segment structure defined in dos/dos.h:

```
struct Segment {
  BPTR  seg_Next;
  LONG  seg_UC;
  BPTR  seg_Seg;
  UBYTE seg_Name[4];
};
```

This structure shall never be allocated by a user program, it is created by AddSegment() instead, see section **??** more details. The elements of this structure have the following meaning:

The seg_Next element is a BPTR to the next segment in the list of resident segments; it is ZERO for the last resident segment. Thus, resident segments form a singly linked list. This BPTR shall not be modified by the user.

The seg_UC element is a use counter, or a type identifier. If the value is positive, the resident segment represents a disk-based command that had been added to the list either implicitly by the shell due to a set h bit, or had been made resident explicitly.

In such a case, seg_UC is incremented every time the segment is used, e.g. when the shell locates the command on the resident segments and executes its code, and decremented once the segment is not used anymore, e.g. if the resident command terminates execution. It is confusingly initially 1 (not 0) if the segment is unused. Resident commands with a seg_UC counter larger than 1 cannot be unloaded as they are currently in use[14].

Despite these regular entries used for resident disk-based commands, the following special values for seg_UC exist; they are also defined in dos/dosextens.h:

The value CMD_INTERNAL indicates a command that is available for execution, but whose code is part of the ROM. The AmigaDOS Shell adds these commands to the resident list during its initialization.

Segments indicated by the use counter set to CMD_SYSTEM are not commands that could be executed, but rather segments of system components. The RAM-Handler, the CON-Handler and the FFS there named "FileHandler" are indicated as system segments. These segments cannot be executed and are not found by the Shell when scanning the resident list for commands.

Segments with a seg_UC smaller or equal than CMD_DISABLED are neither found by the Shell; they are temporarily disabled. This type is created if the Resident command is used to remove a built-in command or a system segment.

The seg_Seg element is a BPTR to the segment list corresponding to the loaded or resident command or system segment. It is a segment list as returned by LoadSeg(), see also chapter **??** and section **??**.

The seg_Name is not only 4 characters long as indicated, but a variable length field. It contains the name of the command or system segment as BSTR.

15.6.1 Find a Resident Segment by Name

The FindSegment() locates a resident segment by name.

```
segment = FindSegment(name, start, system) /* since V36 */
D0                    D1     D2     D3

struct Segment *FindSegment(STRPTR, struct Segment *, LONG)
```

[14] To add to the confusion, the Resident command prints not seg_UC, but seg_UC-1.

This function locates the segment named name in the list of resident segments, using a case-insensitive string comparison. This function scans either the entire list if start is NULL, or at all segments following the segment pointed to by start. This allows continuing a scan for another segment of the same name. It returns the resident segment found, or NULL in case no matching segment is found. Note that the returned pointer is a Segment structure as defined in section **??**, and not a BPTR to a segment of a binary executable as returned by LoadSeg().

The system flag indicates whether the function scans for regular segments or system/internal segments. If system is DOSFALSE, only regular entries with a positive seg_UC counter are located. If system is DOSTRUE, only system or built-in segments whose seg_UC value is negative are found. In the latter case, the above function does not attempt to filter out disabled segments; if required, this shall be done by the caller.

This function neither increases the use counter of a found resident segment and thus, it cannot guarantee that the segment will be unloaded by another process at the time the function returns. Thus, this function shall be called while a Forbid() is active, and the caller shall increment seg_UC for non-system segments it attempts to use before calling Permit().

Unlike regular segments, system segments cannot go away, they can only be marked as CMD_DISABLED, and thus seg_UC need not to be altered by the caller.

This function does not alter IoErr() on success, and it sets it to ERROR_OBJECT_NOT_FOUND in case no matching segment could be found.

15.6.2 Adding a Resident Segment

The AddSegment adds a segment to the list of resident segments and makes it available to other processes.

```
success = AddSegment(name, seglist, type) /* since V36 */
D0                   D1     D2      D3

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

This function adds the segment list seglist, e.g. as returned by LoadSeg(), under the name given by name to the list of resident segments. The seg_UC type and use counter is initialized to type.

The name is copied and its original buffer may be released or reused after the function returns, but the segment becomes part of the system database of resident segments.

The value of type shall be selected as follows: For regular pure executables[15] that are made resident and by that available to other processes, type shall be set to 1.

For system segments that are not commands, type shall be set to CMD_SYSTEM; this value is defined in dos/dosextends.h. Note that such segments cannot be safely unloaded anymore and remain resident for the lifetime of the system.

While other values such as CMD_INTERNAL and CMD_DISABLED are possible, they do not serve a practical purpose. The first one indicates commands that are provided by the AmigaDOS Shell, and this value shall only be used by the ROM Shell. The second value indicates system segments or built-in commands that are currently disabled. It is only used by the Resident command on an attempt to remove system or internal segments.

On success, this function returns a non-zero value and does not alter IoErr(). On error, it returns DOSFALSE and sets IoErr() to ERROR_NO_FREE_STORE.

[15]The official autodocs and [**?**] are wrong in this regard, they suggest 0, but this value is incorrect and neither used by the Shell nor the Resident command.

15.6.3 Removing a Resident Segment

The RemSegment() function removes a resident segment from the system list of resident segments and makes it unavailable to other processes. It also releases all resources associated to this segment.

```
success = RemSegment(segment) /* since V36 */
D0                    D1

BOOL RemSegment(struct Segment *)
```

This function attempts to remove the resident segment provided as argument from the AmigaDOS list of resident segments. If an attempt is made to remove a system segment, an internal command or a regular resident command that is currently in use, DOSFALSE is returned and the resident segment remains in the list. Otherwise, the segment is removed, and all its resources are released, including the segment list associated to the resident segment which is purged through UnLoadSeg().

The resident segment to be removed is typically acquired by FindSegment(). However, any other process can attempt to unload the same resident segment at the same time, causing a race condition. Therefore, a caller shall stop multitasking with Forbid(), then locate the segment with FindSegment(), and if this function succeeds, unload the found segment with RemSegment(). Then, finally, Permit() shall be called to re-enable multitasking.

If the segment passed in cannot be removed because it is in use or is a system segment, this function returns DOSFALSE and sets IoErr() to ERROR_OBJECT_IN_USE. On success, it returns DOSTRUE[16].

15.7 Writing Custom Shells

AmigaDOS allows adding custom shells to the system that may optionally also replace the AmigaDOS Shell. A shell is a resident system segment, see section **??**, similar to the CON-Handler or the RAM-Handler on the same list. When launching a shell with the System() function of section **??**, a custom shell may be requested by providing the name of its resident segment to the SYS_CustomShell tag. A custom shell may even replace the AmigaDOS Shell it by making it resident under the name "shell". On the AmigaDOS Shell, the following command will perform this step:

```
resident shell MyShell replace system
```

where MyShell is the file name of the new shell. However, a shell is not a regular program and requires to go through a custom startup mechanism.

First of all, AmigaDOS supports a BCPL shell under the name "CLI" which will be initialized as BCPL program using the BCPL runtime binding protocol explained in section **??**. AmigaDOS also supports C and assembler based shells under the names "shell" and "BootShell". A user shell replacing the "shell" entry in the resident segment list will thus follow the C/Assembler binding.

Regardless of whether the Shell is run as BCPL or C program, it receives a startup-package in the form of a DosPacket structure similar to handlers and file systems. The startup code from section **??** provides a minimal interface for BCPL runtime binding and is also safe to use for shells implemented in C or assembler. The example code in this section assumes that this startup code is used as initial segment of the shell code; it will call the main() function of the provided example.

Even though there is no need to interpret the elements of the startup packet as of AmigaDOS version 47 anymore, the following information is provided for the sake of completeness:

If dp_Type is non-zero, and dp_Res1 and dp_Res2 are both 0, this is a regular shell startup, and this is the only startup that still exists in AmigaDOS version 47, thus no particular test for dp_Type needs to

[16] Due to a defect in the current version of the *dos.library* it unfortunately *also* sets IoErr() to the same error code on success.

be made above version 45. Under AmigaDOS versions 36 to 45, the following legacy startup types existed: if dp_Type was 0, then this used to indicate the creation of the "Initial CLI" booting the system, and if dp_Res1 and dp_Type were non-zero, this used to indicate the creation of a shell through the NewShell command. Under AmigaDOS version 34 and before, dp_Type was a pointer to a BCPL function that performed the initialization of the shell.

In addition, the AmigaDOS Shell launches instances of itself as new processes, for example to implement pipes or the run-back operator ("&"), see sections **??** and **??**, and to indicate this private startup mechanism, the AmigaDOS Shell uses an even different combination of values in dp_Type, dp_Res1 and dp_Res2. As this is a shell-internal mechanism that is not imposed by AmigaDOS, it will not to be documented here. Custom shells may use whatever mechanism they seem fit if they need to start instances of itself, provided it does not conflict with the above identification of the AmigaDOS startup packet.

Next, the shell shall allocate all system resources it requires, and if this step fails, the startup code shall set the dp_Res1 element of the startup packet to 0, place an error code in dp_Res2, then reply the packet by sending it back to dp_Port and exit.

If this first initialization succeeds, the segment array of the calling process in pr_SegList needs to be reorganized, see also chapter **??**. The regular process startup code of AmigaDOS places the segment list of running program in index 3 of this array, though commands executed by the shell will require this entry themselves, and thus the shell shall move its own segment from entry 3 to entry 4.

Initialization continues with the *dos.library* function CliInitRun() which extracts parameters from the startup packet and from that initializes the CommandLineInterface structure representing the shell. This structure keeps the publicly accessible status of the shell, it is specified in section **??**.

The CliInitRun() function returns a LONG that identifies whether the initialization of the CLI structure was successful. If bit 31 is clear, and IoErr() is equal to the pointer to the shell process, i.e. to FindTask(NULL), initialization failed, and the packet was already replied[17] by CliInitRun(). In this case, the shell shall release all its resources and exit immediately.

If no failure was reported from CliInitRun(), its return value configures the shell, and defines when to reply the startup packet. More on this in section **??**. At this point, the shell is initialized as far as AmigaDOS is concerned and can start its work.

The following code implements a minimal shell startup code:

```
#include <exec/types.h>
#include <exec/alerts.h>
#include <dos/dos.h>
#include <dos/dosextens.h>
#include <proto/exec.h>
#include <proto/dos.h>
#include <string.h>

/* Shell startup bits */
#define FN_VALID        31
#define FN_ASYNC        3
#define FN_SYSTEM       2
#define FN_USERINPUT    1
#define FN_RUNOUTPUT    0
#define FROM_SYSTEM     ((1L << FN_VALID) | (1L << FN_SYSTEM))
#define IS_ASYNC        ((1L << FN_VALID) | (1L << FN_ASYNC))
```

[17]This communication protocol is surely needlessly bizarre, and this case a legacy of Kickstart 2.0 and not part of the Tripos shell startup.

```
void __asm main(register __a0 struct DosPacket *pkt)
{
  struct DosLibrary *DOSBase;
  struct ExecBase   *SysBase = *(struct ExecBase **)(4L);
  struct Process    *proc    =  (struct Process *)(FindTask(NULL));

  /* C startup needs to wait for the */
  /* startup packet manually         */
  if (pkt == NULL) {
    struct Message *msg;
    WaitPort(&proc->pr_MsgPort);
    msg = GetMsg(&proc->pr_MsgPort);
    pkt = (struct DosPacket *)(msg->mn_Node.ln_Name);
  }
  /* check the packet for validity */
  if (pkt->dp_Type == 0 || pkt->dp_Res2 || pkt->dp_Res1) {
      /* Some other form of startup, not from the Os */
      Alert(AN_CLIObsolete|AT_DeadEnd);
  } else {
   LONG fn;
   BPTR *segs;
   struct CommandLineInterface *cli;
   /* Perform shell initialization, here an example */
   DOSBase = (struct DosLibrary *)OpenLibrary("dos.library",47);
   if (DOSBase == NULL) {
     /* Initialization failed, return a meaningful  */
     /* error by replying the packet                */
     pkt->dp_Res1 = 0;
     pkt->dp_Res2 = ERROR_INVALID_RESIDENT_LIBRARY;
     PutMsg(pkt->dp_Port,pkt->dp_Link);
     /* Done with it */
     return;
   }
   /* fixup the segment array */
   segs = BADDR(proc->pr_SegList);
   /* move shell's seg list to the next slot */
   if (segs[4] == NULL) {
     segs[4] = segs[3];
     segs[3] = NULL;
   }
   /* Perform system initialization through */
   /* the dos.library                       */
   fn = CliInitRun(pkt);
   /* Was this an error?                    */
   if (fn > 0 && IoErr() == (LONG)proc) {
     /* An error, release resources and die */
     CloseLibrary((struct Library *)DOSBase);
     /* The Packet is already replied       */
     return;
   }
   if ((fn & IS_ASYNC) == IS_ASYNC) {
```

```
      /* check for async system call */
      /* so reply now */
      ReplyPkt(pkt,pkt->dp_Res1,pkt->dp_Res2);
      pkt = NULL;
    }
    /* Main function continues                */
    pkt = ShellMain(pkt,fn,SysBase,DOSBase);

    cli = Cli();
    /* Shutdown for run and System(NULL,...) */
    if (!(fn & (1L << FN_VALID)))
      fn = (1L << FN_RUNOUTPUT) | (1l << FN_ASYNC);

    /* Shutdown code                          */
    Flush(Output());
    if (!(fn & (1L << FN_USERINPUT)))
      Close(cli->cli_StandardInput);
    if (fn & (1L << FN_RUNOUTPUT))
      Close(cli->cli_StandardOutput);
    /* if not yet replied, do finally now    */
    if (pkt)
      ReplyPkt(pkt,cli->cli_ReturnCode,cli->cli_Result2);
  }
}
```

15.7.1 Initializing a new Shell

The `CliInitRun()` function initializes a process and its `CommandLineInterface` structure from a shell startup package.

```
flags = CliInitRun( packet ) /* since V36 */
D0                   A0

LONG CliInitRun( struct DosPacket * )
```

This function is part of the shell initialization. The `packet` is the `DosPacket` the shell received as startup information; the function initializes from it all elements of the `CommandLineInterface` structure stored in the `pr_CLI` BPTR of the calling process.

The arguments of the shell startup packet are intentionally not documented here, and AmigaDOS may change or extend how the packet is populated in the future.

`CliInitRun()` returns a collection of flags packet in a 32-bit `LONG` value. Unfortunately, these flags are not defined in any of the official AmigaDOS headers; instead, the following names for these bits are suggested here:

Table 15.1: CliInitRun() flags

Acronym	Bit Number	Notes
`FN_VALID`	31	indicates that remaining bits are valid
`FN_ASYNC`	3	asynchronous execution intended
`FN_SYSTEM`	2	shell terminates on EOF of `cli_CurrentInput`
`FN_USERINPUT`	1	if 0, close `cli_StandardInput` on exit
`FN_RUNOUTPUT`	0	if 1, close `cli_StandardOutput` on exit

If bit FN_VALID flags is 0 and IoErr() equals pointer to the pointer of the process structure of the caller, then initialization failed. At this point, the passed in packet has already been replied, and the shell shall only release the resources it acquired so far and exit. In all other cases, packet has not yet been returned to the caller yet.

If the bit FN_VALID of flags is 0 and IoErr() does *not* equal the pointer to the process structure of the caller, then initialization was successful. In such a case, the startup packet shall only be replied after the first command to be executed by the shell has been loaded, leaving the dp_Res1 and dp_Res2 unaltered from the values left by CliInitRun(). When the shell terminates, the shell shall close the cli_StandardOutput and cli_StandardInput file handles of the CommandLineInterface structure. This type of startup is used when the shell is instructed to execute commands in the background, e.g. by the Run command, and if Execute() Execute() or System() are called with NULL as its first argument and SYS_Asynch is not set, see section **??**.

The reason for delaying the startup packet until after loading of the first command is to avoid unnecessary head movements of floppies as otherwise two processes could attempt to access the disk simultaneously: the shell starting a command with Run, and the shell loading the command in the background.

If the bits FN_VALID and FN_ASYNC are set, then asynchronous command execution is requested. The shell shall then reply the startup packet immediately, leaving dp_Res1 and dp_Res2 unaltered from the values left by CliInitRun(). If FN_VALID is set and FN_ASYNC is not set, then the startup packet shall only be replied when the last command of the shell returned. Then, dp_Res1 shall be set to the return code of the last command as found in cli_ReturnCode, and dp_Res2 shall be set to the error code the command left in cli_Result2. The FN_ASYNC bit reflects the value of SYS_Asynch of the System() call.

If bits FN_VALID and FN_SYSTEM are set then the shell shall terminate if the cli_CurrentInput runs into an EOF and there is no higher level script to continue executing from. This corresponds to the case where the first argument of System() is non-NULL. In such a case, the commands from this string are executed, which is delivered through a string stream placed in cli_CurrentInput. Once done, the shell stops and returns. Shells created this way are never interactive, i.e. cli_Interactive shall always be DOSFALSE.

If the FN_VALID flag is set and FN_SYSTEM is reset, then the shell continues to read commands from cli_StandardInput once cli_CurrentInput depletes and there is no higher level script to continue executing from. Only if the two input streams are equal and and EOF condition is detected, the shell terminates. This corresponds to the case where the the shell was initiated through Execute() with a non-NULL command string.

If bits FN_VALID and FN_USERINPUT are set, then the shell *shall not* close cli_StandardInput when terminating. If FN_VALID is set and FN_USERINPUT is reset, then cli_StandardInput shall be closed on exit.

If the bits FN_VALID and FN_RUNOUTPUT are set, then cli_StandardOutput shall be closed when the shell process terminates. Otherwise, if FN_VALID is set and FN_RUNOUTPUT is reset, then the cli_StandardOutput shall remain open. Note that the FN_RUNOUTPUT logic is the inverse of the FN_USERINPUT logic.

A successful startup with FN_VALID reset is thus approximately equivalent to the bit combination of FN_RUNOUTPUT and FN_ASYNC, except that the elements of the CommandLineInterface structure are initialized differently and the startup packet is replied at a different stage.

The following source code implements a main program of a very primitive shell which lacks many features such as variable expansion, redirection, resident commands, a configurable prompt, it does not even scan the path. This main program fits to the shell startup code in section **??**.

```
/* A rather primitive shell main program */
```

```
struct DosPacket *ShellMain(struct DosPacket *pkt,LONG fn,
    struct ExecBase   *SysBase,
    struct DosLibrary *DOSBase)
{
  struct CommandLineInterface *cli = Cli();
  UBYTE  cmd[256];
  UBYTE  args[256];
  LONG   result;
  LONG   ch;

  do {
    do {
      /* Compute the interactive flag */
      cli->cli_Interactive    = (!cli->cli_Background  &&
                                 cli->cli_CurrentInput ==
                                 cli->cli_StandardInput &&
                                (fn & FROM_SYSTEM) != FROM_SYSTEM)?
                               DOSTRUE : FALSE;

      SelectInput(cli->cli_CurrentInput);
      SelectOutput(cli->cli_StandardOutput);

      if (cli->cli_Interactive) {
        if (!cli->cli_Background) {
          Printf("SimpleShell > ");
          Flush(Output());
        }
      } else {
        /* Check for script termination */
        if (CheckSignal(SIGBREAKF_CTRL_D)) {
          PrintFault(ERROR_BREAK,"SHELL");
          break;
        }
      }

      /* Read the command */
      SetIoErr(0);
      cmd[0]          = 0;
      cli->cli_Module = 0;

      ch     = FGetC(Input());
      UnGetC(Input(),ch);
      result = ReadItem(cmd,sizeof(cmd),NULL);
      if (result == ITEM_UNQUOTED || result == ITEM_QUOTED) {
        cli->cli_Module = LoadSeg(cmd);
        /* Reply to pkt if not done so */
        if (pkt && !(fn & (1L << FN_VALID))) {
          ReplyPkt(pkt,pkt->dp_Res1,pkt->dp_Res2);
          pkt = NULL;
        }
        if (FGets(Input(),args,sizeof(args))) {
```

```
      if (cli->cli_Module) {
        SetProgramName(cmd);
        cli->cli_ReturnCode =
          RunCommand(cli->cli_Module,
                     cli->cli_DefaultStack << 2,
                     args,(LONG)strlen(args));
        cli->cli_Result2 = IoErr();
        if (cli->cli_ReturnCode >= cli->cli_FailLevel &&
            !cli->cli_Interactive) {
          Printf("%s failed : %ld\n",cmd,cli->cli_ReturnCode);
          break;
        }
        if (cli->cli_Module)
          UnLoadSeg(cli->cli_Module);
      } else {
        cli->cli_Result2    = IoErr();
        cli->cli_ReturnCode = 10;
        PrintFault(cli->cli_Result2,cmd);
        if (cli->cli_ReturnCode >= cli->cli_FailLevel &&
            !cli->cli_Interactive)
          break;
      }
    }
  }
  /* abort on system */
  if (((fn & FROM_SYSTEM) == FROM_SYSTEM) &&
      (cli->cli_CurrentInput == cli->cli_StandardInput))
    break;

} while(ch != -1);

/* cleanup command file left by execute */
{
  UBYTE *commandfile = BADDR(cli->cli_CommandFile);
  if (commandfile && *commandfile) {
    SetProtection(commandfile+1,0);
    DeleteFile(commandfile+1);
    commandfile[0] = 0;
  }
}

if (cli->cli_CurrentInput == cli->cli_StandardInput) {
  /* endcli only if its an interactive CLI */
  if (!cli->cli_Background)
    Printf("SimpleShell terminating...");
  break;
} else {
  /* Script is done, revert input */
  Close(cli->cli_CurrentInput);
  cli->cli_CurrentInput = cli->cli_StandardInput;
  /* If coming from System(), close down */
```

```
      if ((fn & FROM_SYSTEM) == FROM_SYSTEM) {
        break;
      } else {
        cli->cli_FailLevel    = 10;
      }
    }
  } while(1);

  return pkt;
}
```

Historically, two additional functions existed to initialize a shell in AmigaDOS version 36 to version 45. `CliInit()` had to be called by the `Initial CLI` and not only initialized the shell to read from the `S:Startup-Sequence`, it also initialized the *dos.library* and mounted all ROM-resident handlers. These tasks are now taken over by a Kickstart module of its own, *System-Startup*, which in its final step, creates the `Initial CLI` through the `System()` function. Section **??** provides more details on this module.

The second function to initialize a shell was `CliInitNewShell()` which was exclusively used by a shell created by the `NewShell` command. In the latest version of AmigaDOS, `NewShell` also calls through `System()` and thus only depends on `CliInitRun()`.

The shell startup mechanism under AmigaDOS 34 and below was even different. There, `dp_Type` was a pointer to a BCPL function the shell had to call to get initialized by AmigaDOS.

Chapter 16

Miscellaneous Functions

In this section, miscellaneous functions are specified that logically do not belong into any other section. These are constructor and destructor functions for AmigaDOS objects, and functions for error handling.

16.1 Object Constructors and Destructors

Some AmigaDOS objects shall not be allocated manually through `AllocMem()` or other generic memory allocation functions. This is either because the generic functions do not initialize such objects fully, or because the objects contain internal elements beyond the end of their documented structure such that the `sizeof` operator of the C language does not describe their true size. A paired destructor function destroys such objects and releases all resources associated to them.

16.1.1 Allocating a DOS Object

The `AllocDosObject()` function constructs instances of various objects used by the *dos.library* and initializes their elements. The functions listed in this section all take a `type` ID that describes the type of the object to be created, and additional arguments that are used to initialize this object. The functions all correspond to the same entry in the *dos.library* and only differ in their name and their calling conventions.

The `AllocDosObject()` and `AllocDosObjectTagList()` functions are identical and take additional parameters in the form of a tag list. The second function only exists to following naming conventions and to allow tools to automatically generate prototypes for the third.

The `AllocDosObjectTags()` function receives the tag list in the form of a variably sized argument list consisting of tags and whose last tag shall be `TAG_DONE` terminating the list.

```
ptr = AllocDosObject(type, tags) /* since V36 */
D0                   D1    D2

void *AllocDosObject(ULONG, struct TagItem *)

ptr = AllocDosObjectTagList(type, tags) /* since V36 */
D0                          D1    D2

void *AllocDosObjectTagList(ULONG, struct TagItem *)

ptr = AllocDosObjectTags(type, Tag1, ...) /* since V36 */

void *AllocDosObjectTags(ULONG, ULONG, ...)
```

The above functions all create objects of the *dos.library* and initialize them according to the tag list. The `type` argument provides the type of the object to be constructed; types are defined in `dos/dos.h`:

Table 16.1: AllocDosObject() type IDs

Type	Description
DOS_FILEHANDLE	construct a `FileHandle` structure, as in **??**
DOS_EXALLCONTROL	construct a `ExAllControl` structure, see **??**
DOS_FIB	construct a `FileInfoBlock` structure, see **??**
DOS_STDPKT	construct a `StandardPacket` structure, as in **??**
DOS_CLI	construct a `CommandLineInterface` structure, see **??**
DOS_RDARGS	construct a `RDArgs` structure, as defined in section **??**

The `DOS_FILEHANDLE` type creates a `FileHandle` structure as introduced in section **??**. This structure describes an opened file and shall only be created through this function as it contains some hidden internal elements. Creating this structure is documented to take one parameter, namely `ADO_FH_Mode` defined in `dos/dostags.h`, which is the mode in which the file is to be opened. This tag takes an argument from table **??** of section **??** which corresponds to the second argument of the `Open()` function specified in the same section. It defaults to `ACTION_FINDINPUT`, i.e. the the file handle describes a file that is opened non-exclusively. File handles are typically created by the *dos.library* only. However, even though this tag may sound useful[1], nothing in the initialization of the handle currently depends on its value.

The `DOS_EXALLCONTROL` type creates an `ExAllControl` structure used to iterate over directory contents, see section **??**; no additional tags are required.

The `DOS_FIB` type creates a `FileInfoBlock` structure as defined in section **??**, no further tags are necessary. This structure may also be manually constructed as it does not contain any hidden elements. However, the structure shall be aligned to long word boundaries. If the memory is taken from the stack, the macro `D_S()` in section **??** should be used to ensure alignment.

The `DOS_STDPKT` type creates a `StandardPacket` structure used for communication between clients and handlers or file systems. It is an aggregate of a `Message` and a `DosPacket` structure that are chained correctly to each other. The `DosPacket` structure is discussed in section **??**. There is no need to use a `StandardPacket` for handler communication provided the `DosPacket` is aligned to a long word boundary and linked to a `Message`, see section **??**. `AllocDosObject()` does actually not return a pointer to the `StandardPacket` itself, but rather to its `DosPacket` element. This constructor does not take any additional tags.

The `DOS_CLI` type creates a `CommandLineInterface` structure as defined in section **??**. While it was possible to construct this structures manually in previous versions of AmigaDOS, this is no longer the case from AmigaDOS 47 onward as it has been extended by private elements. A shell which is equipped with a manually constructed `CommandLineInterface` structure will not offer the full functionality of regular AmigaDOS Shells.

While the sizes of various buffers of the created structure can be set by the following tags, they are actually not particularly useful. Nothing in the CLI structure indicates the actual buffer sizes and the AmigaDOS Shell therefore uses hard-coded limits instead:

`ADO_PromptLen` is the size of the `cli_Prompt` buffer containing the formatting string of the prompt, in bytes.

`ADO_CommNameLen` defines the size of the `cli_CommandName` buffer in bytes, containing the file name of the currently loaded command.

`ADO_CommFileLen` selects the size of the `cli_CommandFile` buffer in bytes; this buffer holds the name of a temporary script file generated by `Execute`.

[1] Tags that would create string streams would be even more useful.

ADO_DirLen defines the size of the cli_SetName buffer in bytes which contains the path of the current directory.

The DOS_RDARGS type creates a RDArgs structure used by the ReadArgs() function for command line argument parsing, see section **??**. This constructor is also called internally by the *dos.library*, but if a custom RDArgs structure is required for parsing, for example, from an alternative source and not the standard input, AllocDosObject() shall be used to create one. No tags are defined for this object.

AllocDosObject() returns on success the pointer to the constructed object. Note that for the type DOS_STDPKT this is a pointer to a DosPacket structure and not to the StandardPacket structure. The AllocDosObject() function does not alter IoErr() on success. On error, this function returns NULL and sets IoErr() to ERROR_NO_FREE_STORE.

16.1.2 Releasing a DOS Object

The FreeDosObject() function destroys an AmigaDOS object created by AllocDosObject() and releases the resources associated to it.

```
FreeDosObject(type, ptr) /* since V36 */
              D1   D2

void FreeDosObject(ULONG, void *)
```

This function destroys an object created by AllocDosObject(). The type argument is one of the types from table **??** in section **??**, and ptr is a pointer to the object to be destroyed. In case type is DOS_STDPKT, the ptr argument shall be a pointer to the DosPacket element of the StandardPacket aggregate.

In AmigaDOS versions 37 and before, releasing a CommandLineInterface structure did not free all buffers associated this structure and thus caused a memory leak. This was fixed in version 39.

Passing NULL as ptr performs nothing, the function exits in this case without performing any action. This function does not set IoErr().

16.2 Reporting Errors

While AmigaDOS identifies errors by a unique ID returned by IoErr(), only printing a number as error report is not very helpful. The *dos.library* provides multiple functions to generate human readable error messages, either for printing them on the console, or by creating a requester with an error message on a screen.

16.2.1 Display an Error Requester

The ErrorReport function creates an error requester based on an error code and a file system object such as a file handle or a lock and waits for the response of the user who may either abort or retry an activity.

```
status = ErrorReport(code, type, arg1, device) /* since V36 */
D0                   D1    D2    D3     D4

BOOL ErrorReport(LONG, LONG, ULONG, struct MsgPort *)
```

This function creates an error requester either on the Workbench screen, or on the same screen as the window given by the pr_WindowPtr of the calling process. If pr_WindowPtr is -1, then no requester

is shown and the function returns immediately with a non-zero return code, indicating that cancellation of the activity is desired.

This function is implicitly called by most functions of the *dos.library* in case of error, and thus rarely needs to be called by the user explicitly. However, `ErrorReport()` may *also* used by file systems to generate error requesters[2], and in such a case, it checks the `pr_WindowPtr` of the file system process and not that of the file system client; thus, such requesters cannot be easily suppressed by application programs.

The `code` argument is an error identifier returned by `IoErr()`. Only a subset of the error codes listed in section **??** and defined in `dos/dos.h` are supported by this function; all others result in a non-zero return code, indicating cancellation of the activity. Table **??** lists the error codes for which this function is able to create a requester:

Table 16.2: Errors supported by ErrorReport()

Error	Description
`ERROR_DISK_NOT_VALIDATED`	Reports that a disk is corrupt and not validated
`ERROR_DISK_WRITE_PROTECTED`	Reports that a disk is write protected
`ERROR_DISK_FULL`	Reports that a volume is filled completely
`ERROR_DEVICE_NOT_MOUNTED`	A particular handler or file system is not mounted
`ERROR_NOT_A_DOS_DISK`	A volume does not carry a valid file system
`ERROR_NO_DISK`	No physical medium is inserted in the drive
`ABORT_DISK_ERROR`	Reports that a medium has a physical read/write error
`ABORT_BUSY`	Requests to insert a particular medium into the drive

The error `ABORT_BUSY` is defined in `dos/dosextens.h` and shows a requester that the user *MUST* replace a particular volume of a drive. It is generated if a file system needs to write out its dirty buffers to a volume that is no longer inserted into the drive managed by it. This type of error and thus this error report is only generated by file systems and never forwarded to applications by means of `IoErr()`.

`ABORT_DISK_ERROR` is also never directly forwarded to user code and thus does neither appear as return value of `IoErr()`. It shows a requester that a particular volume has a read/write error; file systems generate this error if access to a medium fails on the physical level. Same as the above, this code is only intended to be used by file systems and not by application code. The symbol is again defined in `dos/dosextens.h`.

The `type` argument defines the type of the AmigaDOS object passed in as `arg1`; it provides an additional source of information used to generate the error requester. This information is used for example to include a device or volume name in the requester. The following types, defined in `dos/dosextens.h` are supported by this function:

Table 16.3: Error sources supported by ErrorReport()

Type	Description
`REPORT_STREAM`	`arg1` is a BPTR to a `FileHandle` structure (section **??**)
`REPORT_LOCK`	`arg1` is a BPTR to a `FileLock` structure (section **??**)
`REPORT_VOLUME`	`arg1` is a BPTR[3] to a `DosList` structure (chapter **??**)
`REPORT_INSERT`	`arg1` is a *regular* pointer to an absolute path

If `type` is `REPORT_INSERT`, then `arg1` shall be a pointer to a path that is split at a colon (“`:`”) to extract a volume or device name. This type is used by the *dos.library* when attempting to locate an entry on the device list from an absolute path failed.

The `device` argument is a (regular) pointer to a `MsgPort` structure that is only used if `type` is `REPORT_LOCK` and `arg1` is `ZERO`, thus the lock representing the root of a file system; `device` is also

[2]The FFS *does not* make use of this service and builds requesters itself, though checks its own `pr_WindowPtr` as well.

[3]Unlike what the official documentation and [**?**, **?**] say, this is a BPTR, not a regular pointer.

used if the error indicates that the faulting volume is not accessible. This port is assumed to be a MsgPort of a file handler that is contacted with ACTION_CURRENT_VOLUME of section **??** to learn about the name of the currently inserted volume[4]. This mechanism is for example used when reporting errors through a DevProc structure whose lock and port are passed into the error report, see section **??**.

Finally, for the error ERROR_DEVICE_NOT_MOUNTED and the type REPORT_INSERT, the *dos.library* attempts to find out whether a request for particular volume should be suppressed completely, and for that calls the VolumeRequestHook() function of section **??**. If it returns DOSFALSE, no requester is shown and ErrorReport() aborts immediately, indicating cancellation.

The ErrorReport() function returns a Boolean indicator whether the currently ongoing operation shall be aborted or retried. If the result code is DOSFALSE, the operation shall be retried. This result code is also implicitly generated if the user inserts a medium into a drive while the requester is shown. If the requester is suppressed because the pr_WindowPtr of the calling process is set to -1, or the system run out of memory, or an error not listed in table **??** is reported, then this function always returns DOSTRUE, requesting to abort the operation. On return, this function also sets IoErr() to the error identifier in code.

16.2.2 Receive Information when a Volume is Requested

The VolumeRequestHook() function is called by the *dos.library* whenever it attempts to request the user to insert a volume.

```
res = VolumeRequestHook(volume) /* since V47 */
D0                          D1

LONG VolumeRequestHook(UBYTE *volume)
```

This function is not supposed to be called by clients, it is rather called by the *dos.library* and provided to be patched by application programs that want to learn whenever AmigaDOS is about to show a requester to ask for a specific volume.

The argument this function receives is the name of the volume AmigaDOS is about to ask for, without a trailing colon (“:”). If this function returns DOSTRUE, then the *dos.library* progresses to show a requester to ask the user to insert volume. If this function returns DOSFALSE, the requester is suppressed as if the user pressed “Cancel”, thus refused to insert the volume.

The purpose of this function is to provide assign-wedge functionality that allows users to create an assign to a directory at the first time it is accessed by a program, or to suppress requesters to a volume that is never going to be inserted.

16.2.3 Generating an Error Message

The Fault() function fills a buffer with an error message from an error code and an initial string.

```
len = Fault(code, header, buffer, size) /* since V36 */
D0           D1     D2      D3    D4

LONG Fault(LONG, STRPTR, STRPTR, LONG)
```

[4]If the faulting volume is not accessible, the *dos.library* also attempts to find the name of the handler responsible for the volume, lock or file handle. It does so by matching the port from these objects, namely dol_Task, fl_Task or fh_Type to the ports of the file systems recorded in the device list. This silently assumes that the port which locks, file handles or volumes provide as contact point is identical to the port the file system leaves in dol_Task of its DosList, though this assumption is not necessarily true if the file system uses multiple ports to communicate to its clients. Again, this is probably a defect and instead of testing the port, mp_Task of the candidate ports should be checked. The algorithm that is used if *no* matching file system is found is even weirder as the code then assumes to find an exec device unit number in IoErr(), probably as part of a legacy Tripos interface which required the file system to deposit its unit here before calling ErrorReport(). The FFS, however, no longer uses this function and thus does not require this legacy interface.

This function fills the buffer that is size bytes long with an optional header, followed by a string generated from the error code given as first argument. The generated string does not include a line feed.

If header is non-NULL, it is first copied into the buffer followed by a colon and a space (": "). The colon and space are not copied if header is NULL. This is followed by a (localized) error message created from code. If code is 0, the buffer is left untouched and nothing is inserted into it. If no localized error message is available, the numeric value of the error is inserted instead.

The size of buffer shall not be 0. The byte for the NUL terminator of the string to be created shall be included in size.

Due to a defect present even in AmigaDOS version 47, the return value of this function is unlike documented *not* the number of characters inserted into the buffer and is thus not usable. As a workaround, use strlen() on the buffer if code is non-zero; if it is 0, the buffer has not been modified:

```
if (code) {
  Fault(code,header,buffer,size);
  len = strlen(buffer);
} else {
  len = 0;
}
```

Fault() does not change IoErr().

16.2.4 Printing an Error Message

The PrintFault() function prints an error message consisting of a header and a description of an error code over the error output stream of the calling process.

```
success = PrintFault(code, header) /* since V36 */
D0                   D1     D2

BOOL PrintFault(LONG, STRPTR)
```

This function implements elementary error reporting by printing an error message over the error output pr_CES of the calling process, see chapter **??**. If that stream does not exist, the error is reported over the standard output stream pr_COS of the caller. Under AmigaDOS version 45 and before, this function always printed over the standard output.

If header is non-NULL, this string is printed first, followed by a colon and a space (": "). Otherwise, nothing is printed upfront. If a textual description of the error code code is available, it is printed behind it, followed by a line feed. Otherwise, just a generic error message consisting of the provided error number is printed.

If code is 0, nothing is printed at all and IoErr() remains unaltered. Otherwise, this function sets IoErr() to code.

This function returns a Boolean success code. If the error is 0, or a description of the error is available, this function returns a non-zero success code. Otherwise, this function returns DOSFALSE.

16.2.5 Printing a String to the Error Stream

The PutErrStr() function writes a string to the error output stream of the calling process.

```
error = PutErrStr(str) /* since V47 */
D0                D1

LONG PutErrStr(STRPTR)
```

This function writes the NUL terminated string str over the error output pr_CES of the calling process. If this stream does not exist, this function writes the string over the standard output pr_COS. The process structure and the file handles it contains are explained in detail in chapter **??**.

This function returns 0 on success, or ENDSTREAMCH on an error. The latter constant is defined in dos/stdio.h and equals to -1. The error code IoErr() is only adjusted if the buffer of the used file handle is flushed. Except for the target stream, this function is similar to the FPuts() function introduced in section **??**.

Chapter 17

The DOS Library

This chapter documents the *dos.library* base structure and structures linked from it. These structures should not be accessed directly as the library provides accessor and manipulator functions for its elements.

17.1 The Library Structure

The *dos.library* base structure is documented in `dos/dosextens.h` and looks as follows:

```
struct DosLibrary {
    struct Library      dl_lib;
    struct RootNode    *dl_Root;
    APTR                dl_GV;
    LONG                dl_A2;
    LONG                dl_A5;
    LONG                dl_A6;
/* the following elements exist from V36 onwards */
    struct ErrorString *dl_Errors;
    struct timerequest *dl_TimeReq;
    struct Library     *dl_UtilityBase;
    struct Library     *dl_IntuitionBase;
};
```

The elements of this library are as follows:

The `dl_lib` element forms a regular exec library structure as it is defined in `exec/libraries.h`.

The `dl_Root` pointer points to the `RootNode` structure documented in section **??**. This structure contains global objects of the library[1]. The `RootNode` is created along with the *dos.library* and this pointer remains constant throughout the life time of the system.

The `dl_GV` is the pointer to the (fake) BCPL Global Vector which contains a table of BCPL functions the *dos.library* provides to (legacy) BCPL components. Today, nothing should depend on this vector anymore as all AmigaDOS components have been rewritten in C or assembler, and no particular advantage can be taken from the functions in this vector as its functions are also accessible as regular functions in the *dos.library*.

The `dl_A2`, `dl_A5` and `gl_A6` elements are the initializers for the BCPL runtime system. They are not relevant nowadays anymore. `dl_A2` is a pointer to the Global Vector and thus identical to `dl_GV`. `dl_A5` is a function pointer to the BCPL function caller, and `dl_A6` is a pointer to the BCPL "return from

[1]The reason why this information is not directly located in the library is that the *dos.library* is historically a thin wrapper around Tripos that was referenced from the library.

subroutine" function. These functions reorganize the stack to align it to the BCPL convention of the stack growing towards higher addresses.

The dl_Errors pointer provides (localized) strings for all messages the *dos.library* generates, including error messages. This element is private and shall not be accessed. Instead, localized error messages can be retrieved by the Fault() function of section **??**.

The ErrorString structure is actually documented as follows in dos/dosextens.h:

```
struct ErrorString {
    LONG  *estr_Nums;
    UBYTE *estr_Strings;
};
```

dl_Errors points to an array of the above structures. The estr_Nums element points to two long words identifying the range of error codes (or rather message codes as not only error messages are in this structure) covered by one ErrorString structure. The first long word is the lower (first) error code in the range, the second the upper (inclusive) end of the range. The last element in the array is identified by a lower end of 0.

The estr_Strings points to an array of bytes that contains the sizes and the localized versions of the error messages as one long array of UBYTEs. Each error message is represented by a single byte that gives its length including the terminating NUL, followed by the NUL-terminated message itself. If Fault() requires a particular message, its skips each message by adding the length+1 until either the requested message is found, or the end of the current ErrorString element is reached. However, for all practical purposes, dl_Errors shall be left alone and instead Fault() shall be used to retrieve a particular message.

The dl_TimeReq element is a private pointer to a timerequest structure the *dos.library* uses for functions such as Delay() in section **??** or for retrieving the system time in DateStamp(), see **??**. If a particular process requires it, the *dos.library* first makes a copy of this timerequest structure, fills in pr_MsgPort of the caller as reply port of the request, and then calls through DoIO() the *timer.device* to implement the required function. Similar to all other elements of the *dos.library*, this element shall not be accessed directly, but only through services offered by the library.

The dl_UtilityBase element provides a pointer to the *utility.library* the *dos.library* uses for parsing tag lists and performing elementary arithmetic.

The dl_IntuitionBase is the pointer to the *intuition.library* that is used to generate requesters.

17.2 The Root Node

The RootNode structure is accessible from the dl_Root pointer in the DosLibrary structure and provides global information of the library. It is also defined in dos/dosextens.h:

```
struct RootNode {
    BPTR              rn_TaskArray;
    BPTR              rn_ConsoleSegment;
    struct DateStamp  rn_Time;
    LONG              rn_RestartSeg;
    BPTR              rn_Info;
    BPTR              rn_FileHandlerSegment;
/* The following elements have been added in V36 */
    struct MinList    rn_CliList;
    struct MsgPort   *rn_BootProc;
    BPTR              rn_ShellSegment;
    LONG              rn_Flags;
};
```

The rn_TaskArray element is a BPTR to the (legacy) Tripos process table, and is part of the complete table reachable through rn_CliList. Instead of going through this element, the FindCliProc() function specified in section **??** shall be used. This first array consists of a long word that indicates the size of the first part of the table, all remaining entries of the array are pointers to the pr_MsgPort of Process structures (see chapter **??**) or NULL if the corresponding entry is not used. This array is indexed by the CLI number, and thus only processes that are associated to a shell are listed here.

The rn_ConsoleSegment is the segment of the BCPL entry point of the shell. Despite its name, it is in no correspondence to the CON-Handler or the console.device. Even though this pointer is initialized, it is no longer used. Instead, the shell is located as part of the resident segments by the FindSegment() function which is described in section **??**. The corresponding resident segment whose BPTR is stored here has the name "CLI".

The rn_Time contains the current time of the system, it is updated every time DateStamp() (see section **??**) is called; applications shall not copy the time and date from here, but rather call through the above function to retrieve it.

The rn_RestartSeg is obsolete and no longer used. AmigaDOS version 34 and before stored here a BPTR to a segment that performed validation of inserted disks. If this validator found an invalid bitmap, see section **??**, it loaded the "L:Disk-Validator" from disk which then recomputed the bitmap by a full disk scan. With the introduction of the FFS in AmigaDOS version 34, disk validation became part of the file system, even though this element remained in use for the BCPL-based OFS.

The rn_Info element is a BPTR to the DosInfo structure described in section **??**. Like the RootNode, it remains constant throughout the lifetime of the system and contains additional global information.

The rn_FileHandlerSegment element is the BPTR to the segment of the ROM default AmigaDOS file system. Only the expansion.library function AddDosNode() function still accesses this element, but it should be considered obsolete and private. It is populated by the resident segment "FileHandler", and is even updated if a newer version of the FFS is found in the RDB of the booting disk. From AmigaDOS version 34 onward, file systems should be obtained through the *FileSystem.resource* and not through this BPTR which only provides the FFS.

The rn_CliList contains the full process table containing all shell processes; its entries are accessible through FindCliProc(). The total number of CLIs found in here is returned by MaxCli(), see section **??**. This element is a MinList structure as defined in exec/lists.h. Each node on this list keeps a consecutive range of shell processes and looks as follows:

```
struct CliProcList {
  struct MinNode   cpl_Node;
  LONG             cpl_First;
  struct MsgPort **cpl_Array;
};
```

This structure *shall not* be allocated by the user; instead, it is potentially created by the *dos.library* if the current process table is too small when launching a new shell through System() or Execute().

The cpl_Node is a MinNode structure that queues all CliProcList nodes in the rn_CliList.

The cpl_First element is the CLI number of the first shell in this node, and thus the process number of the shell kept in cpl_Array[1].

The cpl_Array element contains the number of entries in the table and pointers to the pr_MsgPorts of the Process structures running shells. The first element cpl_Array[0] entry is the number of shell processes administrated in this node and is typically 20. All remaining entries are pointers to MsgPort structures, namely the pr_MsgPort of the processes executing the shells; the process structure is defined in chapter **??**.

The cpl_Array of the first CliProcList structure is also accessible through rn_TaskArray, and the nodes kept in rn_CliList are part of an extension that overcomes the limitation of at most 20 shells in AmigaDOS version 34 and before.

The rn_BootProc element is a pointer to the MsgPort of the file system that booted the system. This is not necessarily the file system whose segment is recorded in rn_FileHandlerSegment. It is different if the booting file system is not the ROM-based FFS and neither an updated version of it from the RDB. This pointer is used to initialize the pr_FileSystemTask of all processes AmigaDOS creates, and thus the file system responsible for the ZERO lock.

The rn_ShellSegment is a pointer to the C segment of the shell. This pointer is also not used anymore, but is still initialized. It corresponds to the resident segments "shell" and "BootShell", where the former may be replaced by a custom user-provided shell and the latter is always the shell that booted the system. In AmigaDOS version 47, the shell segment is located via FindSegment() instead.

The rn_Flags element contains flags that configure AmigaDOS. There is currently only one publicly accessible flag defined here, namely RNF_WILDSTAR. If this flag is set, the pattern matcher discussed in chapter **??** supports the asterisk "*" as a synonym for "#?", i.e. the pattern that matches a sequence of arbitrary characters. While this seemingly brings AmigaDOS closer to other contemporary operating systems, it is a rather bad choice as the asterisk is also the file name of the current console and the escape character of the AmigaDOS Shell. It is therefore recommended *not* to set this flag.

17.3 The DosInfo Structure

The DosInfo structure is pointed to by the rn_Info element of the RootNode. It contains the device list structure discussed in chapter **??** and the resident segments of section **??**.

```
struct DosInfo {
    BPTR    di_McName;    /* NOT the resident segments */
    BPTR    di_DevInfo;
    BPTR    di_Devices;
    BPTR    di_Handlers;
    APTR    di_NetHand;   /* actually, a BPTR */
/* the following elements are new in AmigaDOS 36 */
    struct  SignalSemaphore di_DevLock;
    struct  SignalSemaphore di_EntryLock;
    struct  SignalSemaphore di_DeleteLock;
};
```

The include file dos/dosextens.h also defines the name di_ResList for the di_McName element. Unlike what this name suggests, di_McName is *not* the list of resident segments. It is actually not used.

The di_DevInfo element is a BPTR to the first entry of the device list introduced in chapter **??**. This is a BPTR to a DosList structure which form a singly linked list whose head is here. However, client programs should not access this BPTR directly, but rather go through the functions provided in section **??**.

The di_Devices and di_Handlers elements are Tripos legacies and not in use by any version of AmigaDOS.

The di_NetHand element contains the resident segments, despite its name. This element is neither an APTR as the official includes suggest, but is rather a BPTR. It points to a Segment structure as introduced in section **??**, and all resident segments are queued here in a singly linked list. This list of resident segments should not be accessed through this element, but through the functions in section **??**.

The di_DevLock, di_EntryLock and di_DeleteLock semaphores are private to the *dos.library*. They are used by the LockDosList() and related functions of section **??**.

17.4 The AmigaDOS Boot Process

For starting the system, AmigaOs initializes the resident modules stored in `SysBase->ResModules` in the order of decreasing priority. The last module that is initialized in this way is the `strap` module. It locates the boot volume, and either performs boot-point or boot-block booting, see [**?**] for more details. The boot code of the host adapter responsible for the boot volume, or the bootstrap code from the boot block either loads an alternative operating system, a self-booting program, or starts AmigaDOS by initializing the resident module named `dos.library`.

The boot code in the host adapter ROM or in the boot block therefore includes a `InitResident()` call which starts the *dos.library*. In AmigaDOS version 36, this process was augmented by also setting a flag in the *expansion.library* that prevents opening the boot console immediately. The standard boot block on floppies created by the `Install` command, in fact, performs *only* the above two steps. Even though continuing AmigaDOS initialization from AmigaOs initialization is rather minimal, the *dos.library* is *not* part of the `exec` module initialization chain, even though it is represented as a resident modules.

The *dos.library* then first builds a minimal instance of its library lacking lacking many resources. Under AmigaDOS version 45 and before, initialization is completed through the shell running the `InitialCLI`. The shell startup code, see section **??**, received a startup packet that requested the shell to call `CliInit()`. This function then completed the initialization of AmigaDOS, including opening a console window as standard output and the `S:Startup-Sequence` script as input. The shell then starts the system by reading commands from this file.

This changed in version 47 which separates AmigaDOS initialization from the initialization of the boot shell. There, instead, the *dos.library* searches the list of resident modules for the `System-Startup` module, and initializes a `Process` from it whose purpose is to complete the initialization of the system, see section **??**. The final step of this process is to create a boot shell through the `System()` function which then starts executing the `S:Startup-Sequence`. Thus, the `System-Startup` module can be roughly compared to the `init` process of Unix like systems.

17.4.1 The System-Startup Module

The first step of `System-Startup` is to complete the initialization of the *dos.library*; this includes allocating the `DosInfo` structure (see section **??**) and initializing it. Next, the task array `rn_CliList` is initialized and linked into the `RootNode` structure described in section **??**.

In the following step, the `DosList` structures created by autobooting devices stored in the *expansion.library* are carried over into the device list, more on the `DosList` structure is found in chapter **??**. This mounts all devices that were recorded in expansion. The first one added to this list will become the boot volume.

The `System-Startup` process then locates the FFS in the exec list of resident modules. It is represented there as a module named "`filesystem`". If an autobooting host adapter loaded a newer version of the FFS from the Rigid Disk Block (RDB), this version replaces the ROM version. It is then also used to update the mount entries of all other devices using the FFS, for example the disk drive `DF0`. From this file system, or the ROM version of the FFS if no newer version was found, a resident segment (see **??**) named "`FileHandler`" is created and linked into the `DosInfo` structure.

In the next step, `System-Startup` creates the mount entries for `PRT`, `PAR`, `SER` and `PIPE`, and then starts the file system of the boot device. This is possibly, but not necessarily the FFS found in the previous step. While the FFS is only a file system, it also contributes to the bootstrap process by checking whether a valid system time is available, and if not so, initializes the date and time of AmigaDOS from the creation date of the boot volume.

After devices, the system assigns are created, namely `L`, `FONTS`, `DEVS`, `LIBS`, `S`, `C`, `SYS` and `ENVARC`. The latter is a late-binding assign pointing to "`SYS:Prefs/Env-Archive`".

After the file system of the boot device has been started in the previous step, at this point all remaining file systems are started and thus their devices become accessible as well.

Next, another set of ROM-resident handlers are mounted, namely CON and RAW. The corresponding handler segment is taken from the exec resident module "con-handler", but as devices are already running at this point, System-Startup also looks into the L directory of each running device to locate a newer version replacing the ROM version, unless ROM Updates are explicitly disabled in the boot menu. Thus, the boot device can replace the ROM version of the handler, similar to the FFS which can be updated from the RDB. The RAM device is mounted next; its segment is also taken either from the ROM module "ram-handler", or from the L: directory of mounted devices if they contain a newer version of the handler and ROM Updates are permitted.

Locating the boot shell is the next step, namely as "shell" from the exec resident modules, or from the L: directories if permissible. However, due to AmigaDOS version 34 legacies, this time the disk-based module is named "Shell-Seg". This segment is placed in the elements rn_ConsoleSegment and rn_ShellSegment of the RootNode structure, and the resident segments "CLI", "BootShell" and "shell" are created from it.

Unless disabled in the boot menu, System-Startup now checks for updates of ROM modules on mounted volumes. If a resident module is a library, the LIBS directories are scanned, and for devices, the DEVS directories are checked. Even the *intuition.library* can be replaced by a disk based version, though the procedure is more involved because intuition is already working at this point. However, version 47 of intuition was extended by the ability to shut itself down for replacement. If a newer disk-based version is found, the ROM-based library is torn down, the existing library entry points are patched to a call to Alert(), and the new library base is installed into the *dos.library* and the BCPL Global Vector. File systems should therefore not buffer the intuition library base in its code, but rather open it only when needed.

In the next step, AGA extensions are enabled if the boot menu was displayed in a 31kHz DblPPAL or DblNTSC mode. This step is would usually be performed by SetPatch, though is already executed here; it allows users to show the boot console on a VGA monitor even without requiring to load SetPatch in order to trace or debug the Startup-Sequence if the ECS/OCS resolutions cannot be displayed.

This is followed by initializing all resident ROM modules of lower priority than *dos.library* marked as RTF_AFTERDOS. That includes for example the *audio.device* and the *mathffp.library*. Note that these modules are not necessarily coming from ROM, they could have been replaced by disk-based modules by the ROM update mechanism executed before.

The next step consists of loading the system preferences from DEVS:System-Configuration and installing them into intuition. While this is a legacy mechanism that is overloaded by the IPrefs program, it enables users to already customize the boot console to a limited degree, such as setting its colors and the shape of the mouse pointer. The boot console is opened next, using the AUTO mode of the CON-Handler, see section **??**. This delays opening the console window, and by that also of the intuition screen it appears on, until some program attempts to print text on the console.

Unless booting from the S:Startup-Sequence is disabled, this file is now opened to become the command stream of the Shell. Next, it is checked whether the boot code of the autobooting device or the code in the boot block of a floppy indicated the AmigaDOS version 36 boot protocol by testing a private flag in the *expansion.library*. If this flag is *not* set, a legacy boot block is assumed and the console and boot screen is forced open by printing a carriage return (0x0d) character. For the legacy boot protocol, the current directory of the boot shell is changed to ZERO as compatibility measure for old applications. Otherwise, this lock is set to the root directory of the boot device, i.e. to SYS:.

In the following, System-Startup checks whether logging was enabled in the boot menu. If so, the "syslog" module is located either from the list of resident modules, or from the L: directory if ROM Updates are permitted and a newer version is available there. If it is found and enabled, the "debug" shell variable is set to "on", and a the syslog process is started. This process continuously monitors the exec debug output function, namely RawPutChar(), though which diagnostic information is written by many

debugging tools. By that, diagnostic output is redirected to the file `RAM:syslog` where it can be inspected by the user. One source of diagnostic output is the Shell itself which echos all commands it executes if the "`debug`" variable is found enabled.

If tracing was enabled in the boot menu, the shell variable "`interactive`" is set to "`on`". This variable is also tested by the AmigaDOS Shell; if it is set, the Shell requests confirmation for each single command it executes. This allows users to trace through the `Startup-Sequence` and therefore identify problems more easily.

As final step, the `InitialCLI`, that is, the Boot Shell, is launched through a `System()` function call. Unless executing the `S:Startup-Sequence` is disabled, the command stream is opened from this file and provided to the Shell for execution. Once this Shell is running, the `System-Startup` process terminates and leaves it to the Shell to continue the boot process and boot the system to the Workbench.

Bibliography

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

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

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

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

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

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

[7] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, 2nd edition, Prentice Hall (1988)

[8] ECMA: *Control Functions for Coded Character Sets.* ECMA International (1998)

[9] Brian Sawert *The Programmer's Guide to SCSI*, Addison-Wesley Publishing Company (1998)

FIRST EDITION

AMIGA TECHNICAL REFERENCE SERIES

AMIGA ROM Kernel Reference Manual

AmigaDOS

The Amiga computers are exciting high-performance microcomputers with superb graphics, sound, multiwindow and multitasking capabilities. Their technologically advanced hardware is designed around the Motorola 68000 microprocessor family and sophisticated custom chips. The Amiga's unique system software provides programmers with unparalleled power, flexibility, and convenience in designing and creating programs.

This volume provides a long missing addendum to the AMIGA ROM Kernel Reference Manual Series and adds a comprehensive description of the AmigaDOS system component. The ***AmigaDOS Reference Manual*** presents a detailed description of the central I/O and process subsystem of AmigaOs. This volume covers the latest version of Amiga's operating system. It includes:

- An introduction into elementary components of AmigaDOS such as files and locks
- AmigaDOS handlers and file systems,and how to interface them
- Pattern matching and recursive directory scanning
- A discussion of the AmigaDOS Process management
- A complete breakdown of the Fast File System disk structure
- Direct packet communication and a comprehensive documentation of DosPackets
- A detailed description of the AmigaDOS executable and link file structure
- The documentation of the AmigaDOS ROM Shell, including a tutorial and source code for implementing custom shells
- The AmigaDOS startup mechanism

For the serious programmer who wants to take full advantage of the Amiga's impressive features, the ***Amiga ROM Kernel Reference Manual: AmigaDOS*** is an indispensable source of information on how to use the AmigaDOS file system, how to interact and implement handlers and file systems and how to make productive use of the AmigaDOS Shell.