# Amiga ROM Kernel Reference Manual

# DOS

## 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 provides an interface between such applications and AmigaDOS subsystems such as file systems or the Shell.

While the Amiga ROM Kernel Reference Manuals [7] 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 the Tripos to the Amiga, an experimental operating system developed at the University of Cambridge. Instead, its documentation became available as the AmigaDOS manual [1] 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 [10], 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 library and its subsystems, closely following the style of the ROM Kernel Reference Manuals.

### 1.1.1 Acknowledgements

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 answering many questions while compiling many chapters of this volume.

## 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 practise and recommendations that are advisable, but not strictly necessary to satisfy a particular interface. The word *may* provides a hint to a possible implementation strategy.

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

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

Terms are indicated in *italics*, e.g. the *dos.library* implements interface of *AmigaDOS* 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 Data Types

## 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 data carriers such as floppy disks or hard disks.

Unlike many other operating system, the *dos.library* does not manage disks or files itself, neither does it provide access to hardware interface components. It rather implements a *virtual file system* which forwards requests to its subsystems, called *handlers* or *file systems*. The FFS is only one of the multiple file systems AmigaDOS is able to handle; instead, it provides an abstract interface to handlers that allows its extension by third-party file systems and handlers. Handlers and file systems are introduced in section 2.9, and the interface between the *dos.library* and such handlers is specified in chapter 11.

## 2.2   The DosLibrary Object

The *dos.library* is typically opened by the startup code of most compilers anyhow, 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 its elements are usually not required and should not be accessed directly. However, chapter 15 provides further details and provides informations on all its elements. The most important function of the library is to provide functions to applications due to its library vector offsets — or short `_LVOs` — 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 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 in brackets behind the function def-
inition. 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 avail-
able, and then take the maximum of all the version numbers and supply this maximum as second argument
of `OpenLibrary()`. Table 1 provides the relation between the (published) AmigaDOS versions and the
corresponding library version:

**Table 1: AmigaDOS Version Numbers**

| AmigaDOS version | dos.library version |
|---|---|
| 1.2 | 33 |
| 1.3 | 34 |
| 2.0 | 37 |
| 3.0 | 39 |
| 3.1 | 40 |
| 3.9 | 43 |
| 3.1.4 | 45 |
| 3.2 | 47 |

Most important changes were made in version 37 in which the library was largely extended.

## 2.3   Booleans

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

**Table 2: DOS Truth Values**

| Define | Value |
|---|---|
| DOSFALSE | 0 |
| DOSTRUE | -1 |

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

made whether particular values are 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 descendent 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 implementation had to preserve the existing interface based on BCPL conventions.

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

> *Not on the Stack!* Since BPCL structures must have an address that is divisible by 4, using automatic storage duration for AmigaDOS objects is inappropriate. Compilers will typically keep such objects on the hardware stack of the 68K processor [5], 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 14.1.1), a safe strategy is use exec.library memory allocation functions such as *AllocMem()* or *AllocVec()* to obtain memory for holding them. All three functions ensure proper alignment.

Indicies 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 pointer */
```

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

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

Luckely, in most cases callers of the *dos.library* do not need to convert from and to BPTRs but can rather use such "pointers" as *opaque values* or *handles* representing some AmigaDOS object. Examples for those objects are *file handles* specified in chapter 4, and *locks*, see section 5. 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 longer 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 pointer to a properly aligned `struct FileInfoBlock`, e.g. this is equivalent to

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

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

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

```
#define ZERO 0L
```

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

## 2.5   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 BPCL 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 pointer to BCPL string  */
```

Luckely, functions of the *dos.library* take C strings as arguments and perform the conversion from C strings to their BCPL representation as *BSTR*s internally, such that one rarely gets in contact with this type of strings. They appear as part of some AmigaDOS structures to be discussed, and as part of the interface between the *dos.library* and its handlers, e.g. file systems. However, even though users of the *dos.library* rarely come in contact with 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 an 8-bit byte 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 as of the latest version of *AmigaDOS*, the *dos.library* is ill-prepared to take longer strings, and will likely fail or mis-interpret the string passed in. If longer strings are 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 section 4.3.

## 2.6  Files

Files are streams of bytes together with a file pointer that identifies the next position to be read, or the next byte position to be filled. Files are explained in more detail in chapter 4. AmigaDOS represents files through *file handles* in the form of a *FileHandle* structure, though most of the time, only BPTRs to such handles are passed around.

## 2.7  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 altered or removed by any other process, a locked directory can be altered, but not be deleted. Chapter 5 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 to this structure.

## 2.8  Processes

AmigaDOS is a multi-tasking system operating on top of the *exec* kernel [7]. As such, it can operate multiple tasks at once, where the tasks are assigned to the CPU in a round-robin fashion. A *Process* is an extension of an AmigaOs *Task* that includes additional state information relevant to AmigaDOS, such as a *current directory* operates in, a default file system, a *console* it is connected to, a default input, output and error stream. Most importantly, processes include a `MsgPort`, see [7], through which it communicates with other AmigaDOS components such as handlers or file systems. Processes are explained in more detail in chapter 9.

## 2.9  Handlers and File Systems

*Handlers* are special processes that perform input or output operations to a logical or physical device, such as the serial port, a printer, the floppy or even the RAM. AmigaDOS itself delegates most operations, such as reading from a stream or file to such handlers. Handlers are explained in more detail in chapter 11.

*File systems* are special handlers that organize the contents of data carriers such as hard disks, floppies or CD-Roms in the form of files and (optionally) directories. File systems interpret paths (see 4.3) in order to locate objects such as files and directories on such data carriers. Thus, every file system is a handler — i.e. the Fast File System (FFS) is a handler, discussed in section 11.7 — 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 11.3 provides more details on this particular handler.

## 2.10  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, splits those lines into words, and uses the first word as command to which all following words are supplied as arguments. Commands are binary executable files (see chapter 10 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 can also read commands from any other handler, for example from a file on a partition operated by the Fast File System, and then interprets the commands from such a *Shell Script* or *Batch File*. The most

important Shell script is the *Startup-Sequence* which is interpreted by the Shell when booting AmigaOs. Chapter 13 provides the 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, users may install additional alternative shells. The AmigaDOS interface to shells is specified in section 13.7.

# 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 1$^{st}$ 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 are as follows:

`ds_Days` counts the number of days since January 1$^{st}$ 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`. The ticks, and the seconds that can be derived from it, do not include leap seconds that are added from time to time. In case leap seconds are added, the Amiga clock needs to be adjusted manually.

> *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 is currently limited to 31$^{th}$ 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 15.2) of the *dos.library* remain 0, corresponding to a system date of January 1$^{st}$ 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 11.7.4 and adjusts the system time then through the *timer.device*.

## 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` function and returns an indicator which of the dates are 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 amount 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, makking it available for 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 in the (binary) AmigaDOS representation to human-readable strings, and in the reveverse direction. 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 to create, 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: 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 |

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 this string 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 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.

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
```

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

**Table 4: 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 te current system date, the output date is set to "Yesterday", or a localized version of this string.

If the date 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, same as the `DateToStr()` function would generate them. If the flag is set, weekdays are assumed as references to the future.

`dat_StrDay`: This buffer is only used by 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 colon, e.g. `21:47:16`. If this field is `NULL`, then the time is not converted.

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.

### 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 3.2 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 3.

`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 weekday. Week days always correspond to past days, e.g. "Friday" corresponds to the 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 $60 \times 24$ or the number of ticks is larger than $50 \times 60$. 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 3.2 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 3 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 4. 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

# Files

## 4.1   What are Files?

*Files* are streams of sequences of bytes that can be read from and written to, along with a file pointer that points to the next byte to be read, or the next byte to be written or overwritten. Files 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.

## 4.2   Interactive vs. non-Interactive Files

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

*Non-interactive* files are stored on some persistent data carrier 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 4.4.1 and chapter 5), or even parts of a file (see section 4.10).

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, executed; this so-called *meta-information* is discussed in more detail in section 6.1.

*Interactive* files depend on the interaction of the computer system with the outside world, and their contents can change due to such interaction. Interactive files 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 serial 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 may block indefinitely if the printer runs out of paper or is turned off.

## 4.3 Paths and File Names

Files are identified by *paths*, which are strings identifying a particular file, and from which AmigaDOS also locates a process through which access to the file is managed. Such processes are called *handlers*, or, in case of non-interactive files organized on a persistent data carrier, *file systems*. The *dos.library* does not operate on files directly, but delegates such work 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. It relates to the name of a handler (or file system) of the given name, or a known disk volume, or a logical volume of the name within the AmigaDOS *device list*. These concepts are presented in further detail in chapter 7.

The second, or only part is interpreted by the handler or file system identified by the first part.

### 4.3.1 Devices, Volumes and Assigns

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

> *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 suffers from a similar 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.

#### 4.3.1.1 Devices

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

Table 5 lists all device names AmigaDOS creates itself even without a boot volume available. They can be assumed present any time as they are created by the AmigaDOS ROM components:

**Table 5: 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 drives are connected to the system, they are named `DF1` through `DF3`. If a hard disk is present, then the device name(s) of the harddisk partitions depend on the contents of Rigid Disk Block,

see [8]. These names can be selected upon installation of the harddisks, e.g. through the HDToolBox. The general convention is to name partitions is `DH0` and following.

The devices `SER`, `PAR`, `PRT` and `PIPE` are created by the kickstart ROM, but the corresponding handler is disk-based and loaded from disk as soon as the 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.

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

**Table 6: 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, is the current console of the process, if such a console exists. Any data output to the file named `*` will be printed on the console. 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 failed to open `*` 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, or is used as escape character in AmigaDOS shell scripts.

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 foregroup 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 announced to the system by the *Mount* command, see chapter 11.

#### 4.3.1.2 Volumes

A *volume name* identifies a particular data carrier within a physical drive. For example, it may identify a particular floppy disk, regardless of the drive it is inserted it. For example, the volume name "Workbench3.2" relates always to the same floppy, regardless of whether it is inserted in the first `df0` or second `df1` drive.

### 4.3.1.3 Assigns

An *assign* or *logical volume* identifies a subset of a files within a file system under a unique name. Such assigns are created by the system or by the user helping to identify portions of the file system containing files that are of particular relevance for the system. For example, the assign C contains all commands of the Boot Shell, and the assign LIBS contains dynamically loadable system libraries. Such assigns can be changed or 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 interchangingly with device or volume names and thus form logical volumes within volumes, or even across 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 the bound directory is not available, the system will ask to insert this particular volume, and no other volume, even of identical name, will be accepted.

Regular assigns can also bind to multiple directories at once, in which case a particular file or directory within such a *multi-assign* is searched in all directories 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 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 accessed, 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 7 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 release 36 of AmigaDOS.

**Table 7: 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 7), it was also added in release 36 of AmigaDOS:

**Table 8: System defined assigns**

| Assign Name | Description |
|---|---|
| PROGDIR | Location of the executable |

Thus, `PROGDIR` is the directory the currently executed binary was loaded from. Note that `PROGDIR` does not exist in case an executable was not loaded from disk, probably because it was either taken from ROM or was made resident. More on resident executables is found in section 13.6[1].

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 9 lists some of them.

**Table 9: Assigns made during bootstrap**

| Assign Name | Description |
|-------------|-------------|
| `ENV` | Storage for active preferences and global variables |
| `T` | Storage for temporary files |
| `CLIPS` | Storage for clipboard contents |
| `KEYMAPS` | Keymap layouts |
| `PRINTERS` | Printer drivers |
| `REXX` | ARexx scripts |
| `LOCALE` | Catalogs and localization |
| `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, see section 11 or the AmigaDOS functions discussed in section 7.6.

### 4.3.2 Relative and Absolute Paths

As introduced in section 4.3, 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 running process.

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 this are provided in section 9.

If neither a colon nor a device, volume or assign name is present, the path is a relative path.

The path name behind the colon, or the entire relative path is forwarded to the handler or file system identified by either the current directory or the device, volume or assign name and is not interpreted by the *dos.library*. It is within the responsibility of the handler to interpret this path and locate a file within the data carrier it manages, or to configure an interface to the outside world according to this path. For example, while the Fast File System (FFS) — the Amiga ROM file system — interprets this path as a location within the file system directory hiearchy, the Conole-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 support functions workable and it is therefore advisable for file system implementors to follow these conventions. Section 4.3.4 provides them.

---

[1]Unfortunately, attempting to access a file relative to `PROGDIR` 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.

### 4.3.3    Maximum Path Length

The *dos.library* does not enforce a limit on the size of file or directory names, except that the total length of a path including all of its components shall not be larger than 255 characters. This is due to the AmigaDOS legacy that requires that paths are represented as BSTRs to the handlers. How large the name of a directory or a file, i.e. a component of a path, can be is a matter of the file system itself. The Fast File System includes variants that limit component (file or directory) names to 30, 56 or 106 characters.

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

### 4.3.4    Flat vs. Hiearchical File Systems

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

For this reason, all file systems provided by AmigaOs are *hierarchical* and organize files in nested sets of *directories*, where each directory contains files or more 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. Note that this is different from some other operating system that use a backslash as component separator.

### 4.3.5    Locating Files or Directories

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

Otherwise, it uses the *current directory* of the calling process to locate a handler responsible for the interpretation of the path name. If this current directory is ZERO (see section 2.4), it uses the *default file system* of the process, which by itself, defaults to the file system that booted the system.

The second part of the path interpretation is up to the file system identified by the first step and is performed there, outside of the *dos.library*. If the path name includes a colon ("`:`"), then locating a file starts from the root of the inserted volume.

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

To locate a file, the file system should work iteratively through the path, component by component: A single isolated "/" without a 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 of given by the component, and continue searching there.

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

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

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

### 4.3.6 Case Sensitivity and Character Encoding

The *dos.library* does not define whether file names are case-sensitive or insensitive, except for the device, volume or assign names which are case-insensitive. Most if not all AmigaDOS file-systems are also case-insensitive, or rather should. Some variants of the Fast File System do not handle case-insensitive comparisons correctly on non-ASCII characters, i.e. ISO-Latin code points whose most-significant bit is set, see section 11.7.4 for details. These variants should be avoided and the "international" variants should be preferred, see table 29 in section 7.1.3.

In general, AmigaDOS support functions such as the pattern matching functions discussed in chapter 8 assume that file systems are case-insensitive, and use the ISO-Latin character set. In particular, the functions in the above chapter will not function correctly if component names include code points from the control set of ISO-Latin, i.e. codes between $0x00$ to $0x1f$ or $0x80$ to $0x9f$. These code points are not printable and multiple functions of AmigaDOS will behave erratically if component names containing them are encountered.

## 4.4 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 simulatenously.

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.

### 4.4.1 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* the file to be opened, which is interpreted according to the rules given in section 4.3. The argument accessMode identifies how the file is opened. The function returns a BPTR to a file handle on success, or ZERO on failure. A secondary return code can be retrieved from IoErr() described in section 9.2.9. It is 0 on success, or an error code from dos/dos.h in case opening the file failed.

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

**Table 10: Access Modes for Opening Files**

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

> *Length Limited* As this function needs to convert the path argument from a C string to a BSTR, path names longer than 255 characters are not supported and results are unpredictable if such names are passed into Open(). If such long path names cannot be avoided, it is the responsibility of the caller to split the path name accordingly and potentially walk through the directories manually if necessary. Note that this strategy may not be suitable for interactive files or for handlers that follow conventions for the path name that are different from the conventions described in section 4.3.5.

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

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 within the path if they do not access. Once the file is opened, access to the file is shared, even if it has been just created. That is, multiple processes may then access it for reading or writing. If multiple processes write to the file simultenously, the order in which the writes are served is undefined and depends on the scheduling of the processes.

For AmigaDOS versions 34 and below, MODE_READWRITE implements a somehwat different access mode. Under such releases, a file system attempting to open a file under this mode does not attempt to create files that do not exist, and exclusive access is granted. 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 5.1.1, and then use this lock to create a file handle from it though OpenFromLock(), see section 5.2.3.

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

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

### 4.4.2   Closing Files

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

```
success = Close( file )
D0                D1

BOOL Close(BPTR)
```

The `file` is a BPTR to a file handle identifying the file. The return code indicates whether the file system could successfully close the file and write back any data. If the result code is `DOSFALSE`, an error code can be obtained through `IoErr()` described in section 9.2.9. Otherwise, `IoErr()` will not be altered. Under release 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 running the process, i.e. 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, unfortunately not much can be done 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 — and thus retrying `Close()` does not provide an advantage.

Attempting to close the `ZERO` file handle under AmigaDOS version 36 or later returns success immediately. Under earlier versions, closing `ZERO` causes a crash.

## 4.5 Testing for Interactive Files

As introduced in 4.2, 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.

### 4.5.1 Obtaining the Type from a File

A file can be either interactive, in which case attempts to read or write data to the file may block indefinitely 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 `TRUE` in case the file handle passed in is interactive, or `FALSE` in case it corresponds to a non-interactive stream of bytes, potentially on a file system. This function cannot fail and does not alter `IoErr()`.

The Port-Handler and the devices `PRT`, `SER` and `PAR` managed by it generate 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 generating interactive files. 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 4.6.2 may be used to test for the availability of data. Interactive files typically do not support repositioning the file pointer through `Seek()` (see 4.6.4) or changing the file size through `SetFileSize()`, see section 4.6.5.

### 4.5.2 Obtaining the Type from a Path

A handler that manages physical data carriers and allows to access named files on such data carriers is a file system. Such handlers will generally create non-interactive files. The `IsFileSystem()` function determines the nature of a handler given a path (see 4.3) to a candidate handler.

```
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 to such a hypothetical object regardless whether it exists or not.

It is advisable to provide a path that identifies the handler uniquely, i.e. a string that is terminated by a colon (":"). Otherwise, the call checks whether the handler responsible for the current directory of the calling process is a file system.

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

This function does not exist prior to release 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, i.e. call `Lock()` on the device name that is to be checked, see section 5.1.1 for this function. A similar workaround is also used internally by the library for handlers that do not support the identification natively. A side effect of this workaround, but also of `IsFileSystem()` is that it will load and start the corresponding handler if it is not already active.

## 4.6 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 performs necessarily a task switch from the caller to the handler managing the file, these functions are inefficient for small amounts of data and should be avoided if only small amounts of data are to be transferred. 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 4.7.

### 4.6.1 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 standard 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 9.2.9. 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 4.2) 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.

### 4.6.2  Testing for Availability of Data

An issue of the `Read()` function is that it may block indefinitely on an interactive file if the user 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 `DOSTRUE`. 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 4.7 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 `DOSTRUE`, then for some handlers, in particular for the CON-Handler, a secondary result code is provided through `IoErr()`. The CON-Handler provides here the number of lines available in the input buffer. In particular, one *shall not* depend on `IoErr()` returning 0 on success.

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

Under AmigaDOS versions prior to V36, a timeout value of 0 does not function properly. This was fixed in V36.

### 4.6.3  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 9.2.9. 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.

### 4.6.4 Adjusting the File Pointer

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

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

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

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

**Table 11: 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 definition 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 9.2.9. It is 0 on success, or an error code from `dos/dos.h` in case adjusting the file pointer failed.

> *Not 64bit safe* Unfortunately, it is not quite clear how `Seek()` operates on files that are larger than 2GB, and it is file system dependent how such files could be handled. `OFFSET_BEGINNING` can probably only reach the first 2GB of a larger file as the file system may interpret negative values as an attempt to reach a file position upfront the start of the file and may return an error. Similarly, `OFFSET_END` may possibly only reach the last 2GB of the file. Any other position within the file may be reached by splitting the seek into chunks of at most 2GB and perform multiple `OFFSET_CURRENT` seeks. However, whether such a strategy 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 system internal buffer. That is, it can be safely used by buffered and unbuffered functions.

### 4.6.5 Setting the Size of a File

The `SetFileSize()` function truncates or extends the size of an opened file to a given size. Not all handlers support this function, 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` shall be selected from to table 11. In particular, it is interpreted as follows:

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

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

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

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

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

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

## 4.7 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 is to be read or written in smaller units. All file handles AmigaDOS allow buffered input and output, and are requipped 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 4.7.4 should be used to request larger buffers.

*Performance Improved* While buffered I/O functions of AmigaDOS V45 and below were designed around single-byte functions and thus caused massive overhead in the buffered functions described in this section, the functions in this section were redesigned in AmigaOs 3.2 and now provide significantly better performance. Unfortunately, the default buffer size AmigaDOS uses is quite small and should be significantly increased by `SetVBuf()`. A suggested buffer size is 4096 bytes which corresponds to a disk block of modern hard drives.

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.

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 13.5.1 retrieves them. Thus, any buffered read function 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 `FFlush()` would be necessary. Unfortunately, the input buffer containing the command line parameters is allocated manually following the algorithm in section 4.8.2, and an `FFlush()` 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 code sequence performs this:

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

The include file `dos/stdio.h` also defines a couple of additional macros for buffered I/O that are slightly inefficient by requiring an additional call to `Input()` or `Output()`. If called 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))
```

While 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 rewrite, they became available to the library interface.

### 4.7.1  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 the `count`.

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

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

### 4.7.2   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 AmigaOs 3.2 onwards, `FWrite()` first checks whether the file handle internal buffer is partially filled. 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 to the internal buffer.

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

### 4.7.3   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)`.

### 4.7.4   Adjusting the Buffer

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

---

[2]The information in [1] on returning −1 on error is incorrect.

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

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

This function sets the internal buffer of the file handle `fh` to `size` bytes. Sizes smaller than 204 bytes will be rounded up to 204[3] bytes. If `buff` is non-`NULL`, it is a pointer to a user-provided buffer that will be used for buffering. A user-provided 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 is updated.

The `type` argument identifies the type of buffering according to Table 12; the modes there are defined in the include file `dos/stdio.h`.

**Table 12: Buffer Modes**

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

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

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

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

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

The function returns 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 onwards. Releases prior version 39 always returned 0. Due to another defect, currently 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`.

### 4.7.5   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 position.

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

LONG Flush(BPTR)
```

---

[3]While the official autodocs and [10] reports a default buffer size of 208 characters, this information is not correct.

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

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

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

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 this section for further details.

### 4.7.6   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.

### 4.7.7   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 adjusted when the buffer is flushed.

### 4.7.8 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. The `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()`.

### 4.7.9 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, then the function attempts to refill the buffer from the file and tries again.

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

To 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()` when 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 is available on the (potentially interactive) input stream. This was fixed in version 37.

### 4.7.10 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; or a line-feed character is found, which is copied into the buffer; or if an end of file condition or an error condition is encountered. In either event, the string is `NUL` terminated.

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

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

In AmigaDOS versions prior to version 39, this function would copy one byte too much if no line terminator such as a newline or end of file is found, and it thus overflows 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.

### 4.7.11   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.

## 4.8   File Handle Documentation

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

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

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

### 4.8.1   The struct FileHandle

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 member. The file system or handler shall initialize this value when opening a file and shall initialize it according to the nature of the handler.

fh_Type points to the MsgPort of the handler or file system that implements all input and output operations. Section 11 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 this value 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 4.7 use.

fh_Pos is the byte offset of the next byte to be read or the next available index for absorbing the next written byte within fh_Buf.

fh_End is the size of the buffer in bytes.

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

fh_Func2 is a second function pointer that is called whenever the buffer is full and is to be written out by the handler. Users shall not call this function itself, and the function prototype is neither documented.

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

fh_Arg1 is a file-system specific value the handler or file system uses 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() and Write() requests. See chapters 11 and 12 for details.

fh_Arg2 is currently unused.

Additional elements have been added at the end of the FileHandle structure in release 39. They are not publicly documented. For this reason, this structure shall never be allocated manually, but shall rather be obtained by opening a file, or through AllocDosObject(), see section 14.1.1.

### 4.8.2 String Streams

It is sometimes useful to provide programs with (temporary) input not coming from a file system or handler directly, even though the program uses a file interface to access it. One solution to this problem is to deposit the input data on the RAM disk, then opening this file and providing it as input to such a program. The drawback of this approach is that additional tests are necessary to ensure that the file name is unique, and to avoid that 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 the command to be executed to the Run command. There, the string stream contains the command(s) to be run in background, which is provided

as command stream input file to the shell. The `System()` function of the *dos.library* makes use of the same trick to feed the command(s) to be executed to the shell as well. Thus, even though the shell reads its commands from a file only, AmigaDOS generates such file handles from strings.

The shell uses the same technique to pass arguments to the commands it executes; it deposits the command arguments in the file handle buffer of the input stream where the `ReadArgs()` function called from the command collects them, see also section 13.5.1.

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. 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.

```
#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;
}
```

### 4.8.3 Cloning File Handles

Unlike locks (see 5.1.2), file handles cannot be duplicated in general. However, if the file is interactive (see 4.5.1), 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, and is also able to duplicate a handle to the `NIL:` (pseudo-)device. It returns `ZERO` if no copy can be made (see section 9.2.11 for the `SetConsoleTask()` function):

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

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 with the above algorithm. An alternative algorithm that, however, only works if the source file is opened in shared mode (see table 10 of section 4.4.1) is to first obtain the file name through `NameFromFH()` (see section 6.3.2) and then to re-open another file from the same name.

### 4.8.4 An FSkip() Implementation

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

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

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

```
  if (fh->fh_Pos >= 0 && fh->fh_End > 0) {
    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 as the work is done.

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

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

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

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

### 4.8.5 An FGet() Implementation

While the FRead() function already provides a buffered read function, it is not very efficient prior release 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);
}
```

It reads size bytes from the file fh into the buffer buf, and returns the number of bytes read.

As seen from this implementation, the function attempts to satisfy the read if a partial buffer is present. If not, the above implementation runs into the operating system function. As for the FSkip() implementation presented in section 4.8.4, the file handle requires some preparation by a dummy UnGetC(), see there.

## 4.9  Formatted Output

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

### 4.9.1  Print Formatted String using C-Syntax to a File

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

```
count = VFPrintf(fh, fmt, argv) /* 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. In particular, format strings indicating integer arguments such as `%d` and `%u` assume 16bit integers, independent of the integer model of the compiler. On compilers working with a 32bit integer models, the format modifier `l` must be used, e.g. `%ld` for signed and `%lu` for unsigned integers.

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

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

### 4.9.2  BCPL Style Formatted Print to a File

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

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

```
count = VFWritef(fh, fmt, argv) /* 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 syntax of the BCPL language. The following format identifiers are supported:

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

%Tx   Writes a `NUL` terminated string left justified in a field whose width is given by the character x. The length indicator is always a single character; a digit from 0 to 9 indicates the field widths from 0 to 9 directly. Characters A to Z indicate field widths from 10 onwards. 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 that %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 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 BPCL *Global Vector*, it there takes BSTRs instead of regular C strings for the format string and the arguments of the %S and %T formats.

### 4.9.3  Setting the Console Buffer Mode

The `SetMode()` function sets the behaviour of a 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 an input, or provides each individual key as input to programs, or provides a line buffer with the exception of some special control keys that are transmitted immediately. Section 11.3 contains further details on the console.

```
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 the system, the graphical console of the `CON:` and `RAW:` device, and the serial console corresponding to the `AUX:` device. All three devices are, actually, implemented by the CON-Handler, while the AUX-Handler is just a simple wedge to the first for historical reasons.

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

**Table 13: 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 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 with 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, but rather transmitted directly. If echoing is desired, it needs to be performed manually by the application. This mode corresponds to the `RAW:` device which is nothing but a console operating in this mode by default.

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, key-combinations of the up- and down cursor keys and the TAB key are reported immediately to the caller through control sequences. These sequences are documented in section 11.3.3. The Amiga Shell uses this mode to offer a history and provides through 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 additional 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 0, in case of failure, it 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:

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

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

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

## 4.10    Record Locking

While locks, see chapter 5 control access to a file system 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 record lock on a region that overlaps with the locked region. Record locks therefore require the locking processes to follow the same locking protocol.

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

### 4.10.1    Locking a Region of a File

The `LockRecord` function locks a single region of a file, potentially waiting for a timeout for the region to become available.

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

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 the byte size `length`. The `mode` shall be taken from the constants defined in `dos/record.h`:

**Table 14: 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 failure, exclusivively locked regions cannot overlap with shared locked regions or with each other.

For the `REC_EXCLUSIVE` and `REC_SHARED` modes, the `timeout` value provides a time limit in ticks, i.e. $1/50^{th}$ of a second, after which an attempt to obtain a 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 with `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 and sets `IoErr()` to 0.

### 4.10.2    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.

`rec_Mode` specifies the type of the lock that is to be obtained. It shall be one of the modes listed in table 14; the modes are all defined in `dos/record.h`.

The `timeout` specifies how long each of the attempts to obtain a lock is supposed to wait for a record to become available if a non-immediate record lock is requested. The `timeout` is applied to each of the records in the `RecordLock` array sequentially until either all records could be locked, or until locking one of the records fail. In such a case, the call 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 cannot be locked, the function returns 0 and sets `IoErr()` to an error code. On success, the function returns a non-zero result and sets `IoErr()` to 0.

Unlike what the function prototype suggests, this function is *not atomic*. Instead, 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.

### 4.10.3 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 `LockRegion()` or `LockRegions()`. 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 and sets `IoErr()` to 0.

### 4.10.4   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 4.10.2.

   The function calls `UnLockRecord()` in a loop, and is therefore *not atomic*. In case unlocking any of these records fails, the function returns 0 but attempts to unlock also any remaining records in the array. On success, it returns a non-zero result code and sets `IoErr()` to 0. Unfortunately, the function does not set `IoErr()` consistently in case of failure as the error code is not saved on a failed unlock.

# Chapter 5

# 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, or in case of files, it can no longer be altered either. Depending on the file system, locks may also prevent other forms of changes of the object as well.

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 locks on an exclusively locked object can exist. An attempt to lock an exclusively locked object results in failure with the error code ERROR_OBJECT_IN_USE, and attempting to exclusively lock an object that is already locked by a shared lock will also fail with the same error.

Unlike exclusive locks, multiple shared locks can be kept on the same object simultaneously, though once a shared lock has been obtained, any attempt to lock the same object exclusively fails.

One particular use case of locks is to serve as an identifier of a particular directory or file on a file system. Since paths are limited to 255 characters, see 4.3, locks are the preferred method of indicating a position within a file system. Even though paths are length limited, there is no restriction on the depth within the directory structure of a file system. The ZERO lock identifies the boot volume, also known as SYS:, see also section 4.3.1.3.

Locks are also the building stone of files; every file is typically associated with a lock on the corresponding object, even if the file system does not expose this lock to the caller.

As long as at least a single lock is held of an object on a particular volume, the file system will keep the volume within the device list of the *dos.library*, see chapter 7. This has, for example, the consequence that the Workbench will continue to show an icon representing the volume in its window.

Locks cannot be held on links (see chapter 6.4) because the link — regardless of whether it is a soft or a hard link — is resolved as soon as the lock is acquired, and thus only the link target and not the link itself will be locked.

## 5.1 Obtaining and Releasing Locks

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

### 5.1.1 Obtaining a Lock from a Path

The Lock() function obtains a lock on an object given a path to the object. The path can be either absolute, or relative (see section 4.3) to the current directory of the calling process.

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

BPTR Lock(STRPTR, LONG)
```

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

**Table 15: Lock Access Modes**

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

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

The return code `lock` identifies the lock. It is non-`ZERO` (see 2.4) on success, or `ZERO` on failure. In either case, `IoErr()` is set to 0 indicating success, or an error code on failure.

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

### 5.1.2   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 it returns a shared lock if successful.

```
lock = DupLock( lock )
D0             D1

BPTR DupLock(BPTR)
```

This function copies the (shared) lock passed in as `lock` and returns a copy of it in `lock`. It is not possible to copy a lock by any other means (e.g. copying the `FileLock` structure *does not* work).

In case of error, `DupLock()` returns `ZERO`, and then `IoErr()` returns an error code identifying the error. On success, `IoErr()` is reset. It is not possible to copy an exclusive lock.

### 5.1.3   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 directories, this is the parent directory, for files, this is the directory containing the file.

```
newlock = ParentDir( lock )
D0                    D1

BPTR ParentDir(BPTR)
```

The `lock` argument identifies the object whose parent is to be determined, this argument *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 check 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 a second shared lock.

### 5.1.4   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, the directory given by the last component of the path (see section 4.3) 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`[1]. 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 0 in case the function succeeds.

Note that not all file systems support directories, i.e. flat file systems (see section 4.3.4) do not.

### 5.1.5   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 or modify 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. It does, however, set `IoErr()` if `lock` is not `ZERO`, and thus potential problems of a file system releasing a lock may be identified.

### 5.1.6   Changing the Type of a Lock

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

```
success = ChangeMode(type, object, newmode) /* since V36 */
D0                   D1     D2      D3


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

---

[1]The information in [10] that in such a case a conflicting file or directory is deleted is not correct.

This function changes the access mode of `object` whose type is identified by `type` to the access mode `newmode`. The relation between `type` and the nature of the object shall be as in table 16, where the types are defined in `dos/dos.h`:

**Table 16: 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 15, i.e. `SHARED_LOCK` to make either the file or the lock accessible for shared access, and `EXCLUSIVE_LOCK` for exclusive access.

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

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

### 5.1.7   Comparing two Locks

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

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

LONG SameLock(BPTR, BPTR)
```

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

**Table 17: Lock Comparison Return Code**

| Return Code | Description |
|-------------|-------------|
| LOCK_SAME | Both locks are on the same object |
| LOCK_SAME_VOLUME | Locks are on different objects, but on the same volume |
| LOCK_DIFFERENT | Locks are on different volumes |

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

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 savely interpreted, see also section 5.2.5. AmigaDOS version 36 introduced a new interface to request comparing two locks from the file system, see section 12.2.7. However, even the current *dos.library* falls back to the above algorithm in case this interface is not implemented by the target file system.

### 5.1.8   Compare two Locks for their Device

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

```
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 whether the locks refer to file systems on the same or different physical volumes.

   This function returns a non-zero result if the resonsible file systems operate on 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.

## 5.2 Locks and Files

Each file handle is associated to a lock to the file that has been opened. The type of the lock depends on the access mode the file has been opened with, table 18 for how lock types and access modes relate.

**Table 18: Lock and File Access Modes**

| Access Mode | Lock Type |
|---|---|
| MODE_OLDFILE | SHARED_LOCK |
| MODE_READWRITE | SHARED_LOCK |
| MODE_NEWFILE | EXCLUSIVE_LOCK |

The association of MODE_READWRITE to SHARED_LOCK is unfortunate, and due to a defect in the RAM-Handler implementation in AmigaDOS 2.0 which was then later copied into the Fast File System implementation. In AmigaDOS versions 34 and below, this mode was associated to an EXCLUSIVE_LOCK. For version 36 and above, exclusive access to a file without deleting its contents can, however, be established through the OpenFromLock() function passing in an exclusive lock to the function as argument.

### 5.2.1 Duplicate the Implicit Lock of a File

The DupLockFromFH() function performs a copy of a lock implicit to a file handle of an opened file. For this to succeed, the file must be opened in the mode 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 returns a copy of the lock the file handle fh is based on and returns it in lock, that is, returned lock is a lock on the file described by the handle fh. This file handle remains valid and usable after this call. In case of failure, ZERO is returned. In either case, IoErr() is set to 0 in case of success, or an error code on failure.

### 5.2.2 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 DupLockFromFile(), and then calling ParentDir() on it, except that this function also applies to 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 0 in case of success or to an error code from dos/dos.h in case of failure.

### 5.2.3 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. 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 0 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)
{
  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, a feature every AmigaDOS file system supports, see also section 12.1.1.

### 5.2.4 Get 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 12.7.3.

```
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 querried; its only purpose is to identify it. The function fills out an `InfoData` structure that shall be aligned to long-word boundaries.

This structure 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 19: 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 writeable |

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 12.7.7.

A volume gets 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 and then 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 will remain ID_VALIDATING.

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

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 version 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 file system on the disk. As it is dependent on the file system how many blocks it needs 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 adminstration. 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 that generated the lock argument can identify the disk structure and claims responsibility for it. Unlike what the name suggests, it is *not* a general identifier of the type file system itself and shall not be used to identify a paritcular file system. For legacy reasons, the various flavours 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 20: 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 knows 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 |
| 'RAW\0' | Not a file system, but returned by the CON-Handler |
| All others | The first long word of the first block of the medium |

As mentioned above ID_DOS_DISK is the id_DiskType file systems should return in case they recognize 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 29 of section 7.1.3.

> *Not the DosType* While mountlists include a DOSTYPE field that identifies the file system uniquely, the id_DiskType member *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.

ID_NOT_REALLY_DOS and ID_UNREADABLE_DISK both indicate disks the file system cannot make use of. The first because the logical structure of the disk content cannot be interpreted, and the second because 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 these two disk types, see section 12.7.3.

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 12.7.3.

'BUSY' is a four-character constant that is not documented in dos/dos.h, but returned whenever a file system has been inibited, i.e. its access to the physical layer has been stopped. Thus, any attempt to access this file system is currently suspended, probably because some program attempts to operate on the medium on a lower level. Disk editors or disk salvage programs will typically make use of this practise to avoid file systems from touching the medium while they work on it. The Inhibit() function of section 7.7.4 blocks file system access to the physical layer and thus causes this 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 11.3. As they do not (in general) hand out locks, the `Info()` function will usually not return these two types, but direct handler communication with a packet type of `ACTION_DISK_INFO` can. The CON-Handler populates the `InfoData` structure in a different way than file systems do, this is documented in section 12.8.3.

All other types are returned in case the file system cannot interpret the disk structure, and are then copied from the first 4 bytes of the medium or partition into `id_DiskType`. In case these bytes are all 0, it is changed to `ID_NOT_REALLY_DOS`. 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 medium containing the characters 'KICK'. 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. For this structure, see chapter 7.

`id_InUse` counts the number of locks and files currently open on the medium identified by `lock`.

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

### 5.2.5 The struct FileLock

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

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

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

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

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

The `fl_Link` member has no practical value for users; the file system can use it to keep multiple links on object on the same volume in a list. The Fast File System creates a singly linked list of all locks of a medium it is responsible for once the medium is ejected, and queues them in the `dol_LockList` element of the `DosList` structure such that a FFS incarnation responsible for the same medium, but potentially inserted into a different drive can pick them up from there, see also chapter 7.

The `fl_Key` member can be used by the file system to identify the object that has been locked. It may not necessarily be an integer, but can be any data type, potentially a pointer to some internal management object. It shall not be interpreted in any particular way as its interpretation 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 11.7.4, 11.7.5 and 11.7.6 for the structure of these blocks.

The `fl_Access` member 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` member points to the message port of the file system for processing requests to the lock. Any activity on the lock goes through this port. In case the medium 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 7 provides further information on this list and its entries.

# Chapter 6

# 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 4.3.4.

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.

## 6.1   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*. The `ExAll()` function is an advanced interface that allows to read multiple directory entries at once, potentially filter them by name, and thus minimize the call overhead.

> *May go away while you look!* As AmigaDOS is a multitasking operating system, the contents may change anytime while scanning a directory; 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 this directory goes away, it does *not* prevent other processes to add or remove objects to this directory, so beware.

While `ExAll()` seems to provide an advantage by reading multiple directory entries in one go, the AmigaOS ROM file system — the FFS — does usually not profit from this feature, at least not unless a directory cache is used. The latter has, however, other drawbacks and should be avoided for different reasons, see section 11.7. 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 issues in the past. 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 meaning of the members of this structure are as follows:

fib_DiskKey is a file system internal identifier of the object. It shall not be used, and programs shall not make any assumptions on its meaning. The Fast File System stores here the disk block (the "key") 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 which 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 21: 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 6.1.3 and ExAll() in section 6.1.4 can when iterating over the contents of a directory. While the FFS implements hard links through separate types, other file systems cannot identify hard links at all as they 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 systems. Thus, one cannot depend on hard links being easily identifyable. 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 types larger than zero indicate directories, and all other types smaller than zero indicate files. Section 6.4 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 flavour of the FFS, see table 29 in section 7.1.3.

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

### Table 22: 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 Delete(), but also erasing it though creating a new file on top through the Open() mode MODE_NEWFILE. Attempting to delete a protected file system object returns the error code ERROR_DELETE_PROTECTED.

The FIBB_EXECUTE flag is only interpreted by the Shell (see chapter 13). If the bit is set, the Shell refuses to load the file as 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 upon 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 subdirectories are *not* reflected in the parent directory. In other words, this bit is not recursively updated. 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 modifyable 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 allow to generate pure code. Please check the corresponding compiler manual.

The Shell command Resident loads pure binaries into RAM for future usage and thus reduces loading time. Unfortunately, nothing in the shell actually *checks* whether a binary is, actually pure, and its segment has actually 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 or an ARexx script. If this flag is set, and the script is given as command to the Shell, it will forward this file to a suitable script interpreter, such as ARexx

or `Execute`. AmigaDOS versions 45 and up also allow additional script interpreters, to be indicated in the first line of a shell script, see section 13.1.8 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, and the shell loads the file as executable binary, and the `FIBB_PURE` bit is also set, the file is kept in RAM and stays there for future execution. 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 target. Separate protection bits for links have only a questionable value as they cannot be easily checked without iterating through a directory — remember that 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 access, see also section 6.4.

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

The `fib_Size` member indicates the size of the file in bytes. It should have probably be defined as an unsigned type, 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` member 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 11.7.8, and not only 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 flavours of the same file system. For some file systems, it need not to be consistent even for identical file contents.

The `fib_Date` member indicates when the file was changed last; depending on the file system, the date may also indicate when the last modification was made for a directory, such as creating or deleting a file or directory within. For the Fast File System modification of file or directory metadata, such as file comments or protection bits are not reflected here. Which operations exactly trigger a change of a directory is file system dependent. The `DateStamp` structure is specified in section 3.1.1.

As for protection bits, the AmigaDOS core 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` member contains a `NUL` terminated string to a comment on the file. Not all file systems support comments. The comment has no particular meaning, it is only shown by some Shell commands or utilities and can be set by the user. 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` 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. The AmigaDOS ROM file systems do not support these fields, and no provision is made to moderate access to a particular file according to an owner or its group. The two concepts are alien to AmigaDOS itself. These fields were introduced in AmigaDOS version 43, though remain unsupported by the ROM file systems.

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

### 6.1.1 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 6.1 in more detail. As the `Lock()` function always resolves links, this function will not return a `fib_DirEntryType` that corresponds to a link.

The function returns non-zero in case of success, and 0 for failure. In either case, `IoErr()` is filled, by 0 on success, on an error code on failure.

> *Keep it Aligned!* As with most BCPL structures, the `FileInfoBlock` shall be aligned to a long-word boundary. For that reason, it should be allocated from the heap. Section 2.4 provides some additional hints on how to allocate such structures.

### 6.1.2 Retrieving Information from a File Handle

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

```
success = ExamineFH(fh, fib) /* 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 and thus its change can be changed any time, and thus the information returned by this function may not be fully up-to-date, see also the general information in section 6.1.

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 0 on success and to an error code otherwise.

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

### 6.1.3 Scanning through a Directory Step by Step

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

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

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

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

This call returns information on the subsequent entry of a directory identified by `lock` and deposits this information in the `FileInfoBlock` described in 6.1. The `lock` shall be a lock on a directory, in particular, and shall have been prepared for a directory scan by `Examine()` first.

Unlike `Examine()` and `ExamineFH()`, this function is, depending on the file system, able 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 6.1 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 0 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()` is set to `ERROR_NO_MORE_ENTRIES`.

> *Same Lock, Same FIB* To iterate through a directory, a lock to the same directory as passed into `Examine()` shall be used. Actually, the same lock should be used, and the same `FileInfoBlock` should be used. As important state information is associated to the lock and `FileInfoBlock`, `UnLock()`ing the original lock and obtaining a new lock on the same directory looses this information; using a different `FileInfoBlock` also looses this state information, requiring the file system to rebuild this state information, which is not only complex, but also slows down scanning the directory. In particular, you shall *not* use the same `FileInfoBlock` you used for scanning one directory for scanning a second, different directory as this can confuse the file system. Also, as for `Examine()`, the `FileInfoBlock` shall be aligned to a long-word boundary. Similarly, the same lock should not be used by more than one directory scan at the same time. This is particularly important for locks returned by `GetDeviceProc()` for assigns. As the same lock will be provided to each caller, the returned lock should be duplicated first by `DupLock()` before initiating a directory scan on the assign target.

The following simple code lists the contents of the directory represented by a lock on the console. It uses the `D_S` macro of section 2.4 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;
}
```

### 6.1.4 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. This buffer is filled by a linked list of `ExAllData` structures, see below for their layout. The `type` argument determines which elements of `ExAllData` is filled.

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 14.1.1 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 fit into the `buffer`. Note that this number may well be 0, which does not need to indicate termination of the scan. Callers shall instead check the return code of `ExAll()` to learn on whether scanning may continue or not.

`eac_LastKey` is a file system internal identifier of the current state of the directory scanner. This member shall not be interpreted nor modified in any way. 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 wild card pointed to by this member. This entry shall be either `NULL`, or a pre-parsed pattern as generated by `ParsePatternNoCase()`, see section 8.2.2 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 members are `NULL`, all entries match.

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

```
struct ExAllData {
   struct ExAllData *ed_Next;
   UBYTE  *ed_Name;
   LONG    ed_Type;
   ULONG   ed_Size;
   ULONG   ed_Prot;
   ULONG   ed_Days;
   ULONG   ed_Mins;
   ULONG   ed_Ticks;
   UBYTE  *ed_Comment;
   UWORD   ed_OwnerUID;
   UWORD   ed_OwnerGID;
};
```

The members 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 21 in section 6.1 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 22 in section 6.1 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 6.2.5 defines these elements more rigorously.

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

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

Which members of the `ExAllData` structure are filled in is selected by the `type` argument. It shall be selected according to table 23. The values are defined in `dos/exall.h`:

**Table 23: Type Values**

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

The return code `continue` of `ExAll()` is non-zero in case the directory contents was too large to fit into the supplied `buffer` completely. Then `IoErr()` will be set to 0. 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. Addiional 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 — actually, none delivered with AmigaOs — support `ED_OWNER`. If `continue` is `DOSFALSE` and `IoErr()` is `ERROR_BAD_NUMBER`, try to reduce `type` and call `ExAll()` again.

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

The following example code lists the directory contents through `ExAll()`; for demonstration, 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 historically issues. 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 within the version 36 of the *dos.library*. The FFS did not fill in `ed_Comment` correctly prior to version 39, and the FFS flavours 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 of the library.

There was no possibility to abort a running directory scan prior to version 39 of AmigaDOS which introduced `ExAllEnd()` (see section 6.1.5). 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 flavours of the FFS, see Table 29 in section 7.1.3, and the RAM-Handler profit noticably 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.

### 6.1.5  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 terminates due to an error or due to the end of the directory. That is, `ExAllEnd()` shall be called whenever 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 may proceed offline on a separate server. The arguments to `ExAllEnd()` shall be exactly those supplied to `ExAll()` which it is supposed to terminate. Note in particular that the `lock` shall be identical to the lock passed into `ExAll()`, and not just a lock to the same object.

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.

## 6.2 Modifying Directory Entries

While the functions in section 6.1 read directory entries, the functions listed in this section modify the directory and its entries, that is, they delete entries, or adjust metadata of directory entries.

### 6.2.1 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 6.1), and the object must not be locked (see section 5) or accessed through a file handle[2]. To be able to delete a directory, it 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 $0$ on success or an error code in case of failure.

### 6.2.2 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 position within the object to relocate, i.e. you cannot move a directory into itself.

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

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

### 6.2.3 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.

---

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

```
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 0 on success or to an error code otherwise.

### 6.2.4   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 22 of section 6.1. The `mask` value corresponds to what `Examine()` returns in the `FileInfoBlock` structure in `fib_Protection`, see also section 6.1. 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 0 on success or to an error code otherwise.

### 6.2.5   Set the Modification Date

The `SetFileDate()` function sets the modification date of an object of a file system. Despite its name, the function can also set the modification date of directories 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 is specified in section 3.1.1.

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 0 on success or to an error code otherwise.

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

Even though even AmigaDOS releases 34 and earlier supported setting the modification date of file system objects through the packet interface, see section 12.5.5, a convenient interface on the level of the *dos.library* did not exist.

### 6.2.6  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 seems to offer some multi-user capability, this is not the case. User and group ID are purely metadata that is returned by the functions discussed in section 6.1, they usually ignore them. AmigaDOS has no concept of the current user of a file system and thus cannot decide whether a user is privileged to access an object on a file system. In fact, all ROM based file systems delivered with AmigaDOS do not support setting the user or group ID.

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

BOOL SetOwner (STRPTR, LONG)
```

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

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

## 6.3  Working with Paths

The *dos.library* contains a couple of support functions that help working with paths, see also section 4.3. 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 4.3.3) as the path is never communicated to a file system. 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.

### 6.3.1  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.

### 6.3.2  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 identified by 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 even works if the file has been opened for exclusive access.

The returned path is not necessarily identical to the path through 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 not useful.

### 6.3.3  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 according to the rules explained in section 4.3; the colon identifies the root of the volume, and a leading slash the parent directory upon which the trailing component of the input path is removed.

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

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

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

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

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

### 6.3.4   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()`.

### 6.3.5   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 buffer 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 potential trailing slash. That is, if the last character of the input path of `PathPart()` would be a slash, then `PathPart()` would return a pointer to this slash, but `FilePart()` would advance beyond this slash. That is, the "file part" of a path that explicitly indicates a directory is empty, though the "path part" is the same path without the trailing slash.

### 6.3.6   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 top to bottom.

```
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. These return values are also valid even in case the found component was too large to fit into `buf` and had to be truncated.

This function does not provide error codes, 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 progresses. That is, if result code `newpos` is not negative, it should be passed back into this function as `oldpos` for a subsequent scan which then extracts the next component of the path. The intended users of this function are file systems for locating objects in the file system tree. For most (if not all) AmigaDOS file systems the sparator 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.

## 6.4   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 changed using the path of one location, the changes are reflected in another location. Put differently, creating a link is like copying an object except that copy and original remain always identical. The storage for the file 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 directory, 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 at 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 becomes zero.

Soft links work differently and can also be established between different 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 *dos.library*.

The *dos.library* supports soft links through the functions listed in Table 24:

**Table 24: Softlink aware functions**

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

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, 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 6.4.2 contains 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 then, obviously, 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; these packets are specified in chapter 12. If the handler addressed by the packet

---

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

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 softlink via the *ReadLink()* function which sends a packet of type `ACTION_READ_LINK` back to the handler, see section 12.4.2. The handler then creates from the original path and the link target stored in the soft link an updated path. 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 softlink is provided in section 12.4.2. 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.

This process 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.

Finally, the RAM-Handler supports a special type of hard link that goes across volumes. These *external links* copy the link target on a read-access into the RAM disk, i.e. the RAM-Handler implements a *copy on access*. This feature is used for the `ENV:` assign containing all active system settings. 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 6.1 identifies links through the `fib_DirEntryType` member. As seen from table 21, 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 21 also provides the `fib_DirEntryType` for soft links, namely `ST_SOFTLINK`. As the target of a soft link may not under control of the file system 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.

### 6.4.1 Creating Links

The `MakeLink()` function creates a hard link, soft link or external link to an existing 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 25, defined in `dos/dos.h`:

**Table 25:** `Link Types`

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

If `soft` is `LINK_HARD`, `dest` is a lock represented by `BPTR`. For most file systems, `dest` shall be on the same volume as the one identified by the path in `name`. The 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 target object of an external object 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 accessing the link contents. Thus, the link becomes a mirror of the link destination 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 6.4.2. 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.

This 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 0 on success.

### 6.4.2 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 in the original path. In most cases, namely for the function listed in table 24, the *dos.library* calls this function itself internally and its user does not need to worry about soft links.

If, however, a *dos.library* function or a packet in direct communication with a handler (see chapter 11) returns the error `ERROR_IS_SOFT_LINK`, then this function helps to resolve the soft link in the path. The access is then typically retried with the path constructed by this function. Note well that this path can be that of yet another soft link, requiring recursive resolution of the link. To avoid endless recursion, this loop should be aborted after a maximum number of attempts, and should 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

BOOL 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 somewhere in this path. 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 7.2.1 for this structure and the function.

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

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

Section 12.4.2 provides further details on the file system side of the implementation of this function, in particular how it combines the `path` with the target of the soft link.

The following source demonstrates how to use `ReadLink()` to implement a `Rename()` function that accepts sources containing 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;
  struct DevProc *dp = NULL;
  LONG levels       = MAX_LINKS;
  LONG error        = 0;
  LONG success;

  while (!(success = Rename(srcpath,dstpath)) &&
          IoErr() == ERROR_IS_SOFT_LINK) {
    UBYTE *newbuf;
    /* If too many levels passed, fail */
    if (--levels < 0)
      break;
    /* Allocate a target buffer for the new name */
    if (!(newbuf = AllocMem(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))) {
          srcpath = newbuf;
          newbuf  = buf;
          buf     = srcpath;
          error   = 0;
          break;
        } else {
          error = IoErr(); /* get the error */
          /* 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
          ** right type of 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)
    FreeMem(newbuf,BUFSIZE);
  /* Fixup or reset the error          */
  SetIoErr(error);
  /* Abort if link resolution failed   */
  if (!success)
    break;
}

/* Release the temporary/last buffer */
if (buf)
  FreeMem(buf,BUFSIZE);

return success;
}
```

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

## 6.5   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 onwards. However, please be aware that not all file systems are able to support 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 application of notification requests is the IPrefs program which uses them to monitor changes of the preferences files. If it detects any changes of the preferences, it reloads the contents of the files and re-installs the preferences into the components it serves, most importantly the *intuition.library*.

### 6.5.1   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 placed in 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 request 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 through exec 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 assignments are made to a monitored directory, nor if the volume containing the object is removed, reinserted or access to it is inhibited (see section 7.7.4).

`nr_FullName` is initialized by the file system and shall be left alone by the caller. The *dos.library* uses it to store the full path of the object to monitor. This element is supposed to be used by the file system only, see section 12.6.1.

`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` identifies the activity that is performed when a change on the monitored object has been detected by a file system. Currently, the following flags are defined in `dos/notify.h`:

**Table 26: 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.

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 received. This allows applications to roll both the initial action and the response of the notification into a single function — for example, for loading an initial version or re-reading a preferences file.

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 informed, 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; its elements are 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 12.6.1. 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 notification.

`nm_Code` is always set to `NOTIFY_CODE`, again defined in `dos/notify.h`. This again may be used to identify notifications.

`nm_NReq` is a pointer to the `NotifyRequest` through which this message was triggered. This may allow 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 or interpreted by the caller or the client.

By design, the `NotifyMessage` reassembles the layout of an `IntuiMessage` and thus allows reusing an IDCMP port (see [7]) of an intuition window for receiving messages — as long as it can be ensured that the port remains available until the notification request is canceled.

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

### 6.5.2 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()`[4]. 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.

---

[4]The information in [1] on this is incorrect, `EndNotify()` is *not* safe to call on requests that failed to start.

# Chapter 7

# 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 section 11.1). The library, however, also needs to obtain these ports somehow from a device name as included in absolute paths. For example, if the file "`S:Startup-Sequence`" is to be opened, the relation between the "`S`" assign and the handler responsible for implementing opening and reading from this file must be established.

Each volume, assign, handler or file system is therefore 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 15.3) named the *device list*. If the library requires to contact a handler through its device name, it calls internally the `GetDeviceProc()` function which 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 7.1. 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, 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 on this list, which goes through the functions listed in section 7.6. Some assigns, namely those listed in table 7 of section 4.3.1.3, 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 on demand 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 [8].

Some special names such as `NIL` or `*` from table 6 in 4.3.1.1 are hard-coded into `GetDeviceProc()` and are not represented as 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 quite central to the *dos.library* and establishes the link between the device name as it appears on an absolute path upfront the colon (":") and the port behind it responsible for implementing functions on such a device. 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 throught 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`, again just with different names at the same byte offsets, to be used only for handlers; e.g. `dvi_Handler` and `dol_handler.dol_Handler` access the same elements. Both, the `DevInfo` and the `DeviceList` are defined in `dos/dosextens.h`.

Finally, `dos/filehandler.h` contains a third structure, named `DeviceNode`, which is just another representation of a `DosList`, also only to be used for file systems, and thus equivalent to a `DevInfo`. 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].

_____

[1]The reason for this confusion lies likely again in the history of AmigaDOS actually being a port of Tripos, written in BCPL. This

The elements of a `DosList` now 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 15.3. However, this list should not be walked manually, but instead `FindDosEntry()` should be used for iterating through this list.

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

**Table 27: *DosList* Entry Types**

| dol_Type | Description |
|---|---|
| DLT_DEVICE | A file system or handler, see 4.3.1.1 |
| DLT_DIRECTORY | A regular or multi-assign, see 4.3.1.3 |
| DLT_VOLUME | A volume, see 4.3.1.2 |
| DLT_LATE | A late binding assign, see 4.3.1.3 |
| DLT_NONBINDING | A non-binding assign, see 4.3.1.3 |
| DLT_PRIVATE | Used internally for resolving late binding assigns |

`dol_Task` is the `MsgPort` of the handler to contact for the particular handler, assign or volume. If the `DosList` represents a handler or file system, this element can be `NULL` if the handler is not yet started, or a new handler process is supposed to be started for each file opened. This is, for example, the case for the console which requires a process for each window it handles. It is up to the handler to deposit a port here, and this is typical for file systems that handle all activies on the same physical device through a single process.

Volumes keep here the `MsgPort` of the file system that operates the volume, but it to `NULL` in case the volume goes away, e.g. is ejected, though locks on the volume remained and thus the volume itself remains known to AmigaDOS.

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 directory bound to. All additional ports can be reached through the locks that are are part of the `dol_List` discussed below. For late binding assigns this member is initially `NULL`, but will be filled in as soon as the assign in bound to a particular directory, and then becomes the pointer to the port of the handler managing the volume the assign is located on. Finally, for non-binding assigns this member always stays `NULL`.

`dol_Lock` is only used for assigns, and only if it is bound to a particular directory. It is then the lock 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 member stays `ZERO`.

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

The members 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` fields of the mountlist, see section 7.1.1.

---

language represent structures through long word indices relative to a base index in manifests, and thus does not offer structures as known from C.

`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 28 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 `BYTE`s. 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 commumicate a configuration to the handler. For example, the CON-Handler uses this element to distinguish whether it is used as `CON` or `RAW` device. 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 11.2.1 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 file system, though section 7.3.6 provides additional hints and a recommended algorithm for interpreting this element. One way to set it is through the `STARTUP` in the mountlist, see 7.1.2 for details.

`dol_SegList` is a BPTR to the chained segment list of the handler if it is loaded, see chapter 10 for additional details how binaries are represented in memory. For disk-based handlers, this member 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 field 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, otherwise it first scans the `FileSystem.resource` for an entry whose `DOSTYPE` matches. If nothing is found there, loading the handler remains to the first process requiring it. Details depend on nature of the handler or file system and are provided in section 11.2.1.

`dol_GlobVec` identifies the nature of the handler as AmigaDOS (still) supports BPCL and C/assembler handlers and defines how access to the device list is secured during handler loading and startup. BCPL handlers use a somewhat more complex loading and linking mechanism as the 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 11.2.1 and 10.5.3. Another aspect of the startup process is how the device list is protected from conflicting accesses through multiple processes. Two types of access protection are possible: Exclusive access to the list, or shared access to the list. Exclusive access protects the device list from any changes while the handler is loaded and until handler startup completed. This prevents any other modification to the list, but also from read access through any other process to the list. Shared access allows read accesses to the list while preventing exclusive access to it.

The value in `dol_GlobVec` corresponds to the `GlobVec` entry in the mountlist. It shall be one of the values in table 28:

**Table 28: 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 |
| $>0$ | BPCL 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. This vector contains all global objects and all globally reachable functions of a BCPL program, including functions of the *dos.library*. The values $0$ and $-3$ fill this vector

---

[2]The information in [10] is not accurate, the stack size is dependent on the GLOBVEC.

with the system functions first, and then use the BPCL binding mechanism to extend or override entries in this vector with the values found in the loaded code. Any values $> 0$ require a BPTR to a custom vector which is used instead for initializing the handler. This startup mechanism has never been used in AmigaDOS and is not quite practical as this vector needs to be communicated into the *dos.library* somehow. For new code, BCPL linkage and binding should not be used anymore.

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

`dol_VolumeDate` is the creation date of the volume. It is a `DateStamp` structure that is specified in section 3.1.1. 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` member of the `FileLock` structure, see also section 5.2.5. 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 do 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[3], see section 5.2.4.

Members of the `dol_assign` structure are used for all other types, i.e. 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 member is used only on the first attempt to access the assign at which `dol_Lock` is populated, and this element is released with `FreeVec()` and then 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 subsquent directories, represented as a `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 7.6.4 and 7.6.5.

---

[3]This is not accurately stated in [10].

## 7.1    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 there. The latter two structures will be introduced in the following.

### 7.1.1    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[4]. 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. Useful values are between 0 and 19.

The `GLOBVEC` keyword sets `dol_GlobVec` element, and by that also the type of the handler. Table 28 in 7 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 if the argument is enclosed in double quotes, a string. In the latter case, `dol_Startup` is set to a BPTR to a BSTR which is, to ease handler implementation, also `NUL` terminated. The quotes become part of the string, and the handler has to remove them when interpreting this startup string. 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.

### 7.1.2    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;
};
```

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 11.4.2 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.

---

[4]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.

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 device 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 the following section.

### 7.1.3 Keywords controlling the Environment Vector

The `fssm_Environ` element of the `FileSysStartupMsg` is a BPTR to another structure that describes, amongst others, the layout (or "geometry") of a file system on a disk; 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 32 and below, at least all elements up to `de_NumBuffers` shall be present, and thus `de_TableSize` is at least 11. AmigaDOS 33 added `de_BufMemType`, and AmigaDOS version 34 `de_MaxTransfer` and `de_Mask`. Version 36 added the remaining fields.

The keyword `SECTORSIZE` and `BLOCKSIZE` 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 blocks, or 1024 for 4096 byte blocks. However, not all file systems support all block

sizes, and some only support the default value of 128, i.e. 512 byte blocks. 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 meaning for this member is to interpret it as the number of read-write heads of a magnetic disk drive. However, as the exec device `trackdisk` interface for magnetic storage media does not actually allow access to such low-level features that are instead abstracted away by the device, 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 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 longical storage block. Not all file systems support values different from 1 here; the FFS does, and always reads and writes data in units of `de_SectorsPerBlock` times `de_SizeBlock` of long-words. For the exec `trackdisk` interface, the byte size and offset are the only values that matters; however, the "direct SCSI" transfer the FFS version 45 and above offer addresses the disk in terms of physical sectors, and then `de_SizeBlock` defines the size of a sector in long-words as addressed by SCSI commands, whereas `de_SectorsPerBlock` is the number of physical sectors the FFS reads for one logical block. For many purposes, only the product of the sector size as defined by `de_SizeBlock` and the block size in sectors defined by this keyword matter, though both enter in different places when relating the block, i.e. the so-called *key* the FFS uses to adminstrate, to the byte offset on the medium where information is found. Section 11.7.3 provides the details. In general, `de_BlocksPerTrack` is measured in physical sectors (despite its confusing name) as defined by `de_SizeBlock`, and not in logical blocks as computed from `de_SectorsPerBlock`.

The `SECTORSPERTRACK` and `BLOCKSPERTRACK` keywords set both the `de_BlocksPerTrack` element of the `DosEnvec` structure. The first name is actually most 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 historical 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, and the location where the root block of an FFS partition is located, see section 11.7.4. The 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*. As the FFS reserves the block number 0 (actually, the key 0, see 11.7.3) to indicate a free or non-existing linkage, the very first block of a partition cannot be made use of anyhow, and thus a positive number is required here. By convention, the value is 2, which is 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. However, current FFS versions completely ignore it and no traces of its use are found in the current source. Historically, the BPCL implementation of the OFS shipped with AmigaDOS 34 and before used this value instead to compute the number of blocks the file system could pre-read from the disk and store in its internal buffers without a penalty. 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 value alone as its interpretation seem to have changed over time.

The `INTERLEAVE` keyword defines the lower 16 bits of the `de_Interleave` element. Tripos and AmigaDOS versions prior release 45 reserved the entire long-word to define the interleave factor of the disk, which is the difference between two sector addresses the file system is supposed to reserve for subsequent storage. This interleave factor historically helped to speed up data rates of slow harddisk interfaces; as data is transferred, 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 reappear. Setting an interleave

factor larger than 0 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 0, regardless of what is set here. The upper 16 bits of `de_Interleave` suit now a different purpose and are set by separate keywords in the mountlist. Thus, for all practical purposes, this keyword serves no purpose, and even the legacy BPCL 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. $de\_LowCyl \times de\_Surfaces \times de\_BlocksPerTrack + de\_Reserved \times de\_SectorsPerBlock$ is the first physical sector number on a disk that can carry payload data of the file system.

The `HIGHCYL` keyword sets the `de_HighCyl` element; it defines the (inclusive) upper end of the partition in units of cylinders. That is, $(de\_HighCyl + 1) \times de\_Surfaces \times de\_BlocksPerTrack - 1$ is the last sector of the partition or medium the file system can allocate for payload 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 7.7.1. Each buffer holds data of one (logical) block, though a small adminstrative 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 adminstrating the cache has a negative impact from a certain amount of buffers onwards. For the FFS, larger blocks (rather than a larger number of buffers) are recommended as more data and administrational overhead 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 to. While exec device drivers should be able read and write data to any memory type, several legacy drivers are not able reach every memory type; this is often due to limitations of the hardware such that Zorro-II hardware cannot reach 32-bit memory on CPU expansions. 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 memory 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 onwards aligns buffers it allocates to multiples of 16 bytes, suitable for a mask whose last digit is 0. Requiring even more least siginficant bits to 0 thus makes no sense. FFS versions 45 and below could only ensure that the last two significant bits of its own buffer memory are 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 BUFFMEMTYPE memory, and copy the data manually (i.e. by the CPU) the buffer content 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. enforces 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, BUFFMEMTYPE 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 work around 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 sectors 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. Obviously, it makes little sense to set this keyword in a mountlist that is interpreted after the system had already booted. However, autobooting 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 flavour 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. The Format command uses the value deposited here to initialize a partition for a mounted file system, whereas the file system determines its type from the boot block, i.e. the first block of a partition, see section 11.7.2. Table 29 lists some AmigaDOS file systems:

**Table 29: Fast File System Flavours**

| FFS Flavour | 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 compatible to FFS |
| `'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 reimplementation, the FFS.

The OFS variants embed additional administration information (see section 11.7.12) 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 requires an additional copy. The FFS variant immediately below addressed this issue; due to its reimplementation 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 character in the extended ISO-Latin-1 set (i.e. printable characters outside the ASCII range) correctly (see also section 11.7.4 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 flavours which are, despite the layout of data blocks, otherwise identical. All types from that point on in table 29 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 the directory, 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 11.7.11 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 11.7.10.

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 flavours 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 MBR type partition table, as found on floppy disks. They are unsuitable for harddisk and thumbdrive partitions as both typically include a master boot record (MBR).

The second type indicates FAT 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, adpoting 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 next 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 mountlists. 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 obviously 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 onwards, and by the AUX-Handler version 47 onwards.

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. In the latter case, the argument to `CONTROL` shall be enclosed in double quotes, and the `Mount` command then converts this string, including the quotes, to a BSTR and installs a BPTR to this string in `de_Control`. 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 string. The Port-Handler, see section 11.4.2, and AUX-Handler (see section 11.3.4) interpret this parameter to define the parameters of a serial connection.

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 [8]. The implied value for the floppies is thus 2, and zero 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. 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, though the former also requires the `'FAT\0'` dos type.

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 through 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 anchient 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` field; 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` field 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 the line `ENABLENSD = 0` in the mountlist should it 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 ont. 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 enforce that a particular version of the file system is loaded from disk. This keyword does neither impact the `DosEnvec` structure.

## 7.2   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 functions search the device list, check whether the handler is already loaded or load it if necessary, then check whether the handler is already running, and if not, launch an instance of it. If multi-assigns are involved, it can become necessary to contact multiple file systems to resolve the task and thus to iterate through multiple potential file systems to find the right one.

### 7.2.1 Iterate through Devices Matching a Path

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

```
devproc = GetDeviceProc(name, devproc) /* 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 reponsible for it.

> *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 release 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 that potentially contains the path in `name`. If the file system object cannot be reached through this file system relative to `dvp_Lock`, 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.

`dvp_Lock` is a directory represented as a lock relative to which the path in `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 in `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()`.

`dvp_Flags` identifies the nature of the found port. The flags are also defined in `dos/dosextens.h`. 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 pair returned does not necessarily contain the (or an) object identified by the `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()`. It shall not be interpreted or altered by the caller.

The member `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 11 and in particular section 11.1.1. The packets listed in chapter 12 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 the components of `name` behind the colon (":"), or the full `name` without one. 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;
  const UBYTE *name;
  struct Buffer {
    UBYTE buf[256];
  };
  D_S(struct Buffer,buf);
  /*
  ** Convert to BSTR on the stack - no size check performed!
  */
  name = strchr(path,':');
  if (name == NULL) {
    name = path;
  } else ++name;
  buf->buf[0] = strlen(name);
  memcpy(&buf->buf[1],name,buf->buf[0]);
  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;
  }

  if (dp)
    FreeDeviceProc(dp);

  SetIoErr(error);

  return lock;
}
```

It uses the `D_S` macro from section 2.4 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 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 11.2.1. 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 12.1.1 and following), causing the side effect of creating a stray window or even an error on 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 11. Handlers such as the CON-Handler will process this packet, and if after processing this packet, no file handle is open, will terminate itself again, see also section 11.2.1.

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 need to be aware of this defect.

If the path starts with `CONSOLE:`, then this function reports the port of the current console, see 7.2.4, and a `ZERO` lock, but sets `IoErr()` (incorrectly) to an error code even if process is equipped with a console. The same happens for paths relative to `PROGDIR:`, the home directory of the calling process. Thus, `IoErr()` shall not be interpreted in case of success.

Also, `GetDeviceProc()` does not handle the path "`*`" at all, even though it indicates the current console and the CON-Handler is responsible for it. This case also needs to be detected by the caller, and in such a case, the port of the console should be obtained through `GetConsoleTask()` specified in section 7.2.4.

> **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 7.6.5. A use-count identfying 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. This is because this function extracts the device name into a temporary buffer of this size, and truncates it if it is longer.

## 7.2.2 Releasing DevProc Information

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

```
FreeDeviceProc(devproc)
                     D1

void FreeDeviceProc(struct DevProc *)
```

This function releases the `DevProc` structure and all its resources from an iteration through one or multiple `GetDeviceProc()` calls. It shall be called only to abort an iteration over devices, not within each iteration, see section 7.2.1 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 4.8.1 and 5.2.5.

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.

### 7.2.3   Legacy Handler Port Access

The `DeviceProc()` function is a legacy variant of `GetDeviceProc()` that should not be used anymore. It is not able to handle 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 a handler responsible for the path `name`. It returns `NULL` on error in which case it sets `IoErr()`.

If the passed in `name` indicates an assign, the handler port responsible for the directory the assign binds to is returned, and `IoErr()` is set to the lock of the assign. Similar to `GetDeviceProc()`, this lock is owned by the *dos.library* and shall be duplicated before it is used to iterate over a directory.

> *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 properly handle `NIL:` and "`*`".

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.

### 7.2.4   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()`.

### 7.2.5   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 itself.

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

```
port = GetFileSysTask() /* 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 assign, they can be different.

## 7.3   Iterating and Accessing the Device List

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

### 7.3.1   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 a flag that defines 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 7.3.5.

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

**Table 30: LockDosList Flags**

| Flags | Description |
|---|---|
| LDF_DEVICES | Access handlers and file system entries, see 4.3.1.1 |
| LDF_VOLUMES | Access volumnes, see 4.3.1.2 |
| LDF_ASSIGNS | Access assigns, see 4.3.1.3 |
| LDF_ALL | Combination of all of the above |
| LDF_ENTRY | Handler startup access the device list |
| LDF_DELETE | Lock device list for deletion |
| LDF_READ | Shared access to the device list |
| LDF_WRITE | Exclusive access to the device list |

At least `LDF_READ` or `LDF_WRITE` shall be included in the flags, they shall not be set both. For example, `GetDeviceProc()` performs a read access on the list while looking for a particular device, while adding assigns through `AssignLock()` (see section 7.6.1) 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 `LDF_DELETE` is set, then access is granted for removing entries. Both flags are only set internally by the *dos.library* and shall not be used 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 a `DosList` entry representing a volume inserted into the drive maintained by the file system. Unfortunately, even some system programs *do* keep a lock (despite the above warning), and attempting to retrieve a lock from within the file system would then cause a deadlock. Instead, handler and file system implementations should rather call `AttemptLockDosList()` specified in section 7.3.2 and defer the intended operation until after the list becomes accessible.

For backwards compatibilty 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()`.

## 7.3.2   Requesting Access to the Device List

The `AttemptLockDosList()` requests access to the device list or a subset of its entries, and, in case it cannot gain access, returns `NULL`. Unlike `LockDosList()`, it does not block, 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 30, and the semantics of the flags are exactly as specified for `LockDosList()`, see 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 7.3.1.

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, and thus – if compatibility to such versions is intended — both return codes would need to be checked. This was fixed in release 40 at least.

This function does not alter `IoErr()`.

### 7.3.3 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 again. 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.

### 7.3.4 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 elements of Table 30 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:

```
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 at this point locked. See also the notes in section 7.3.1. If the names are required, they shall be copied within the loop and then printed later.

### 7.3.5   Find a Device List Entry by Name

The `FindDosEntry()` function finds a `DosList` structure of a particular type and particular name, from a particular entry on, or the handle returned by `LockDosList()` 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 30 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 section 4.3. It may be `NULL` in which case every entry of the requested type matches.

The returned `newdlist` is a pointer to a `DosList` structure that matches the name (if provided) and flags passed in, or `NULL` in case no match could be found and the entire list has been scanned. Note that the returned `DosList` may be identical to the `dlist` passed in if it already fits the requirements. Thus potentially, `NextDosList()` may be called upfront to 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 6 in section 4.3.1.1 and assigns from table 8 in section 4.3.1.3 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()`.

### 7.3.6   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 27 in section 7 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` field; it is, however, up to the handler to interpret it, and AmigaDOS does not define its syntax. The `Mount` command can place three 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 mount information.

The following algorithm attempts to detect the nature of the information provided in `dol_Startup` given a pointer to a `DosList`, for example as provided 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;
           /* Ok, this is probably good enough... use the
           ** elements of the environment up to the
           ** one indicated by env->de_TableSize
           */
           ...
         }
       }
     }
   } else {
     /* This is probably a string.
     ** "mount" puts NUL-terminated strings in
```

```
       ** here.
       */
       text = ((UBYTE *)startupmsg)+1;
       ...
   }
 } else {
   /* Likely some sort of handler, and s is
   ** probably some integer
   */
   ...
 }
}
```

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.

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 contain all the elements listed in section 7.1.3, instead, only the first `de_TableSize` elements are present, and everything beyond this point shall not be interpreted or modified. AmigaDOS 32 and above allocate at least 11 entries here, 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.

Nevertheless, getting hands on the `FileSysStartupMsg` from an unknown handler is not completely waterproof at this moment, and to this end the author of [10] proposed to introduce a `DosPacket` (see table 146 in section 12.11) 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 heuristics may 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 written in such a 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.

## 7.4 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 upon 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. Thus, at the time the handler is initiated, it may find the device list unaccessible. Attempting to lock it 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. The following sections provide workarounds how to avoid this situation, see also section 11.2.1 for details on the handler and file system startup mechanism.

> *Locking the device list in file systems* 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 to reach out for the same 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.

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

Assigns *shall not* be added to the device list through this function, but rather through the functions in section 7.6. 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 7.

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

### 7.4.2 Removing an Entry from the Device List

The `RemDosEntry()` removes a `DosList` entry from the device list, making it unacessible for Amiga-DOS.

```
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 represening their volumes and hence should be aware how to release the memory taken by them, there is no good solution on how to recycle memory for `DosList` structures representing handlers, file systems or assigns. Some manual footwork is currently required, see also `FreeDosNode()`. In particular, as entries representing handlers and file systems may have been created in multiple ways, their memory cannot be safely recycled. Assigns *shall not* be manipulated through this function, but rather through the functions in section 7.6.

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 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_DELETE|LDF_ENTRY|LDF_WRITE)) {
  rc = RemDosEntry(volumenode);
  UnLockDosList(LDF_VOLUMES|LDF_DELETE|LDF_ENTRY|LDF_WRITE);
}
```

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

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

## 7.5 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 7.6 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 `DosList`s along with all its resources is unfortunately much harder. For assigns, regardless of their type, `AssignLock()` from section 7.6.1 with a `ZERO` lock is a better 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.

### 7.5.1 Creating a Device List Entry

The `MakeDosEntry()` creates an empty `DosList` structure of the given type, and performs elementary initializations. 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 7.6 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 27 in chapter 7. 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 members except `dol_Type` and `dol_Name` are initialized to 0, and additional initialization is 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.

### 7.5.2 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 7.4.2. 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 7.6 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 7.6 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. This function is, however, called internally from other functions of the *dos.library*.

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

## 7.6 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 and enqueue and remove `DosList` structures representing assigns in a single call and are thus easier to use.

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 such locks are passed out by `GetDeviceProc()` (see section 7.2.1) 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 to another type.

Multi-Assigns are created by starting from a regular assign to one target directory with `AssignLock()`, see section 7.6.1, and then iteratively adding directory after directory to the assign with the `AssignAdd()` function specified in section 7.6.4.

### 7.6.1 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, or the assign is canceled.

```
success = AssignLock(name,lock) /* since V36 */
D0                      D1    D2

BOOL AssignLock(STRPTR,BPTR)
```

This function creates, updates or canceles the assign identified by `name` which shall not include a trailing colon (":") separating the assign name from the rest of the path. 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 into 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 7.2.1.

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`.

### 7.6.2  Create or Update a Non-Binding Assign

The `AssignPath()` function creates or updates 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 = 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 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 directory of the calling process.

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`.

### 7.6.3  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`.

### 7.6.4   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 should no longer be used by the caller. The assign is converted into a multi-assign on access if it is not already one. The `lock` is added at the end of the directory list, i.e. the new directory is scanned last when resolving the assign.

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

### 7.6.5   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.

## 7.7  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.

### 7.7.1  Adjusting File System Buffers

The `AddBuffers()` function increases or reduces the number of buffers of a file system.

```
success = AddBuffers(filesystem, number) /* since V36 */
D0                              D1          D2

BOOL 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 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 7.2.1 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 onwards, the underlying packet, see section 12.9.1, 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, it sets `IoErr()` to 0, otherwise it is set to an error code. Some file systems return the current number of buffers allocated; callers should thus be prepared that the return code is not equal to `DOSTRUE` to indicate success.

### 7.7.2 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 relables 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, the device or the volume name. As `volumename` 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. 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, 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; for file systems that do not, this function fails.

Even though this function only exists from AmigaDOS version 36 onwards, the underlying packet, see section 12.7.4, 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 the first case, it sets `IoErr()` to 0, in the latter case to an error code.

### 7.7.3 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 adminstration 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 a path name relative to the current directory, and thus will initialize the file system containing it.

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.

To block processes from accessing information on the file system while it is initializing, it should be inhibited upfront, e.g by `Inhibit(filesystem,DOSTRUE)` or by lower level packet communication to the handler, see section 12.9.2. Initializing is the only operation file systems are able to perform while being inhibited.

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 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 the command `TD_FORMAT`, 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 flavour of file system created on the device if the file system allows multiple variations. The flavours the Fast File System supports along with other file systems are listed in table 29 in section 7.1.3. This corresponds to the `DOSTYPE` in the mountlist. File systems may also ignore this argument if they only support a single flavour.

Unfortunately, AmigaDOS does not provide an easy way to access the flavours supported by a file system. The `Format` command of the Workbench offers the types listed in the first half of table 29 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 7.1.

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.

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 set to 0 on success or an error code on failure.

### 7.7.4  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, but initializing a file system also requires this function, see section 7.7.3.

```
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.

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 5.2.4. 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 onwards, the underlying packet, see section 12.9.2, 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 0. On error, it returns 0 and provides an error code in IoErr().

### 7.7.5   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" on it, thus refusing 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.

# Chapter 8

# Pattern Matching

Unlike other operating systems, it is neither the file system nor the shell that expands wild cards, 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 *tokens* that define which names to match. The following tokens are currently defined, see also `dos/dosasl.h`:

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

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

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

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

[] The square brackets ("[]") matches a single character from a range, e.g. [a-z] matches a single alphabetic character and [0-9] matches a single digit. Multiple ranges and individual characters can be combined, for example [ab] matches the characters a and b, whereas [a-cx-z] matches the characters from a to c and from x to z. If the minus sign ("-") is supposed to be part of the range, it shall appear first, directly within the bracket, e.g. [-a-c] matches the dash and the characters a to c. If the dash is the last character in the range, all characters up to the end of the ASCII range, i.e. 0x7f match, but none of the extended ISO Latin 1 characters match. If the closing square bracket ("]") is to matched, it shall be escaped by an apostroph ("'"), i.e. [['']] matches the 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 apostroph (') is the escape character of the pattern matcher and indicates that the next character is not a token of the matcher, but rather stands for itself. Thus, '? matches the question mark, and only the question mark, and no other character. The apostroph only escapes the tokens in this list, if a non-wildcard character follows the apostroph, it stands for itself. That is, the pattern 'a stands for itself, i.e. a two-character sequence starting with the apostroph.

% 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 tokens to its left or the tokens 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 inherits from the BCPL syntax and Tripos. While there is a flag in the *dos.library* that makes the asterisk *also* available as a wildcard, such usage is discouraged because it can lead to situations where the 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 wildcard. The directory scanning function MatchFirst() performs this conversion internally, and thus no additional preparation is required by the caller in this case. However, if the pattern matcher is used to search for strings or wildcards within a text file, the pattern tokenizers ParsePattern() or its case-insensitive counterpart ParsePatternNoCase() shall be called first to preprocess the pattern.

> *Only ISO-Latin Codepoints* The pre-parsing step that prepares from the input pattern its tokenized version uses the code points 0x80 to 0x9f for tokenized versions of wild-cards and other instructions for the pattern matcher. This is identical to the extended ISO-Latin control sequence region. The code points in this range do not represent printable characters. The Fast File System also disallows such characters in file names.

## 8.1 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 forewards compatibility. All remaining elements are initialized by the *dos.library* itself. In the simplest case, the structure is allocated from the exec memory pool with the MEMF_CLEAR flag.

This structure contains the state of the directory scanner, including a FileInfoBlock structure (see section 6.1) 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 2.4.

Then `MatchFirst()` shall be called, retrieving an `AnchorPath` structure as second argument, see section 8.1.1. 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 8.1.2. Once the end of a recursively entered directory has been reached, `MatchNext()` sets the `APF_DIDDIR` flag, then reverts back to the parent directory continuing the scan there. As `APF_DIDDIR` is never cleared by the pattern matcher, the caller should clear it once the end of a sub-directory had been noticed.

The above iterative procedure of `MatchNext()` may continue, either until the user or the running program requests termination, or until `MatchNext()` returns an error. Then, finally, the scan is aborted and all resources but the `AnchorPath` structure shall be released by calling `MatchEnd()` described in section 8.1.3.

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 semantic 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`. This structure is allocated and released by the *dos.library*, transparently to the caller. The `AChain` structure describes a directory in the potentially recursive scan through a directory tree. 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 top-most directory at which the scan started. These two locks have been obtained and will be unlocked by the *dos.library*; they may be used by the caller provided they are not unlocked manually.

`an_Info` is only used internally and is the `FileInfoBlock` of the directory being describes by the `AChain` structure, see section 6.1 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. 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 recur-sively, 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 emu-late 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 recog-nized 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 may cause endless recursion if a link within a directory points to a parent directory. Thus, callers should be aware of such situations and store directories that have already been analyzed. Otherwise, it is safer to keep this flag cleared.

`ap_Strlen` is the size of the buffer `ap_Buf` that 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 returned. If the complete path is not required, this member 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 $l$ bytes, in total sizeof(AnchorPath)+l-1 bytes are required to store the structure and the buffer. The byte size of this additional buffer shall be placed in ap_Strlen. If this buffer is not present, ap_Strlen shall be set to 0.

### 8.1.1  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 8.2), i.e. MatchFirst() performs the initial step of preparsing 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 path name of the matching objects. If this path name is not required, ap_Strlen shall be set to 0. ap_Flags shall be set to the flags you need, see the parent section. 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 wildcard 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 wildcard, 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 represented as a lock containing the found object is available in `ap->ap_Current->an_Lock`. You would typically set the current directory to this lock, then access this object, then revert the directory. This lock *shall not* be released; they are implicitly released by the functions of this section.

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.

### 8.1.2   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 `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 2.4 to place the `AnchorPath` structure on the stack and align it properly.

```
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 8.1.1 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 11.7.4. 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 29 in section 7.1.3 which consider upper and lower characters from the ISO-Latin-1 supplement block as non-identical. For the remaining flavours, the FFS interpretation of case-insensitive comparison is, most likely, the correct one assuming the locale is also based on ISO-Latin-1.

### 8.1.3 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 8.1.1.

## 8.2 Matching Strings against Patterns

While the prime purpose of the pattern matcher is to scan directories, it can also be used to check whether an arbitrary string matches a wildcard, for example to scan for a pattern within a text document. This requires two steps: In the first step, the wildcard is preparsed, generating a tokenized version of the pattern. The second step checks whether a given input string matches the pattern. You would typically tokenize the pattern once, and then use it to match multiple strings against the pattern. The encoding of the tokens is actually not specified, and can change from version to version, however see also the warning at the end of section 8, namely that they use the ISO-Latin extended control character range, and thus, patterns are restricted to the ISO-Latin character set. The tokens the tokenizer understands are those from section 8.

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

### 8.2.1 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. However, as future implementations can require larger buffers, the result code shall be checked nevertheless for error conditions. The result code `IsWild` is one of the following:

`1` is returned if the source contained wildcards.

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

-1 is returned in case of an error, either because the input pattern is ill-formed, or because `DestLength` is too short. 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.

### 8.2.2   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. However, as future implementations can require larger buffers, the result code shall be checked nevertheless for error conditions. The result code `IsWild` is one of the following:

1 is returned if the source contained wildcards.

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

-1 is returned in case of an error, either because the input pattern is ill-formed, or because `DestLength` is too short. 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.

### 8.2.3   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.

### 8.2.4 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 9

# Processes

*Processes* are extensions of exec *tasks*. The most important extensions are that processes include a message port in the form of a `MsgPort` structure for inter-process communication to handlers, a current directory to resolve relative paths, 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;

    /* 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 */
```

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 [7]. The only difference between an exec *Task* and a *Process* is that `pr_Task.tc_Node.ln_Type` is set to `NT_PROCESS` instead to `NT_TASK`. 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.

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.

`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 chapter 11.

`pr_Pad` is unused and only included in the structure to ensure that all following members are aligned to long word boundaries.

`pr_SegList` contains an array of segments which are used to populate the BCPL Global Vector by the BCPL runtime binder, see also section 10.5.3. The usage of the segment array and the Global Vector is questionable as version 47 removed the last BCPL compiled handler from the system. In consists of an array of BCPL words, thus 32 bit entries representing either `LONG`s or `BPTR`s, understanding the equivalence of both types in BCPL. Its contents will be briefly described in the following for historical purposes:

The first entry in this array is a 32-bit integer indicating the number of elements the BPCL runtime binder scans for populating the Global Vector, the remaining entries are BPTRs to segment lists containing AmigaDOS and program globals. Some entries may be `ZERO` indicating that the corresponding entry is currently not used. Segments are explained in more detail in chapter 10. 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 loaded binary. 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 28 in chapter 7. For BCPL code, the program segments are also used to populate the Global Vector, see section 10.5.3 for details, and thus the count value is then 3. Segments of C and assembler programs lack BCPL runtime binding information, they are excluded in 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 13.7, 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 much relevant information and the BCPL runtime binding mechanism will likely be phased out at some point.

`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. 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 13.2.6. 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 opposide 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 better left alone.

`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 9.2.9, and it can be altered by `SetIoErr()`, see section 9.2.10. 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 throug this lock, i.e. they are relative to `pr_CurrentDir`. If this member 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 9.2.7 and `CurrentDir()` from section 9.2.8 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 9.2.1. 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 9.2.2.

`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 9.2.3. 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 9.2.4.

`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 9.2.11. The pointer stored in this element should better be obtained by `GetConsoleTask()`, see section 7.2.4.

`pr_FileSystemTask` is the `MsgPort` of the file system that is contacted in case a path relative to the `ZERO` lock is to be resolved, i.e. if `pr_CurrentDir` is `ZERO`. This member is initialized to the `MsgPort` of the file system the system was booted from, but can be changed by `SetFileSysTask()`, see section 9.2.12. This member is also returned by `GetFileSysTask()` specified in section 7.2.5, 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 member is ZERO. This is for example the case for programs started from the Workbench, or handler or file system. This structure is specified in more detail in section 13.3.7, and the structure should be preferably 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 9.1.3. 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 mechanism, 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. If this is NULL, the system default function is used. 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. 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 11.1.1 and chapter 11 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 14.2.1.

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 13.6, 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 8. If this lock is `ZERO`, any attempt to resolve a path within `PROGDIR:` will create a request to insert a volume named `PROGDIR:`[1].

`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 member or extend the flags stored here. Table 31 lists the currently used flags defined in `dos/dosextens.h`:

**Table 31: 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 9.1.1.

The `PRF_FREECURRDIR` flag releases `pr_CurrentDir` via `UnLock()` upon termination of the process. This flag is implied when creating processes by `CreateNewProc()` and a non-`ZERO` lock is made the current directory of the created process. The only way how to clear this flag is to install a `ZERO` lock by the `NP_CurrentDir` tag, see also section 9.1.1 and the description of this tag.

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 quite more useful that `pr_ReturnAddr`. The function prototype is as follows:

---

[1]This is probably unintended and a defect of `GetDeviceProc()`.

```
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 9.1.1.

`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 corresponds to 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 13.3.7. The argument string can also be found in register `A0`, or in the buffer of `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 member remains `NULL`. The command line arguments can also be retrieved through `GetArgString()` specified in section 9.2.15. Even though this is possibly a bit pointless, the argument string can also be set by `SetArgString()` from section 9.2.16, 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 13.1.5 and following. The structure of a node in this list is specified in section 13.4.3 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, the list and its contents should not be accessed directly at all, but through the accessor functions listed in section 13.4.

`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 9.2.6. If `pr_CES` is ZERO, processes should print errors through `pr_COS` instead; preferably, processes should use the `ErrorOutput()` function defined in section 9.2.5 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 onwards.

> *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.

## 9.1   Creating and Terminating Processes

AmigaDOS provides several functions to create processes: `CreateNewProc()` is the extended 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 available under all AmigaDOS versions. Shells as created by the `System()` function implicitly also create a process, but are not discussed here, but in section 13. For that reason, `System()` shares a couple of options with `CreateNewProc()`.

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 also terminates a process, but from within the process to abort and not from its outside. Worse, it is typically not suitable for this job either as it does not release resources acquired by the program itself, but only those allocated by the `CreateProc()` or `CreateNewProc()` functions. In no event should processes be terminated by the *exec.library* function `RemTask()` as this function will miss to release even the resources allocated by the *dos.library* when creating processes, leave alone resources allocated by the running program.

`CreateProc()` and `CreateNewProc()` are in general not sufficient to launch programs designed to be run from the Workbench, nor to start command line programs as if they were called from the shell. While the Workbench also goes through `CreateNewProc()`, the delivery of the `WBStartup` message documented in `workbench/startup.h` is necessary in addition. Such a message can be created manually and delivered to the process once started. The `WBLoad` program added to AmigaOs version 45 emulates the Workbench startup mechanism by such a technique.

For the latter, the `RunCommand()` function of section 13.2.3 or the `System()` function described in section 13.2.1 are more appropriate. The former overlays the calling process with a command represented by a segment list, e.g. obtained from `LoadSeg()`, but requires the presence of a shell environment on the caller side. The latter creates a shell, loads a command from within this new shell and executes it there. More details on these functions are found in the corresponding sections.

### 9.1.1 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 points 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; 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 $20$. The default is $0$.

`NP_ConsoleTask` specifies a pointer to a `MsgPort` to the handler that is responsible for the console of the process to be created. That is, if the created process opens "`*`" or a path relative to `CONSOLE:`, it will use the specified handler. The default is to use the console handler 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 14.2.1. 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 the provided 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 13.3.7 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 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 13.3.7. 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 13.3.7.

`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 4.7 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 9 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 9.

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.

> *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.

### 9.1.2 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 member `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 may be released again. The Workbench startup protocol described in [7] 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.

### 9.1.3 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()`, `CreateProc()` or 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 9 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 therre 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.

## 9.2 Process Properties Accessor Functions

Many elements of the `Process` structure described in section 9 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.

### 9.2.1 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 13.1.1. Programs started from the workbench do not receive an input file handle at all, `pr_CIS` remains `ZERO` unless explicitly set[2].

### 9.2.2 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.

### 9.2.3 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 13.1.1. Programs started from the workbench do not receive an output file handle at all, `pr_COS` remains `ZERO` unless explicitly set[3].

### 9.2.4 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.

---

[2]The information in [10] that programs run from the workbench receive a `NIL:` handle is incorrect.

[3]The information in [10] that programs run from the workbench receive a `NIL:` handle is incorrect.

### 9.2.5 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 13.1.1. Programs started from the workbench do not receive an error file, `pr_CES` remains `ZERO` unless explicitly set.

### 9.2.6 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.

### 9.2.7 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 created the lock are used to resolve relative paths, see also section 4.3.

```
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 9.2.8.

### 9.2.8 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 section 4.3.

```
oldLock = CurrentDir( lock )
D0                       D1

BPTR CurrentDir(BPTR)
```

This function sets the current directory to `lock` and returns in `oldLock` the previously installed current directory. 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 13.3.1 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.

### 9.2.9 Return the Error Code of the Previous Operation

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 4.6 provide an error code in case of failure, or deliver 0 as secondary result in case of success. The buffered functions in section 4.7 generally only set a secondary result code in case an I/O operation is required, but do not touch `IoErr()` if the call can be satisfied from the 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 8) in `dos/dosasl.h`. Generally, handlers and file systems can select error codes as they seem fit, the list below provides a general indication how the codes are used by the *dos.library* itself, or what its suggested usage is:

`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.

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 cannot hold more than 20 processes. See also section 13.2.6 and following for more details. As this limitation was removed, the error code is unused in AmigaDOS version 36 and above.

ERROR_BAD_TEMPLATE: This error code indicates that the command line template for `ReadArgs()` is syntactical incorrect, see section 13.5.1. It is also set by the pattern matcher in case the pattern is syntactically incorrect, see 8.2.

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 6.1.4.

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 13.5.3.

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 6.3 and the soft link resolver `ReadLink()` of section 6.4.2.

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.

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 part of a binary file (see chapter 10). The structure of such files allowed runtime linking to libraries, though this possibility was phased out in AmigaDOS version 36 as it is incompatible with the AmigaOs library calling conventions. 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) 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 place, and is used as such by multiple Os components. AmigaDOS file systems use it, for example, when attempting to create a directory, but a file or a directory of the requested name is already present.

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

ERROR_OBJECT_NOT_FOUND: This is a generic error code that indicates that the object on which a particular operation is to be performed does not exist. It is for example generated on an attempt to open a non-existing file or to lock a file or directory that could not be found. 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. AmigaOs versions 45 and before generated if an attempt was made to provide a non-interactive input file to the `NewShell` command.

`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 11.1) 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, or from the extended ISO-Latin control sequence region between `0x80` to `0x9f`.

`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.

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

`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 6.4.2. This error is also generated by the `MatchNext()` function of the directory scanner specified in section 8.1 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 an unknown volume that is currently not available in any mounted drive.

`ERROR_SEEK_ERROR`: This error is generated by an attempt to `Seek()` to a file position that is either negative, or behind the end of the file. It is also signalled if the mode of `Seek()` or `SetFileSize()` is none of the modes indicated in table 11 of section 4.6.4. The FFS also sets this mode if it cannot read one of its administration blocks and thus is unable to find the data blocks that represent the 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 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, see table 22 in section 6.1.

ERROR_WRITE_PROTECTED: This error is generated by file systems if a write is attempted to a file that is write protected, i.e. whose FIBB_WRITE bit is set. This error is also generated if an attempt is made to write to a medium that is write protected, either by hardware protection or protected by software, e.g. through the Lock command, see section 12.7.7.

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 generated by a file system on an attempt to read from a disk that is not strutured according to the requirements of the file system, e.g. because it is initialized by another incompatbile file system different from the mounted one. Unfortunately, AmigaDOS does not have a control instance that selects file systems according to the disk structure.

ERROR_NO_DISK: This error indicates that an attempt was made to access a physical device which contains no medium, e.g. an empty disk drive.

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 24 of section 6.4 for the list.

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

ERROR_BAD_HUNK: Generated by LoadSeg() and NewLoadSeg(), this error code indicates that the binary file includes a hunk type that is not supported or recognized by AmigaDOS. The Hunk format for binary executables is documented in section 10.3.1.

ERROR_NOT_IMPLEMENTED: This error code is not used by any ROM component, but several Workbench components signal this error indicating that the requested function is not supported by this component. For example, the Format command generates it on an attempt to format a disk with long file names if the target file system does not support them. This error can 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 4.10 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, see section 4.10 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 section 8.

ERROR_BREAK: This error is also raised by the pattern matcher if it received an external signal for aborting a directory scan, see section 8.1. 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.

### 9.2.10   Setting IoErr

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

```
oldcode = SetIoErr(code) /* 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 4.7 that do not touch `IoErr()` in case the input or output operation can be satisfied from the buffer. A good practise is to call `SetIoErr(0)` upfront to ensure that these functions leave a 0 in `IoErr()` on success.

This function returns the previous value of `IoErr()`, and thus the same value `IoErr()` would return. See section 9.2.9 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.

### 9.2.11   Select the Console Handler

The `SetConsoleTask()` function selects the handler responsible for the "`*`" file name and `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 file relative to the `CONSOLE:` pseudo-device. Note that the argument is not a pointer to the handler process, but rather to a `MsgPort` through which this process can be contacted. It returns the previously used console handler `MsgPort`. This function sets the `pr_ConsoleTask` element of the calling process.

This function is the setter function corresponding to the `GetConsoleTask()` getter function introduced in section 7.2.4.

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 4.8.3.

In command line executables, any modification to the console task shall be reverted before the program returns to the shell.

### 9.2.12    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 `MsgPort`. This function is the setter equivalent of `GetFileSysTask()` introduced in section 7.2.5. It sets the `pr_FileSystemTask` element of the `Process` structure of the calling process.

### 9.2.13    Retrieve the Lock to the Program Directory

The `GetProgramDir()` returns a lock to the directory that contains the binary from which the calling process executes, if such a directory exists. If the executable was taken from the list of resident segments (see section 13.6), 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.

### 9.2.14    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 why 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.

### 9.2.15   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. 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 to the `pr_Arguments` element of the `Process` structure.

### 9.2.16   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 to `pr_Arguments` member 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 the program starts up.

# Chapter 10

# Binary File Structure

The AmigaDOS *Hunk* format represents executable, linkable object files and link libraries. The first type includes application programs that are loaded from the Workbench or the shell. They are specified in section 10.2. Object files are intermediate files that are generated by compilers from a translation unit, i.e. typically one source file and its headers. Object files still contain references to other parts of the compiled source which are resolved by linking, upon which they are merged together to an executable file. Object files are introduced in section 10.6. Link libraries are collections of utility functions that are linked into an application during its building process and then become part of the final executable; unlike AmigaDOS libraries stored in the `LIBS:` assign, they are not loaded at runtime and not shared between applications. The code in link libraries is instead merged into the final executable. Section 10.7 specifies their format.

A file in either format, i.e. regardless whether it is an executable, an object file or a link librariy, consists of multiple *hunks*, each of which defines either payload data as an indivisible segment of code or data that is initialized or loaded from disk, or additional meta-information. This information is interpreted by the AmigaDOS loader — i.e. the `LoadSeg()` function — or consists of references resolved by the linker. Meta information within executables is used to relocate the payload to their final position in memory, to define the size of the sections, to select the memory type that is allocated for the segment, or to interrupt or terminate the loading process. 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.

The `LoadSeg()` function loads an executable file in the Hunk format to memory, where it is represented as a singly linked list of *segments*. This process is described in section 10.3. 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*. Each segment of a binary is allocated through `AllocVec()`, which is sometimes helpful as it allows to retrieve size of the segment from the size of the allocated memory block, i.e. from the long word immediately preceeding the `NextSegment` element of the `SegmentList`.

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 13.6, but also contains a BPTR to a segment list in the above sense. The `pr_SegList` element is an array where each entry but the first consits of a BPTR to a `SegmentList`, see chapter 9.

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 the hunk is corrected according to the addresses hunks are loaded to. The relocation process takes an offset into one hunk, and adds to the longword at this offset the absolute address of any other hunk, see section 10.2.5 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 to their final position in memory.

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. The format is specified in more detail in section 10.4.1.

AmigaDOS also contains a simple run-time binder that is only used by executables written in BCPL, or by code that operates under BPCL 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 (or require) it, despite not being written in BCPL. This mechanism is described in section 10.5.3.

## 10.1 Conventions and Pseudo-Code

In the following, the syntax of files and its hunks is represented as pseudo-code in a three-column table that eases the implementation of an AmigaDOS Hunk format parser and reflects approximately the inner workings of the AmigaDOS loader or the 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 a structure that extends over multiple bytes are read from most significant to least significant bit. That is, the binary file format follows the big-endian convention. If the first column contains a question mark ("?"), the structure is variably-sized, and the number of removed bits is defined by the second column, or the section it refers to. If the first column is empty, no bits are removed from the file.

The second column either identifies the 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 only a single value is listed, then the bits read from the stream shall have this value, otherwise the file is ill-formed. This column can also contain pseudo-code that describes the flow of operations during interpretation of files. Such flow control elements follow closely the convention of the C language. In their absense, the contents of the tables shall be interpreted top to bottom, where the topmost elements are removed from the file 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 a function call 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. An expression such as $t \in [a, b]$ first removes the number of bits as indicated in the left-hand colum 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 even crash the system if the formulated constrains are violated.

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 expresssions += and -= add or subtract the value on its right to the variable on its left.

## 10.2  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 10.1:

**Table 32: Regular Executable File**

| Size | Code | Syntax |
|------|------|--------|
| ? | parse_HEADER | Defines all segments, see section 10.2.1 for details |
| | $i = t_{num}$ | Start with the first hunk, $t_{num}$ 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 10.2.1 |
| 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 advisory hunk |
|---|---|---|
| $32 \times l$ | | *l* long words of hunk contents ignored |
| | `}` | |
| | `else if (h == HUNK_END)`<br>`    i++` | Advance to next segment, see 10.2.11 |
| | `else if (h == HUNK_BREAK)`<br>`    break` | Terminate an overlay, see 10.4.4 |
| ? | `else if (h == HUNK_NAME)`<br>`    parse_NAME` | See section 10.2.8 |
| ? | `else if (h == HUNK_CODE)`<br>`    parse_CODE` | See section 10.2.2 |
| ? | `else if (h == HUNK_DATA)`<br>`    parse_DATA` | See section 10.2.3 |
| ? | `else if (h == HUNK_BSS)`<br>`    parse_BSS` | See section 10.2.4 |
| ? | `else if (h == HUNK_RELOC32)`<br>`    parse_RELOC32` | See section 10.2.5 |
| ? | `else if (h == HUNK_SYMBOL)`<br>`    parse_SYMBOL` | See section 10.2.9 |
| ? | `else if (h == HUNK_DEBUG)`<br>`    parse_DEBUG` | See section 10.2.10 |
| ? | `else if (h == HUNK_OVERLAY)`<br>`    { parse_OVERLAY; break }` | See section 10.4.3 |
| ? | `else if (h == HUNK_DREL32)`<br>`    parse_RELOC32SHORT` | Compatibility kludge<br>Use `HUNK_RELOC32SHORT` instead<br>See section 10.2.6 |
| ? | `else if (h == HUNK_RELOC32SHORT)`<br>`    parse_RELOC32SHORT` | See section 10.2.6 |
| ? | `else if (h == HUNK_RELRELOC32)`<br>`    parse_RELRELOC32` | See section 10.2.7 |
| | `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 times the hunk type is encountered.

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

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 10.4. Finally, an error condition or an attempt to read past the end of file also aborts loading the file.

## 10.2.1 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 33: 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` $\leq$ $t_{\texttt{max}}$`;j++) {` | Iterate over all hunks |
| 2 | $m_t[j]$ | Read memory type of the segment as 2 bits |
| 30 | $m_s[j]$ | Read memory size in long words as 30 bits |
| | `if (`$m_t[j]$` == 3) {` | if the memory type is 3 |
| 32 | $m_t[j]$ | Memory type is explicitly provided |
| | `}` | End of special memory condition |
| | $m_a[j]$ ` = AllocVec(`<br>`sizeof(BPTR)+`$m_s[j]$`×sizeof(LONG),`<br>$m_t[j]$`|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 [10].

The second entry $t_{\texttt{size}}$ contains the number of segments the executable consists of. In case of overlays, it is the total number of segments that can be resident in memory in total. See section 10.4 for more information. This value shall be consistent for all `HUNK_HEADER`s within an overlaid file. In regular executables, only a single `HUNK_HEADER` exists at the beginning of the file.

The members $t_{\texttt{num}}$ and $t_{\texttt{max}}$ 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, $t_{\texttt{num}}$ shall be 0, that is, the first segment to load is 0, the first index in the segment table.

For regular files, $t_{\texttt{max}}$ shall be identical to $t_{\texttt{size}} - 1$, 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, see also `exec/memory.h`. If the two memory type bits are both 1, an additional long word contains the memory type explicitly. This signalling mechanism is, however, only available in AmigaDOS 36 and up.

## 10.2.2  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 hunk of an executable should be a `HUNK_CODE`, and it should start with a valid opcode.

Compilers use typically this hunk to represent the `text` segment, i.e. compiled code and constant data, such as strings.

The structure of this hunk is as follows:

**Table 34: Hunk Code Syntax**

| Size | Code | Syntax |
|---|---|---|
|  | `HUNK_CODE [0x3e9]` | A hunk describing a segment of code and constant data |
| 32 | $l \leq m_s[i]$ | Size of the payload |
| $l \times 32$ | `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 v37 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.

## 10.2.3  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 35: Hunk Data Syntax**

| Size | Code | Syntax |
|---|---|---|
|  | `HUNK_CODE [0x3ea]` | A hunk describing a segment of data |
| 32 | $l \leq m_s[i]$ | Size of the payload |
| $l \times 32$ | `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 v37 onwards. Due to a bug in the loader in later versions, the initialization is also skipped if the hunk length $l$ is 0.

## 10.2.4  HUNK_BSS

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

The structure of this hunk is as follows:

**Table 36: Hunk BSS Syntax**

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

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

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

### 10.2.5  HUNK_RELOC32

This hunk contains relocation information for the previously loaded segment; that is, it corrects addresses within this segment by adding the absolute address of this or other segments to long words at indicated offsets of the previous segments. At 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 37: Hunk Reloc32 Syntax**

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

That is, the hunk consists first of a counter that indicates the number of relocation entries, followed by the hunk index relative to which an address should be relocated. Then relocation entries follow; each long-word defines an offset into the previously loaded segment to relocate, that is, to fix up the address. 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 37 and up (Kickstart 2.0 and later), the number of relocation entries $c$ shall not be larger than $2^{16}$. This is a known defect of the loader that has currently not yet been fixed. If more relocation entries are needed, they need to be split into multiple hunks.

### 10.2.6  HUNK_RELOC32SHORT

This hunk contains relocation information for the previously 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 in AmigaDOS 36[1]. 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 38: 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.

| Size | Code | Syntax |
|---|---|---|
| | `do {` | Loop over relocation entries |
| 16 | $c$ | Number of relocation entries |
| | `if (c == 0) break` | Terminate the hunk if the count is zero |
| 16 | $j \in [0, t_{\texttt{size}} - 1]$ | Read the hunk to which the relocation is relative to |
| | `p += c` | Update padding count |
| | `do {` | Loop over the relocation entries |
| 16 | $r_o \in [0, m_s[i] \times 4 - 4]$ | Relocation offset into this hunk as byte address |
| | `(UBYTE **) (m_a[i] + r_o) += m_a[j]` | Fixup this hunk by the start address of the selected hunk |
| | `} while(--c)` | until all entries are used |
| | `} while(true)` | until a zero-count is read. |
| | `if (p & 1) {` | check whether padding is required. |
| 16 | | dummy for long-word alignment |
| | `}` | |

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.

## 10.2.7 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 39: Hunk RelReloc32 Syntax**

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

Even if this hunk only makes sense for processes from the 68020 onwards, 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 onwards, all elements of this hunk, namely $c$, $r_o$, $j$ 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. Luckely, the support and demand for this hunk type is very limited.

## 10.2.8 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. 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 10.6.

The structure of this hunk is as follows:

**Table 40: 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 |
| $32 \times 1$ | $h_n$ | 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.

## 10.2.9 HUNK_SYMBOL

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

The syntax of this hunk reads as follows:

**Table 41: 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 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 10.6.7.

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 may be referenced by another object file, or they may be used for debugging purposes. 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. Executable files, and thus symbols within `HUNK_SYMBOL`, may only contain symbol definitions as all references had been resolved by the linker before. Thus, bit 7 is always 0 here.

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

**Table 42: Symbol types in HUNK_SYMBOL and HUNK_EXT**

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

Additional symbol types representing references are documented in section 10.6.7. References are only available within object files and as such symbol types representing them can only appear within `HUNK_EXT`.

## 10.2.10   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 hunk to the code that was compiled from this line.

The syntax of this hunk is as follows:

**Table 43: 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 hunk |
| 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 $l_v$ into the hunk to determine the position of a symbol in the hunk.

While [1] and [10] 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.

### 10.2.11   HUNK_END

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

**Table 44: Hunk End Syntax**

| Size | Code | Syntax |
|---|---|---|
| | `HUNK_END [0x3f2]` | Terminate a segment |

## 10.3   The AmigaDOS Loader

The *dos.library* provides service functions for loading and releasing binary executables in the *Hunk* format introduced in section 10. The functions discussed in this section load such binaries into memory, constructing a segment list from the hunks found in the files, or release such files. Overlay files are discussed separately in section 10.4 due to their additional complexity.

A segment list is a linearly linked list of segments as defined in section 10, 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 describe loaded executables. The loaded code can be, for example, either launched as a new process through `CreateNewProc()` explained in chapter 9.1.1 or run as command overloading the current process via `RunCommand()` introduced in section 13.2.3. 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 28 in chapter 7, use the mechanism of section 10.5.3 instead.

### 10.3.1 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 named `name` and returns a BPTR to its first segment in case of success, or `ZERO` in case of failure. The `name` is passed into the `Open()` function, 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 10.3.4.

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 in 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 10.2.6, 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 10.2.7. 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 0 in case of success.

### 10.3.2 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 its first segment, similar to `LoadSeg()`.

Additional parameters may be provided in the form of a TagList, passed in as `tags`. The first two functions are identical and differ only by their naming convention; the last function prototype also refers to the same entry within the *dos.library*, though uses a different calling convention where the second and all following arguments form the 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 over `LoadSeg()` is provided. Version and stack information is instead extracted from by the loaded segments by the mechanisms specified in sections 10.5.1 and 10.5.2.

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 10.3.1. Thus 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 0.

### 10.3.3   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 file 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 10.4 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 file 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 4.6.1) satisfies this interface,

The `AllocFunc()` function allocates `d0` bytes of memory, requiring the memory type provided 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 Rigit Disk Block of a boot partition. In particular, the `fh` argument does not need to be a regular file handle; it is rather an opaque value identifying the source. 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 four bytes 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()`.

### 10.3.4 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 10.4, 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 9.1.2 and the `NP_FreeSeglist` tag of the `CreateNewProc()` function in section 9.1.1.

Segment lists loaded through `InternalLoadSeg()` require in general a more generic unloader. They shall be be released through `InternalUnLoadSeg()` instead, see 10.3.5.

This function returns a non-zero result in case of success, or 0 in case of error. Currently, the only source of error is passing in `ZERO` as segment list, 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.

### 10.3.5 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 that 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 10.4) 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.

## 10.4 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 other currently unused code parts and thus making more memory available.

Overlays are an extension of the AmigaDOS Hunk format that splits the executable into a root node that is loaded initially and stays resident for the lifetime of the program, and one or multiple extension or overlay nodes that are loaded and unloaded on demand. The *overlay manager* locates the overlay nodes, loads them into memory and releases unused nodes again.

AmigaDOS does not provide a ROM-resident overlay manager, i.e. the *dos.library* does not contain an overlay manager, though it provides services overlay managers use. 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; its design and data structures are all alike and discussed in detail in section 10.4.2.

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) onwards, but will nevertheless be described here as a historical artifact. Section 10.4.10 provides details on this flat overlay manager.

---

[2]Loading overlays through `InternalLoadSeg()` is probably a bad idea anyhow.

### 10.4.1 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.

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 therefore the organization of this hunk is depending 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. This hunk does not contain any data. The overall structure of an overlaid binary therefore looks as follows:

**Table 45: 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 10.4.3 |
| `do {` | Repeats over all overlay nodes |
| `HUNK_HEADER` | Defines the segments in this overlay node |
| `HUNK_CODE` or `HUNK_DATA` | First segment of the overlay node |
| ⋮ | Other hunks of this overlay node |
| `HUNK_END` | Terminates the last segment |
| `HUNK_BREAK` | Terminates the first overlay node, see 10.4.4 |
| `} while(!EOF)` | This pattern repeats until end of file |

### 10.4.2 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:
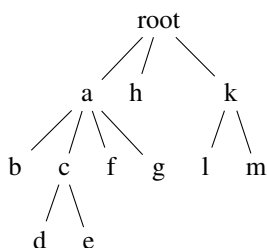


Only those nodes that form a path from the root to one of the nodes of the tree can be in memory at a time. Thus, for the above example, the root node and nodes $a$, $c$ and $e$ can be in memory simultaneously, or the root node, and nodes $k$ and $m$ can be loaded at the same time, but not the nodes $a$, $g$ 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 needs to be loaded as it is the parent of $l$.

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

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

### 10.4.3   HUNK_OVERLAY

This hunk terminates the loading process and indicates the end of the root node. The HUNK_OVERLAY contains meta-data — the overlay table — for the overlay manager. This table contains information where symbols within the overlaid segments are located. Section 10.4 provides more information on overlays. The format of the data within this hunk depends on the overlay manager which shall be included in the first segment of the executable itself as AmigaDOS does not contain a resident overlay manager.

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

**Table 46: Hunk Overlay Syntax**

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

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

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

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

```
struct OverlayHeader {
    UWORD           oh_Jump[2];    /* Forms a branch to the startup code */
    LONG            oh_Magic;      /* Shall be 0x0000abcd                 */
    BPTR            oh_FileHandle;/* Filled by the loader with the fh     */
    struct OVTab *oh_OVTab;        /* Overlay table from HUNK_OVERLAY      */
    BPTR            oh_Segments;   /* Array of segment BPTRs               */
    BPTR            oh_GV;         /* standard Global Vector               */
};
```

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

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

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

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

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

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

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

The ordinate table is followed by the symbol table. The purpose of this structure is that it allows the overlay manager on a reference to such an external symbol to find and load the overlay node containing the symbol, and then resolve references to it. How exactly it does so is explained in more detail in section 10.4.9. The elements of this structure are already briefly introduced in table 46.

`oh_Segments` is filled by the AmigaDOS loader to a BPTR to the segment table of the loaded binary. The size of this table is taken from $t_{size}$ in the `HUNK_HEADER` of the root node, see table 33. Each element in this array contains a BPTR to a segment of the loaded binary, and it is indexed by the segment number, counting from 0 for the first segment of the root node.

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

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

### 10.4.4   HUNK_BREAK

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

**Table 47: Hunk Break Syntax**

| Size | Code | Syntax |
|------|------|--------|
|      | `HUNK_BREAK [0x3f6]` | Terminate a segment |

### 10.4.5   Loading an Overload 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 overlaid node and all segments within it. For that, the file pointer shall first be placed with `Seek()` to the file offset of the `HUNK_HEADER` of the overlaid node. This file offset may, for the standard hierarchical overlay manager, be taken from the `ot_FilePosition` of the overlay table.

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


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

For 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 may be taken from `oh_Segments`. It contains BPTRs to all allocated segments, see section 10.4.3.

`fh` is a BPTR to the *FileHandle* from which the overlay node is to be loaded. This handle may be taken from `oh_FileHandle`, see section 10.4.3.

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 clean up the segment table upfront, see 10.4.7. The information which segments will be populated by an overlay node may be taken from the `ot_FirstHunk` member of the overlay table. Due to the tree

structure imposed by the hierarhical overlay manager, it has to release all segments from `ot_FirstHunk` onwards up to the end of the table, unlink the segments contained therein, and then load another overlay node through `LoadSeg()`.

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

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

### 10.4.6  Loading an Overlay Node through Call-Back Functions

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

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

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

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

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

Like `LoadSeg()`, this function does not release segments in populated entries in the segment list, it is up to the overlay manager to unload these segments. The information which entries of the segment table will be populated by an overlay node may be taken from the `ot_FirstHunk` member of the overlay table, see also 10.4.5.

The `funcs` argument points to a `LoadSegFuncs` structure as defined in section 10.3.3 and contains functions for reading data and allocating and releasing memory.

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

### 10.4.7  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, but its implementation within the standard hierarchical overlay manager documented here for completeness. Other custom overlay managers perform potentially different algorithms.

First, it finds the previous segment upfront the segment to be unloaded, and cleans its `NextSegment` pointer to unlink all following segments. Then the following segments are released through `FreeVec()` or, in case a custom allocator was provided for `InternalLoadSeg()`, whatever memory release function is appropriate.

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

```
void UnloadOverlayNode(struct OverlayHeader *oh,ULONG i)
{
```

```
  BPTR *segtbl  = (BPTR *)BADDR(oh->oh_Segments);
  BPTR *segment = (BPTR *)BADDR(segtbl[i - 1]);
  BPTR  next;

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

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

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

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

### 10.4.8   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, independent on which overlay manager has been used. UnLoadSeg() will detect the overlay manager from the magic value in oh_Magic and will then not only release the segments, but also close the overlay file handle and release the segment table.

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

### 10.4.9   Internal Working of the Overlay Manager

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

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

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

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

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

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

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

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

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

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


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


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

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

        dc.l        $ABCD                   ;Magic longword for UnLoadSeg

;* The next four LWs are filled by the loader (LoadSeg())
ol_FileHandle:  dc.l 0                      ;Overlay file handle
ol_OverlayTab:  dc.l 0                      ;Overlay table
ol_HunkTable:   dc.l 0                      ;BPTR to Overlay hunk table
ol_GlobVec:     dc.l 0                      ;BPTR to global vector (unused)

                dc.l MajikLibWord       ;Identifier word
                dc.b 7,"Overlay"        ;Identifier

;* the following data is specific to this manger

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

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

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

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

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

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

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

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

        do
         move.l (a4)+,d0                 ;next hunk ?
         break.s eq
         lsl.l #2,d0
         move.l d0,a1                    ;->a1
         move.l -(a1),d0                 ;get length
         jsr FreeMem(a6)                 ;free this hunk
        loop.s                           ;and now the next
.retry:
        move.l ol_DOSBase(pc),a6
        move.l ol_FileHandle(pc),d1      ;get our stream
        move.l ot_FilePosition(a3),d2    ;get file position
        moveq #-1,d3                     ;relative to beginning of file
        jsr Seek(a6)                     ;seek to this position
        tst.l d0                         ;found something ?
        bmi.s .loaderror                 ;what to do on failure ?

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

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

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

```
        loadregs
        rts

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

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


;***************************************************
;** NextModule                                  **
;** Open stuff absolutely necessary and         **
;** continue with main program code             **
;***************************************************
NextModule:
                                        ;why safe registers ?
        move.l a0,a2
        move.l d0,d2                    ;keep arguments
        lea ol_SysBase(pc),a3
        move.l ExecBase,a6
        move.l a6,(a3)                  ;fill in Sysbase
        lea DOSName(pc),a1              ;get name of DOS
        moveq #33,d0                    ;at least 1.2 MUST be used
        jsr OpenLibrary(a6)
        move.l d0,4(a3)                 ;Save back DOS base for loader
        beq.s .nodosexit                ;exit if no DOS here

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

        move.l ol_SysBase(pc),a6
```

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

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

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

## 10.4.10   The MANX Overlay Manager

The Aztec C compiler from MANX offers an alternative overlay manager that is related to the Resource Manager from MacOs. It does not organize overlay notes in a hierarchy, but only in a single level; however, all nodes of this single level can be loaded and unloaded independently from each other, either on demand through the function segload() provided by the MANX compiler 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 hunks that are loaded and unloaded jointly. Unfortunately, this overlay manager depends on self-modifying code and ignores the instruction cache present in later members of the Motorola 68K family, and is therefore no longer safe to use.

The MANX overlay manager organizes its nodes through the data in the HUNK_OVERLAY hunk, which, however, is in a different format than the one documented in section 10.4.3, and trampolines that direct the code flow for non-resident functions to the overlay manager. The contents of the HUNK_OVERLAY hunk contains offsets within the file, to the trampoline functions and to the symbol table that provides information in which hunks the overlaid functions reside. The trampolines are not part of HUNK_OVERLAY but reside at the beginning of the HUNK_DATA hunk, relative to the __H1_org symbol.

Even though the structure of the HUNK_OVERLAY differs from the one in section 10.4.3, the AmigaDOS LoadSeg() function does not care about the contents of this hunk as long as its first LONG word indicates its size. It provides the data within to the program as-is through oh_OVTab of the OverlayHeader which shall be present at the beginning of the first segment of the binary; this structure, of course, does not change from its definition in section 10.4.3 except that data within the overlay hunk is organized differently:

```
struct ManxOverlayHeader {
  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 MANXOv *oh_OVTab;     /* Overlay table from HUNK_OVERLAY    */
  BPTR           oh_Segments;  /* Array of segment BPTRs             */
  BPTR           oh_GV;        /* standard Global Vector             */
};
```

The overlay structure for the MANX compiler reads on disk as follows:

**Table 48: Hunk Overlay Syntax**

| Size | Code | Syntax |
|---|---|---|
| | HUNK_OVERLAY [0x3f5] | Overlay table definition for MANX |

| Size | Field | Description |
|---|---|---|
| 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 data hunk of the root node |
| 16 | $o_s[i]$ | Offset to the symbol table relative to $o_p[0]$ |
| | `}` | |
| | $l- = o_d \times 2$ | Count removed long-words |
| | $s = 0$ | Start with symbol 0 |
| | `while(`$l \geq 0$`)  {` | Repeat over the symbol table |
| 16 | $o_h[s]$ | Hunk within which the symbols reside, or 0 for end of symbols for this hunk |
| 16 | $o_c[s]$ | Symbol count for hunk $o_h[s]$ |
| | $l$ `  -=1` | Remove one long words. |
| | $s + +$ | Advance to next entry. |
| | `}` | End of loop over table |

In memory, it is approximately described by the following pseuo-structure, noting that some elements are variably sized and are thus hard to represent in the C syntax:

```
struct MANXOv {
    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[];
}
struct SymTab {
    UWORD ot_Hunk;         /* Hunk containting 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 that can all be loaded or unloaded individually. The element $o_p[i]$ providing `ot_FilePosition` is the offset relative to the start of the binary file at which the `HUNK_HEADER` of the overlay node $i$ is found. The offset $o_t[i]$ giving `ot_TrampolineOff` is the offset of the first trampoline to a symbol within overlay node $i$; the offset is relative to the second hunk 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 `SymTab` 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_Hunk` in memory, is the hunk number containing the overlaid symbol; this clearly cannot be 0 as this would be within root node. The second member of the pair, $o_c[s]$ or `ot_Count` is the

number of symbols within this hunk. 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, symbol offsets are part of the trampoline which is included in the second hunk of the root node. The offset to the first trampoline of an overlay node relative to the start the second hunk is provided by $o_t[i]$, that is, by ot_TrampolineOff.

Each trampoline looks as follows on disk:

**Table 49: 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 itself. As the trampoline is typically in the second hunk of the root node, but the code of the overlay manager is in its first hunk, this branch goes to a long 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, a 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 hunk.

When an overlay node is loaded, the overlay manager uses the symbol table consisting of SymTab 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[3], 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 symbol then looks as follows:

```
struct MANXPatchedTrampoline {
    UWORD mt_JMP;                /* filled with 0x4ef9        */
    APTR  mt_OVSymbol;           /* absolute symbol address    */
    WORD  mt_OvMngrOffset;       /* Offset to overlay manager  */
};
```

---

[3]Here the MANX overlay manager fails to clear the instruction cache, causing failure on later members of the 68K family.

When unloading an overlay node, the original trampolines have to be restored such that a call to an overlay symbol triggers again loading the hunks containing the symbol from disk. For that, `mt_JMP` is again replaced by a `bsr.w` instruction, `mt_OvMngrOffset` is moved to the next word, and the hunk offset is re-injected by counting the hunks within the singly linked list of loaded segments. Finally, `mt_SymbolOffset` is re-computed by subtracting the base address of the segment from the absolute symbol address.

## 10.5 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 their contents.

### 10.5.1 The Version Cookie

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

- The version cookie `$VER:`

- one or multiple blank spaces

- a program name, which is output by the `Version` command

- a decimal number, representing the *version* of the program

- a single dot (".")

- a decimal number, representing the *revision* of the program

- one or multiple blank spaces

- an opening bracket (" (")

- a decimal number, representing the day of the month of the revision

- a single dot (".")

- a decimal number, representing the month of the revision

- a single dot (".")

- a decimal number, representing the year of the revision

- a closing bracket (") ")

- an optional comment that is only output if the `FULL` option of the `Version` is given.

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

An example for the version cookie is

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

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

### 10.5.2  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 stack size in bytes.

The stack of the program is then, potentially increased to the provided size. Note that AmigaDOS also scans alternative sources for a stack size: The `Stack` setting in the icon information window of the Workbench, the `Stack` command of the shell, or the `STACKSIZE` entry in the mountlist. 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.

The *dos.library* provides with the `ScanStackToken()` an optimized function to quickly scan segments for the stack cookie and potentially adjust a default stacksize upwards to the value found in the stack cookie.

```
stack = ScanStackToken(segment, defaultstack)
D0                      D1        D2

LONG ScanStackToken(BPTR segment, LONG defaultstack)
```

This function scans the 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, 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.

### 10.5.3  Runtime binding of BCPL programs

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

Even though this mechanism is deprecated and AmigaDOS has long been ported to C and assembler, some components still depend on this legacy mechanism, namely all handlers and file systems mounted with the `GLOBVEC = 0` or `GLOBVEC = -3` entry in the mount list. While newer handlers should not depend on this mechanism anymore, the Port-Handler mount entry is created in the Kickstart ROM as BCPL handler, beyond control of the user. The same goes for the (now deprecated) CLI, which is also initiated as BCPL process. The AmigaDOS shell uses the same code, but uses the C startup mechanism.

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

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

The *first long-word* of the segment, usually the entry point of the process, contains instead the long-word offset from the start of the process to the start of the *Global Vector* initialization data. This initialization data

is *scanned backwards* from the given offset towards lower addresses, starting with the long-word right before the before the offset.

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

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

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

```
                SECTION text,code

                XREF    _main           ; handler or shell main

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

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

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

                CNOP    0,4

*               BCPL startup function, called for GlobVec=0 or -3
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

BCPLTable:      CNOP    0,4

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

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 it as first BCPL argument in register `d1`. It is here converted to a C-style pointer and provided as first argument to the main function, assuming the SAS/C registerized parameters ABI.

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

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

For the CLI, in particular, a second entry in the *Global Vector* shall be populated, namely the entry `G_CLISTART` at offset `#133`. It may point to the same entry code, more on this in section 13. As this entry is not used for handler startup, it does not hurt to include it in the BCPL initializer in either case, and thus the above code may be used as universal "BCPL kludge" for both the shell and legacy handlers that depend on BCPL startup.

## 10.6   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. Such files can still contain references to symbols that could not be resolved within the translation unit because the corresponding symbol is defined in another unit. The linker then combines multiple object codes, resolving all unreferenced symbol, and generates an executable binary file as output.

The overall structure of an object file is depicted in table 50:

**Table 50: Object File Format**

| Size | Code | Syntax |
|------|------|--------|
| ? | `HUNK_UNIT` | Defines the start of a translation unit, see 10.6.1 |
| | `do {` | Multiple segments follow |
| ? | `HUNK_NAME` | Name of the hunk, defines hunks to merge, see 10.2.8 |
| 2 | $m_t$ | Read the memory type of the next hunk |
| 30 | $h_t$ | Read the next hunk type |
| | `if (`$h_t$` == HUNK_CODE)`<br>`    parse_CODE` | Code and constant data, see 10.2.2 |
| | `else if (`$h_t$` == HUNK_DATA)`<br>`    parse_DATA` | Data, see 10.2.3 |
| | `else if (`$h_t$` == HUNK_BSS)`<br>`    parse_BSS` | Zero-initialized data, see 10.2.4 |
| | `else if (`$m_t$` != 0) ERROR_BAD_HUNK` | Upper bits shall be 0 for all other hunks |
| | `else do {` | Loop over auxiliary information |
| | `    if (`$h_t$` == HUNK_RELOC32)`<br>`    parse_RELOC32` | 32-bit relocation, see 10.2.5 |
| | `        else if (`$h_t$` == HUNK_RELOC32SHORT)`<br>`    parse_RELOC32SHORT` | 32-bit relocation, see 10.2.6 |

| | | |
|---|---|---|
| | else if ($h_t$ == HUNK_RELRELOC32)<br>  parse_RELRELOC32 | 32-bit PC-relative relocation,<br>see 10.2.7 |
| | else if ($h_t$ == HUNK_RELOC16)<br>  parse_RELOC16 | 16-bit PC-relative relocation,<br>see 10.6.2 |
| | else if ($h_t$ == HUNK_RELOC8)<br>  parse_RELOC8 | 8-bit PC-relative relocation,<br>see 10.6.2 |
| | else if ($h_t$ == HUNK_DRELOC32)<br>  parse_RELOC32 | 32-bit base-relative relocation,<br>see 10.6.4 |
| | else if ($h_t$ == HUNK_DRELOC16)<br>  parse_RELOC16 | 16-bit base-relative relocation,<br>see 10.6.5 |
| | else if ($h_t$ == HUNK_DRELOC8)<br>  parse_RELOC8 | 8-bit base-relative relocation,<br>see 10.6.6 |
| | else if ($h_t$ == HUNK_EXT)<br>  parse_EXT | External symbol definition,<br>see 10.6.7 |
| | else if ($h_t$ == HUNK_SYMBOL)<br>  parse_SYMBOL | Symbol definition, see 10.2.9 |
| | else if ($h_t$ == HUNK_DEBUG)<br>  parse_DEBUG | Debug information, see 10.2.10 |
| | else if ($h_t$ == HUNK_END)<br>  break | abort this segment |
| | else ERROR_BAD_HUNK | an error |
| 32 | $h_t$ | Read next hunk type |
| | } while(true) | Repeated until HUNK_END |
| | } while(!EOF) | Repeated with the next hunk until the file ends |

Since there is no HUNK_HEADER in object files, the memory attributes for the hunk are instead stored in the topmost two bits of the hunk type itself. Unlike in HUNK_HEADER, there is no documented way how to indicate a memory type beyond MEMF_CHIP and MEMF_FAST. As the interpretation of object files is up to the linker, a suggested implementation strategy is to set the topmost two bits and store the memory type in the following long-word, similar to how HUNK_HEADER expects it.

## 10.6.1  HUNK_UNIT

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

The structure of this hunk is as follows:

**Table 51: Hunk Unit Syntax**

| Size | Code | Syntax |
|---|---|---|
| | HUNK_UNIT [0x3e7] | A hunk identifying a translation unit |
| 32 | l | Size of the name in long-words |
| $32 \times 1$ | $h_n$ | Unit name |

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

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

## 10.6.2 HUNK_RELOC16

This hunk defines relocation information of one hunk into another hunk, and its format is identical to HUNK_RELRELOC32, see section 10.2.7 and table 39.. 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 52: Hunk Reloc16 Syntax**

| Size | Code | Syntax |
|------|------|--------|
| | HUNK_RELOC16 [0x3ed] | 16-bit PC-relative relocation information |
| | ... | See table 39 |

This restricts possible displacements to 16 bits, and thus the segment containing the 16-bit value to be relocated and the segment the hunk is relative to shall be linked together to form a single segment in the final output. In the notation of section 39, the segments $i$ and $j$ shall thus be merged. To make this happen, their names as provided by HUNK_NAME shall be identical; alternatively the linker shall be configured to the small data or small code model that forcefully merges hunks of the same type.

However, 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 can obviously not 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 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-bit, *ALV*s do not work correctly for data that is addressed by 16-bit PC relative modes. Instead of referencing the intended data, the executing code would then see the *ALV* as data.

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

## 10.6.3 HUNK_RELOC8

This hunk defines relocation information of one hunk into another hunk, and its format is identical to HUNK_RELRELOC32, see section 10.2.7 and table 39.. 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 hunk within which the relocation offset is to be adjusted and the target hunk shall be merged to a single hunk as the AmigaDOS loader cannot resolve 8-bit relocations. This can be either arranged by giving the two hunks the same name, or by configuring the linker to the small code or small data model, depending on the type of the hunk. This will instruct the linker to merge hunks of the same type.

**Table 53: Hunk Reloc8 Syntax**

| Size | Code | Syntax |
|------|------|--------|
| | HUNK_RELOC8 [0x3ee] | 8-bit PC-relative relocation information |
| | ... | See table 39 |

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.

## 10.6.4   HUNK_DRELOC32

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

The format of this hunk is identical to `HUNK_RELOC32`, see section 10.2.5 and table 37, where each 32-bit wide relocation offset $r_o$ points to a long-word within the preceding data or BSS hunk. The long-word at this offset is then adjusted by the position of the first byte of this hunk relative to the start of the target __MERGED hunk into which this hunk is merged; in some cases, linkers can include an additional adjustment, see the next paragraph for details.

**Table 54: Hunk DReloc32 Syntax**

| Size | Code | Syntax |
|---|---|---|
| | `HUNK_DRELOC32 [0x3f7]` | 32-bit base-relative relocation information |
| | `...` | See table 37 |

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

## 10.6.5   HUNK_DRELOC16

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

The format of this hunk is identical to `HUNK_RELOC32`, see section 10.2.5 and table 37, where each 32-bit wide relocation offset $r_o$ points to an unsigned 16-bit word within the preceding data or BSS segment. The word at this offset is then adjusted by the position of the first byte of this hunk relative to the start of the __MERGED hunk into which this hunk is merged.

**Table 55: Hunk DReloc16 Syntax**

| Size | Code | Syntax |
|---|---|---|
| | `HUNK_DRELOC16 [0x3f8]` | 16-bit base-relative relocation information |
| | `...` | See table 37 |

Similar to the comments made in section 10.6.4, 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 hunk if this hunk exceeds 32K in size. In such a case, they need to include an additional negative offset of -32K in $r_0$ when performing relocation.

### 10.6.6 HUNK_DRELOC8

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

The format of this hunk is identical to HUNK_RELOC32, see section 10.2.5 and table 37, where each 32-bit wide relocation offset $r_o$ points to a byte within the preceding data or BSS hunk. The byte at this offset is then adjusted by the position of the first byte of this hunk relative to the start of the __MERGED hunk into which this hunk is merged.

**Table 56: Hunk DReloc8 Syntax**

| Size | Code | Syntax |
|---|---|---|
| | HUNK_DRELOC8 [0x3f9] | 8-bit base-relative relocation information |
| | ... | See table 37 |

### 10.6.7 HUNK_EXT

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

The syntax of this hunk reads as follows:

**Table 57: Hunk EXT Syntax**

| Size | Code | Syntax |
|---|---|---|
| | HUNK_EXT [0x3ef] | A hunk assigning symbols to positions within a segment |
| | do { | Repeat . . . |
| 8 | $s_t$ | Symbol type |
| 24 | $s_l$ | Symbol length in long-words |
| | if (s_l == 0) break | Terminate the hunk |
| $32 \times s_l$ | $s_n$ | 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 ≥ 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 10.2.9.

Entries of a symbol type with $s_t$ < 0x80 are symbol definitions, the symbol value is defined by $s_v$. See table 58 for possible types of symbol definitions.

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

**Table 58: Symbol types in HUNK_EXT**

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

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

Common symbols are symbols that are defined in multiple translation units. The size of the symbol required by the translation $s_c$. If no corresponding symbol definition is found, the linker then allocates space of a size that is determined by the maximum of all $s_c$ values found for the same symbol in all translation units. The symbol will then be created within a BSS segment by the linker without an explicit symbol definition. This mechanism is mostly used by FORTRAN and is therefore rarely used. SAS/C can also be configured to emit common symbols as well.

## 10.7   Link Library File Format

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

part of the generated executable.

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

Link libraries come in two forms: Old-style non-indexed libraries, and indexed libraries that are faster to process. Old-style or non-indexed link libraries are simply a concatenation of object files in the form presented in section 10.6 and table 50. Then, of course, one translation unit as introduced by `HUNK_UNIT` 10.6.1 is not necessarily terminated by an `EOF` as specified in table 50, but possibly by the subsequent program unit, starting with another `HUNK_UNIT`.

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

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 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 59.

**Table 59: Indexed Library**

| Size | Code | Syntax |
|------|------|--------|
| | `do {` | Multiple repetitions of the following |
| ? | `  HUNK_LIB [0x3fa]` | Object code modules, see section 10.7.1 |
| ? | `  HUNK_INDEX [0x3fb]` | Indices into `HUNK_LIB`, see section 10.7.2 |
| | `} while(!EOF)` | Until the end of the file |

## 10.7.1 HUNK_LIB

The `HUNK_LIB` hunk contains the actual playload 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 60 depicts the syntax of this hunk.

**Table 60: 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 | $h_t$ | Read the next hunk type |
| | `if ($h_t$ == HUNK_CODE)` `  parse_CODE` | Code and constant data, see 10.2.2 |
| | `else if ($h_t$ == HUNK_DATA)` `  parse_DATA` | Data, see 10.2.3 |

| | | |
|---|---|---|
| | `else if (`$h_t$` == HUNK_BSS)` `parse_BSS` | Zero-initialized data, see 10.2.4 |
| | `else if (`$m_t$` != 0) ERROR_BAD_HUNK` | Upper bits shall be 0 for all other hunks |
| | `else do {` | Loop over auxiliary information |
| | `if (`$h_t$` == HUNK_RELOC32)` `parse_RELOC32` | 32-bit relocation, see 10.2.5 |
| | `else if (`$h_t$` == HUNK_RELOC32SHORT)` `parse_RELOC32SHORT` | 32-bit relocation, see 10.2.6 |
| | `else if (`$h_t$` == HUNK_RELRELOC32)` `parse_RELRELOC32` | 32-bit PC-relative relocation, see 10.2.7 |
| | `else if (`$h_t$` == HUNK_RELOC16)` `parse_RELOC16` | 16-bit PC-relative relocation, see 10.6.2 |
| | `else if (`$h_t$` == HUNK_RELOC8)` `parse_RELOC8` | 8-bit PC-relative relocation, see 10.6.2 |
| | `else if (`$h_t$` == HUNK_DRELOC32)` `parse_RELOC32` | 32-bit base-relative relocation, see 10.6.4 |
| | `else if (`$h_t$` == HUNK_DRELOC16)` `parse_RELOC16` | 16-bit base-relative relocation, see 10.6.5 |
| | `else if (`$h_t$` == HUNK_DRELOC8)` `parse_RELOC8` | 8-bit base-relative relocation, see 10.6.6 |
| | `else if (`$h_t$` == HUNK_EXT)` `parse_EXT` | External symbol definition, see 10.6.7 |
| | `else if (`$h_t$` == HUNK_SYMBOL)` `parse_SYMBOL` | Symbol definition, see 10.2.9 |
| | `else if (`$h_t$` == HUNK_DEBUG)` `parse_DEBUG` | Debug information, see 10.2.10 |
| | `else if (`$h_t$` == HUNK_END) break` | abort this segment |
| | `else ERROR_BAD_HUNK` | an error |
| 32 | $h_t$ | Read next hunk type |
| | `} while(true)` | Repeated until `HUNK_END` |
| | `} while(!EOF)` | Repeated with the next hunk until the file ends |

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`. The corresponding symbol types to be removed are listed in table 42. Unlike symbol definitions, symbol references as given by table 58 shall be retained as only the reference, 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.

### 10.7.2   HUNK_INDEX

The `HUNK_INDEX` hunk contains a string table and indices into the preceding `HUNK_LIB`.

Table 61 depicts the syntax of this hunk.

**Table 61: 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** |
| | `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 | $u_o$ | Byte offset of the unit name into the string table |
| 16 | $h_o$ | Long-word offset 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 | $h_n$ | Byte offset of the hunk name into the string table |
| 2 | $m_t$ | Memory type of the hunk |
| 14 | $h_t$ | (Shortened) hunk type |
| 16 | $x_c$ | Number of references in the hunk |
| | `for(k=0;k<x_c;k++) {` | Loop over references |
| 16 | $x_n$ | Byte 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 | $d_n$ | Byte offset of the defined name into the string table |
| 16 | $d_o$ | Byte offset into the hunk or absolute 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 | $d_t$ | Type of the definition from table 42 |
| | `}` | |
| | `}` | |
| | `} 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, 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 rest of `HUNK_INDEX` contains the offsets into the hunks along with symbols referenced and defined within them. The memory type of the hunk is again expressed in two bits. The current specification does not define the meaning of both bits set. The hunk type itself is abbreviated, i.e. only the lower 14 bits of the hunk type are stored.

The first part of the subsequent data defines references, that is, symbols that are used but not defined within the hunk. The $x_n$ values define the names of these symbols as offsets from the start of the string table.

These names correspond to symbol types defined in table 58. The precise reference type of the symbols is then found in an HUNK_EXT hunk as part of the preceding HUNK_LIB hunk.

Unlike symbol references, symbol definitions are stored directly in the HUNK_INDEX hunk. While $d_n$ identifies the name of the symbol as offset into the string table, the $d_o$ value is either the offset of the defined symbol within the hunk, or the least significant 16 bits of an EXT_ABS definition. As some absolute values can exceed 16 bits, $a_o$ stores bits 23 to 16 of the symbol value, and bit $a_s$ is copied into bits 31 to 24 of the symbol to enable negative values. The type of the symbol definition is $d_t$, which shall be a value from table 42.

As seen from this definition, symbols can only be defined within the first 64K of a hunk. This is typically not a problem as link libraries typically contain short service functions. For absolute values, larger values are required, e.g. for the base address of the custom chips, and thus split into $a_s$, $a_o$ and $d_o$.

# Chapter 11

# Handlers, Devices and File Systems

The *dos.library* does not implement all 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* provides a virtual file system that provides a common function interface for its clients, but requires the services of handlers to implement them. *File systems* are special handlers that organize data streams on volumes such as hard disks in files, and also provide locks to reserve access rights to files. Most file systems also support directories and thus a hierarchical organization of files.

Handlers are recognized by an entry in the device list whose `dol_Task` element provides a pointer to a message port through which the *dos.library* communicates with the handler, section 7 provides further details on this structure.

To ease communication, the `FileHandle` structure representing a file also includes with its `fh_Type` element a pointer to such a port that is typically a copy of `dol_Task` when a file is opened, see section 4.8.1 for this structure. It is created by the *dos.library* when opening files.

Similarly, the `fl_Task` element of the `FileLock` structure includes the port through which communication with the file system concerning the lock is routed. Unlike `FileHandle` structures, the `FileLock` structure is created by the file system itself which initializes its `fl_Task`. This is again in many cases identical to the `dol_Task` message port within the device list.

## 11.1   The DosPacket Structure

While it is in many cases more practical to interact with handlers through the functions of the *dos.library*, it is also possible to communicate with the handler directly through this port. This communication is based on *packets*, 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 `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`. The reply port of the message in `mn_ReplyPort` is *not* used; instead, the message carrying the packet is send back to `dp_Port`.

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 to the `DosPacket` itself.

`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 `pr_MsgPort` of the process sending the packet. See section 9 for the definition of the `Process` structure.

`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. Section 12 lists the currently documented packet types.

`dp_Res1` is the primary result code of the activity performed by the handler. 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 functions of the *dos.library* return `dp_Res1` as their return code.

`dp_Res2` is the secondary result code installed by the handler and is typically is 0 for success, or an error code on failure. Many functions of the *dos.library* install this error code into `IoErr()`. Section 9.2.9 lists the error codes defined by the *dos.library*.

`dp_Arg1` to `dp_Arg7` provide additional arguments to the handler. They shall be filled by the process submitting a packet to a handler. Most packet types do not require all 7 possible arguments; in such a case, only the necessary arguments may be initialized.

### 11.1.1 Send a Packet to a Handler and Wait for Reply

The `DoPkt()` function creates from its arguments a `DosPacket` structure along with an exec `Message` carrying it on the fly, transmits the packet to a target port and waits for the packet to return.

```
result1 = DoPkt(port,action,arg1,arg2,arg3,arg4,arg5)
D0                      D1    D2    D3    D4    D5    D6    D7

LONG DoPkt(struct MsgPort *,LONG,LONG,LONG,LONG,LONG,LONG)
```

This function performs low-level packet IO to a target message port belonging to a handler. 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. If the communication is related to a lock, the `fl_Task` member of the `FileLock` is the right source. For activities unrelated to locks or files, the `dol_Task` member of the device list is another source.

`action` identifies the activity to be performed by the handler or file system. Section 12 lists the packet types and how they relate to the functions of the *dos.library*.

`arg1` through `arg5` are arguments to the packet and filled into their `dp_Arg1` through `dp_Arg5` elements. If more arguments are required, the packet needs to be created and transmitted manually.

This function returns the primary result code of the handler from `dp_Res1` in `result1`, and `dp_Res2` in `IoErr()`.

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 section 9. Otherwise,

`DoPkt()` waits on `pr_MsgPort` with `WaitPort()` and removes the message through `GetMsg()`. If the caller is a task, the function even builds an exec `MsgPort` on the fly and waits on this temporary port — thus 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, this function aborts with a deadend 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()`; this 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 itself typically 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 prototypes are supplied that do not take all arguments. They all access the same entry 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 type of the packet `action` and 0 to 5 additional arguments which are filled into `dp_Arg1` upwards.

### 11.1.2   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)
         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 packet to return.

The `packet` shall be partially initialized; in particular, `dp_Link` shall point to an exec `Message` whose `mn_Node.ln_Name` field points back to `packet`. This function *does not* supply or initialize a suitable message, this is up to the caller.

`dp_Type` shall be filled with the type of the packet, i.e. an identifier specifying the type of activity requested from the handler, see section 12. Depending on this type, `dp_Arg1` through `dp_Arg7` shall be initialized with additional arguments.

`DosPackets` can be constructed in multiple ways; the `AllocDosObject()` function may be called to construct a `StandardPacket`. This is a structure that contains both the `Message` and the `DosPacket`. It is defined in `dos/dosextens.h`:

```
struct StandardPacket {
   struct Message   sp_Msg;
   struct DosPacket sp_Pkt;
};
```

`AllocDosObject()` ensures that the linkage between `Message` and `DosPacket` are properly initialized.

Another option is to use `AllocMem()` to allocate sufficient storage to hold a `StandardPacket` and initialize the structure appropriately:

```
struct StandardPacket *sp;


sp->sp_Msg.mn_Node.ln_Name = (UBYTE *)&(sp->sp_Pkt);
sp->sp_Pkt.dp_Link         = &(sp->sp_Msg);
```

### 11.1.3 Waiting for a Packet to Return

The `WaitPkt()` function waits on the message port of the calling process for a packet to return.

```
packet = WaitPkt()
D0

struct DosPacket *WaitPkt(void);
```

This function performs all activities to receive a message returning from a handler; it is also implicitly called by `DoPkt()` after sending the messages to the handler.

If the `pr_PktWait` pointer in the `Process` structure is set, `WaitPkt()` calls through this function to wait for the arrival of a message. Otherwise, the `WaitPkt()` calls `WaitPort()` to wait for a 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 corresponding 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.

### 11.1.4 Aborting a Packet

The purpose of the `AbortPkt()` funcktion 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)
          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.

### 11.1.5 Reply a Packet to its Caller

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.

```
ReplyPkt(packet, result1, result2)
           D1       D2       D3

void ReplyPkt(struct DosPacket *, LONG, LONG)
```

This function fills `dp_Res1` and `dp_Res2` of the `packet` with `result1` and `result2`, and sends the packet to `dp_Port`, the initiating port. Note that `mn_ReplyPort` of the message pointed to by `dp_Link` is ignored, i.e. packets do *not* follow the exec protocol for replying messages.

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 replied to a *dos.library* function.

## 11.2 Implementing a Handler

A handler or a file system is an Amiga process that retrieves commands in the form of `DosPacket`s. The main loop of a handler is conceptionally similar to a program implementing a graphical user interface, except that the latter retrieves messages via the *intuition* IDCMP system and works them off, whereas the handlers receive commands from the port in the `pr_MsgPort` of its process, or any other port they may provide to its clients.

### 11.2.1 Handler Startup

When a user of the *dos.library* attempts to access an absolute path, i.e. a path including a device name, the *dos.library* walks the device list (see section 7) to find the handler responsible for it by comparing the device name in the path with the `dol_Name` element of the device list. Once a suitable entry has been found, the *dos.library* checks whether a handler is already running by checking the `dol_Task` field of the `DosList`. If so, a packet corresponding to the called *dos.library* function is send to the port pointed to by `dol_Task`. The relation between library functions and packet types is further explained in subsequent sections.

If `dol_Task` is `NULL`, the *dos.library* checks next whether `dol_SegList` is `ZERO`. If it is, the handler code is not yet present and will be loaded from the file name indicated in `dol_Handler` through `LoadSeg()`, and its return code is used to initialize `dol_SegList`. Once the handler is present, a new process is created from the segment, and a startup packet is delivered to `pr_MsgPort` of the created handler process. The *dos.library* will not continue processing the initiating call until the handler replies to this packet.

How the startup packet is delivered depends on the `dol_GlobVec` element of the device list entry, see Table 28. For C or assembler handlers with a `dol_GlobVec` value of $-1$ or $-2$, the startup packet is delivered to the `pr_MsgPort` of the handler process. For all other values of `dol_GlobVec`, the startup packet becomes the first argument of the `START` entry at index 1 of the global vector of the process.

If the "fake" BCPL startup code from section 10.5.3 is used, this packet is delivered in register `a0` of the main function. The following code demonstrates this:

```
LONG __asm __saveds main(register __a0 struct DosPacket *pkt)
{
    SysBase = *((struct ExecBase **)(4L));
    struct Process *proc = (struct Process *)FindTask(NULL);
    struct Message *msg;
    const UBYTE *path;
    ULONG startup;
    struct DosList *dlist;
    LONG error = 0;

    /* 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);
        ... /* Handler initialization */

        if (error) {
          ReplyPkt(pkt,DOSFALSE,error);
          ... /* shutdown, terminate */
          return;
        } else {
          BOOL run = TRUE;
          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_....
             ....
           }
           ReplyPkt(pkt,res1,res2);
          } while(run);
        }
}
```

The startup packet is populated as follows:

**Table 62: 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 | DOSTRUE |
| dp_Res2 | 0 |

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. However, as its type is identical to ACTION_NIL, the startup packet may also be processed within the main handler loop.

dp_Arg1 is set to a BPTR to a BSTR representing the path name 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 argument is for example taken from the first argument to Open(), just that it has been converted to a BSTR by the *dos.library*. For the CON-Handler for example, this is the window specification that instructs the handler on the position, size and title of the console.

dp_Arg2 is a copy of the dol_Startup element of the DosList structure, see section 7. 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, it is typically, but not necessarily, a BPTR to a FileSysStartupMsg structure. 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 7.1.1 and section 7.1.2.

`dp_Arg3` is a BPTR to the `DosList` structure that triggered starting this handler.

Multiple strategies exist how handlers make use of this information. A file system process would typically handle multiple files at once through the same process. To ensure that the *dos.library* sends requests for the file system to the process just started, the *file system* shall place a pointer to the `MsgPort` incoming packets are supposed to be send to in `dol_Task` of the `DosList` structure received in `dp_Art3`. This is typically, but not necessarily, the `pr_MsgPort` of the handler process.

Whenever a client program opens a file on this file system or attempts to lock an object, the handler process will be contacted by sending a packet to the port listed in `dol_Task`.

A handler such as the `CON` handler requires a separate process for each window it manages. In such a case, `dol_Task` remains `NULL`. Thus, handlers can decide whether they require a new process for each file opened from the handler.

While the *dos.library* will also send a packet to open the file for which the handler was launched, the path of this second packet or any subsequent packet to open a file on the same process is not relevant to the CON-Handler anymore as the window will be opened during startup, and not during opening a file. Even though the handler is started once for the initial open and does not fill in its port in `dol_Task`, multiple requests to open a file may arrive at the handler, e.g. when opening the "`*`" file indicating the current console.

Once the handler or file system initiated itself from the startup packet, the 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 9.2.9 for a list of common error codes. Then, the handler shall release all resources acquired so far and terminate.

## 11.2.2  Handler Main Processing Loop

If startup succeeded, the primary result code for the packet shall be `DOSTRUE`, and the secondary result code shall be 0. Handler operations then proceed by waiting for incoming packets, and processing them one by another.

The *dos.library* keeps the device list locked while the handler is starting up. This means in particular that an attempt to gain access to the device list with `LockDosList()`, see section 7.3.1, can deadlock. Even if such a call is not made explicitly, *dos.library* functions can require implicitly such a lock, and thus attempting to access files or lock objects within handlers should be avoided, not only during startup.

If `dol_GlobVec` is $-2$ or $-3$, the *dos.library* will only acquire a shared lock, and thus will allow the handler to retrieve a shared lock on the device list as well.

Once started up, handlers or file systems should wait for incoming packets. Depending on how a handler is contacted, the *dos.library* uses multiple sources to identify a suitable port:

If the handler is contacted through the special path "`*`" or a path starting with "`CONSOLE:`", the port in `pr_ConsoleTask` is used to send packets to.

If the handler is contacted through a path starting with "`PROGDIR:`" and the `pr_HomeDir` element of the calling process is non-`NULL`, the `fl_Task` element of the lock stored there is used to contact the handler.

If the handler is contacted through an absolute path, the *dos.library* scans the device list to locate a `DosList` structure whose `dol_Name` element matches the device name in the path. The `dol_Task` field, if non-`NULL`, is then used as destination of a packet. If it is `NULL`, a new process of the handler is created. From this follows that a handler or file system can customize the port through which it expects packets. File systems will typically place the `pr_MsgPort` of their process there.

If a handler is contacted through a relative path, and `pr_CurrentDir` of the calling process is non-ZERO, the `fl_Task` element of the current directory of the process attempting to resolve a path is used as target port. If `pr_CurrentDir` is ZERO, the `pr_FileSystemTask` is used as target port.

If the handler is contacted through a file handle, for example to read or write bytes from a file, the `fh_Type` element of the handle is used as destination port. A file system may place a custom port in this element on opening files by `ACTION_FINDINPUT`, `ACTION_FINDOUTPUT` or `ACTION_FINDUPDATE` in the `fh_Type` element of the handle to receive all packets associated to this particular file on an alternative port. The *dos.library* places by default there the port it found through the path, see above.

If the handler is contacted through a lock, the `fl_Task` element of the `FileLock` structure is used to send out packets. A file system thus may customize a port through which all packets related to a particular lock are transmitted by placing this port in `fl_Task` when creating a lock.

The special path "`NIL:`" does not correspond to any handler but sets the `fh_Type` element to ZERO. Any attempt to write out data through such a handle ignores all bytes written, and any attempt to read bytes will return zero bytes. This is a special case in which the *dos.library* does not go through a handler at all.

The main loop of a handler then checks its own process port, or all ports it provided through the above mechanisms for incoming packets and tests their `dp_Type` field, identifying the requested action to be performed. Subsequent sections will provide information on all packets documented within AmigaDOS, though third-party handlers may implement additional packet types.

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 63: 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 |

This ensures that clients of the handler or file system will receive a result that is an indication of an error.

### 11.2.3   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 or disposed. 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. Once the use counter reaches zero, the handler process should die by releasing all of its resources and falling off its main function. This ensures that the system is not congested by creating more and more processes of the same handler that, effectively, cannot be contacted anymore because its process part is not referenced in any object passed out of the *dos.library*.

A typical example of a handler is the CON-Handler of the system that opens its window from its startup message but any file opened to it will interact with the same window. The window will be closed when each of these files had been closed[1].

---

[1]This is a simplification, ignoring `AUTO` and `WAIT` parameters, see section 11.3 for details

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 volume they manage have been released. Thus, in addition to such resource tracking, file systems should check for incoming packets of the type `ACTION_DIE` and then attempt to shutdown, see section 12.9.5.

The `MsgPort` or ports of such a handler may still contain packets that have not yet been worked on after `ACTION_DIE` has been received. 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 63.

Despite such precautions, `ACTION_DIE` cannot be implemented in a fully reliable way as the `MsgPort` of the file system could still be cached by some client programs even if no active locks or open file handles exist. Section 12.9.5 contains further details.

## 11.3 The CON-Handler

The CON-Handler implements the console of AmigaDOS. It not only serves the graphical console, but also the serial console. The AUX-Handler is only a minimal wrapper that locates the Csonole-Handler in the ROM and then initates it with a suitable startup packet. Therefore, it implements the `CON:`, `RAW:` and `AUX:` device.

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. 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.

### 11.3.1 CON-Handler Path for Graphical Consoles

The path through which a stream to the CON-Handler is opened determines for the graphical console the size, position and features of the window the console appears within; for the serial console, it defines connection parameters such as the baud rate and the parity. The path for the graphical console accessed through `CON:`, and for `RAW:` likewise are as follows:

```
CON:<left>/<top>/<width>/<height>/<title>[/<options>]
```

The `<left>` parameter determines the left edge of the window within which the console shall appear; it is measured in pixels. If this parameter is not present or negative, intuition is instructed to pick a default. This position is usually under the cursor position.

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. This is typically the width of the screen. If no parameters are present at all, i.e. the path is just `CON:`, 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.

The `<title>` parameter is the string that is put into the drag bar of the console window to create. 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 path separator[2], the backslash is escaped by itself, i.e. "\\" creates a single backslash in the title.

---

[2]Given the overall syntax of AmigaDOS, the asterisk $*$ would have been a more logical choice.

All following path components configure options for the graphical terminal; multiple of these parameters may be combined, separated by a forwards slash ("/") from each other:

`CLOSE` adds a close gadget to the window. Pressing on it creates an end-of-file indication, similar to the key combination Ctrl+\.

`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 write data to the console. Up to that point the console remains closed. If neither read requests nor write requests are pending, the close gadget will also close the window without queuing an end-of-file; the window will pop-open again as soon as input or output requests are made. Unfortunately, an `ACTION_DISK_INFO` will lock the window open, see section 12.8.3 for alternatives to this packet or how to regain the `AUTO` feature.

`WAIT` delays closing the window after every file handle to the console has been closed. With this flag, the window stays open until the close gadget has been pressed, even if no client is connected to the terminal anymore to receive input or perform output.

`SMART` enforces smart refresh of the window. If this is enabled, more RAM is required, but it is not necessary to re-render the contents of the console window if it is made visible behind another window. In most cases, this makes little difference and only increases the RAM footprint.

`SIMPLE` enforces simple refresh of the window. If hidden window contents are made visible again they 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 automatically the input focus when it is opened. The user has to click into it to type in the console.

`BACKDROP` instructs intuition to place the window behind all other windows on a screen. This works best in conjuction with `NOBORDER`, an empty window title, the `NOSIZE`, `NODRAG` and `NODEPTH` option and a console redirected to a public screen. If the window is made as large as the screen, the result will create 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 top 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.

`WINDOW` instructs the console to hijack an already open window and place the console within this window. This window will also be closed if the console closes. 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.

`SCREEN` provides a name of a public screen on which the console shall appear instead of the Workbench. The public screen name follows, optionally separated by spaces, e.g. `SCREEN myProgram.1`. If the screen name is `*`, then the frontmost public screen will be used as host for the console.

`ALT` provides alternative window dimensions and placement to which the window can be toggled by the top-left zoom gadget of the window. 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. This alternate placement shall follow directly behind the `ALT` keyword, e.g. `ALT 0,0,320,200`.

`ICONIFY` equips the window with an iconification gadget. This requires the `ConCip` program running in order to have an icon available for the console. 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 a window to read from or write data into. 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 12.8.3 for details.

### 11.3.2 CON-Handler Path for Serial Consoles

If the console is a serial console, e.g. mounted as AUX-Handler, another set of parameters becomes available that configures the serial connection; if the console is run on any other than the *serial.device* through mount parameters, this device shall support the `SDCMD_SETPARAMS` command through which these path parameters are installed.

```
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` mount option, and if that is not provided, the settings come from the serial preferences.

`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 "`1`" 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 configuration within which the performance of the `serial.device` is optimal. If this parameter is not present, 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:

`RAW` forces the console into the (non-cooked) raw mode within which input characters are not echoed and no line buffering takes place. The same can be reached with the mount parameter `STARTUP=1`.

`UNIT` defines the unit of the device over which the connection shall be made if multiple units exist. 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 through it or to read data from it. This works 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.

### 11.3.3 CON-Handler Line Buffer Modes

In addition to the window or serial parameters communicated to the path, the console is also configured through the `SetMode()` function, see section 4.9.3 and table 13. This mode sets the buffer mode of the console, regardless of the device the console runs on. In particular, `SetMode()` allows to convert a `CON:` window into a `RAW:` window, create an equivalent of a `RAW:` handler from an `AUX:` connection.

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 line editing functionalities such as cursor movements. Only when the user presses `RETURN`, an entire line of data enters the output buffer, and there becomes available for an `ACTION_READ`.

One particular difference between the V47 console and previous versions its predecessors is that the console no longer offers a history. Rather, the history functions have been moved to the Shell using the medium mode corresponding to buffer mode 2 described below.

In the cooked mode, and also in the medium mode, the CON-Handler interprets the following keystrokes:

**Table 64: Cooked Keyboard Sequences**

| 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+cusor 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* |
| 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+x | Erase entire line |
| Cltr+k | Kill characters into yank buffer |
| Cltr+q | Resume output |
| Ctrl+s | Suspend output |
| Ctrl+r | Do nothing* |
| Ctrl+u | Delete beginning of line |
| Cltr+w | Delete word |
| Cltr+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 65 | |

As seen in the table, some keystrokes operate differently in the medium mode; in that mode, they request a particular function from the shell by sending a "TAB Report" CSI sequence.

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 buffer mode 1 corresponds to a RAW: window and is denoted the *raw mode*. In this mode, every keystroke becomes available to client programs as it enters the output buffer immediately. Cursor movements thus become available as CSI sequences. In this mode, the console does not echo user input. The CSI sequences are those defined by the *console.device*, resp. the current keymap of the device.

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 the shell and forewards TAB-Expansion and history functions to the shell. While the console again provides line editing features and echoes most keys on the screen, some keyboard sequences create immediate output through a "TAB Report" CSI sequence that enters the output buffer immediately. These sequences are interpreted by the Shell and trigger there corresponding functions, such as expanding a pattern or working through the command history. A TAB Report sequence looks as follows:

```
CSI <m>;<s>;<c>U <line>
```

Here `CSI` is the ANSI "control sequence introducer", which has 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 12.8.7.

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.

The `U` is a literal character and identifies this sequence as TAB-Report. `<line>` is a copy of the current line that is edited by the user. For many sequences, this input is used as a template the shell to perform the requested action, e.g. for a TAB-expansion, it provides the entire input, though the command line argument within which the cursor is positioned is used to generate a wildcard the Shell uses to find possible candidates for its expansion.

The keystrokes that trigger such TAB Report sequences, along with their purpose, are as follows:

**Table 65: Medium Mode CSI Sequences**

| Keystroke | Sequence |
|-----------|----------|
| Cursor up | `<U>`=2, move upwards in the history |
| Cursor down | `<U>`=3, move downwards in the history |
| Shift+cursor up | `<U>`=4, search history upwards |
| Shift+cursor down | `<U>`=5, search history downwards |
| Ctrl+r | `<U>`=6 ,search history upwards, partial pattern |
| Ctrl+b | `<U>`=10,rewind history |
| TAB | `<U>`=12,iterate forwards through expansion |
| Shift+TAB | `<U>`=13,iterate backwards through expansion |

The only difference between the cases `<U>`=6 and `<U>`=4 is that the former only uses the line input buffer up to the cursor position as search pattern and ignores everything past it.

### 11.3.4   CON-Handler Startup Parameters

In addition to the window or serial parameters communicated to the path, the console is also configurd through the handler startup mechanism. It provides defaults for the serial console, but also define the line buffer mode of the console described in the previous 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; however, users can create custom mountlists for additional devices with custom mount parameters.

The handler startup mechanism is described in section 11.2.1, and in particular, `dp_Arg2` of the startup packet is a copy of the `dol_Startup` entry of the `DosList` created from the mountlist; details on this mechanism are described in section 7.1.2.

The following table specifies the values of `dol_Startup` the CON-Handler recognizes:

**Table 66: 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(). The buffer mode 3 is currently not used and reserved for future extensions.

The case of a negative dol_Startup value is a legacy startup mechanism of the console. In this confguration, dp_Arg1 of the startup packet (see section 11.2.1) 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 thus it is discouraged, but [10] provides example code how to trigger it. A better mechanism to redirect a console to a particular window is the WINDOW argument of the command path which is described in section 11.3.1.

In all other cases, dol_Startup is a BPTR to a FileSysStartupMsg described in more detail in section 7.1.2. The fssm_Device and fssm_Unit elements of this structure provide a device and unit the console shall run on. The AUX: handler uses this mechanism to create a FileSysStartupMsg to the CON-Handler which points to the *serial.device* unit 0, but also accepts custom mount parameters to mount it on other devices through the EHANDLER mount parameter. The DEVICE parameter of the mountlist is then extracted, and forwarded to the CON-Handler through a startup packet. Thus, a console can be generated on either type of device provided it supports CMD_WRITE to write data out and CMD_READ to receive keystrokes.

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.

To make the Mount command create a DosList corresponding to the latter case, the CON-Handler needs to be mounted as EHANDLER. The following is an example mountlist providing a serial console on a custom serial hardware:

```
EHandler = CON-Handler
Device   = duart.device
Unit     = 0
Baud     = 9600
Control  = "8N1"
```

The very same mountlist can be used with the AUX-Handler; if it is mounted as an extended handler, it will just take its mount parameters and forwards them to the CON-Handler. If mounted with the (regular) HANDLER keyword in the mountlist, it will create a FileSysStartupMsg corresponding to the *serial.device* and provide this to the CON-Handler. The AUX-Handler is therefore rather minimal.

## 11.4   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.

### 11.4.1 Port-Handler Path

Access to the Port-Handler is configurable through the path following the device name. The path looks as follows:

```
SER:[<options>]
```

The following keywords in `options` exist, where options are separated by a forwards slash ("/") from each other.

`BAUD` sets the baud rate, which is only relevant for serial connections. It specifies the number of bits per second transmitted over a serial connection. If this parameter is not present, the default from the serial preferences is used. 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`. Clearly, this parameter is only used by `SER:`.

`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. The string follows the same convention used for the `AUX:` terminal. The first digit is between "1" and "8", specifying the data bits, the second letter 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 digit between "0" and "2".

A typical control string is `8N1` indicating 8 data bits, no parity and 1 stop bit, corresponding to the fastest possible console. Optinally, this parameter may be introduced with the keyword `CONTROL` which shall be separated from the control string by spaces or an equals-sign, e.g. `CONTROL 8N1`.

If this parameter is not present, the default from the serial preferences is used. Obviously, this parameter is only used by `SER:` and ignored otherwise.

`TRANSPARENT` is a boolean switch that, if present, indicates that CSI sequences written over this device shall not be translated to the configured printer, but shall rather be send directly to the printer. By default, the `PRT:` device uses the AmigaDOS defined CSI sequences that are device independent. This option is only used by the `PRT:` device and ignored otherwise.

`RAW` is a boolean switch that, if present, disables translation of the AmigaDOS newline-character `0x0a` to `0x0a 0x0c` pairs, i.e. a line feed followed by carriage return. This switch only applies to the `PRT:` device and is not used otherwise.

`UNIT` takes a numeric argument and provides the device unit that will be opened. This makes most sense if the Port-Handler is not mounted on the standard exec devices, but a custom device is provided through the `DEVICE` keyword of a custom mountlist. The unit number is represented as decimal number separated from the keyword by spaces or an equals-sign, e.g. `UNIT 1`. This option is operational regardless of the underlying device.

`NOWAIT` is boolean switch that, if present, avoids blocking when reading from a serial connection. If no data is available at the serial port, a connection configured with `NOWAIT` will report an end-of-file condition. Otherwise, such a stream would block until data is available. This option is only supported for serial connections.

Another possibility to implement non-blocking reading from the serial port is to use `WaitForChar()` which is also supported by the Port-Handler. It blocks until either the specified timeout is reached, or input data is available on the port.

### 11.4.2 Port-Handler Startup

The Port-Handler is also configurable through a mountlist, though mount parameters for `PAR:`, `SER:` and `PRT:` are hard-coded within the Kickstart ROM. However, users may create custom mountlists and mount the handler under a custom name.

As for all handlers, it is customized through `dol_Startup`, transmitted in `dp_Arg2` in the handler startup packet. The following values are recognized:

**Table 67: Port Handler Startup Code**

| dol_Startup | Description |
|---|---|
| 0 | SER: serial output through the serial.device |
| 1 | PAR: parallel output through the parallel.device |
| 2 | PRT: printer output through the printer.device |
| BPTR | serial output with parameters from the FileSysStartupMsg |

The values 0 to 2 represent the standard handlers `SER:`, `PAR:` and `PRT:` mounted by the kickstart. They can be recreated by a `STARTUP=<n>` entry in a mountlist. For these options, the standard Amiga device names for the serial, parallel and printer interface is used.

The last option allows to customize the target exec device to run on. In this case, `dol_Startup` is a BPTR to a `FileSysStartupMsg` described in section 7.1.2. To make the `Mount` command create a suitable `DosList` entry, `EHANDLER` keyword is used, and a `DEVICE` and `UNIT` option can be provided. If the mountlist includes in addition a `BAUD` value, that is used as default value for the baud rate, which can be overridden by the Port-Handler path described in the previous section. The `CONTROL` parameter in the mountlist, if present, also control sthe serial settings. An example mountlist would look as follows:

```
EHandler = Port-Handler
Device   = duart.device
Unit     = 1
Baud     = 19200
Control  = "8N1"
```

## 11.5   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 entire output. 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.

Even though the Queue-Handler is disk-based, it is already mounted by the Kickstart-ROM as `PIPE:`, though additional instances under other device names can be mounted through the user.

### 11.5.1   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 as long as it uniquely identifies the pipe; the Shell constructs a pipe name from the process indicator and a second unique number that is incremented for each pipe used. The packet `ACTION_FINDINPUT` opens the reading end, and `ACTION_FINDOUTPUT` the writing end. The third open mode, `ACTION_FINDUPDATE` is not supported.

The empty name establishes a special case. It identifies a pipe that is already open as standard input or standard output of a process. In case a pipe is openend without a name, the Queue-Handler checks whether the standard input or standard output of the client process is already open to a pipe, and if so, this end is reused. This allows constructions such as

```
list | type PIPE: hex
```

where the reading end of the pipe is explicitly addressed by an empty pipe name as it is already established by the Shell as standard input of the `type` command. It is necessary here because the command line syntax of `type` does not include provisions to read from its input stream.

The pipe path may follow by options configuring the pipe, thus the complete path specification is as follows:

```
PIPE:name[/quantumsize[/buffercount]]
```

The `quantumsize` is the size of a block the Queue-Handler uses to buffer intermediate data. Written data is is queued up until 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. If all buffers 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. at most 8K of data can be waiting in a pipe.

### 11.5.2   Queue-Handler Startup

The Queue-Handler can also be configured through its startup packet. For this, `dol_Startup` in its `DosList` entry shall be a BPTR to a `FileSysStartupMsg` as specified in section 7.1.2. The device and unit found there are irrelevant, though the environment vector in `fssm_Environ` discussed in section 7.1.3 contains some relevant entries.

`de_SizeBlock` set the size of the Quantum, and thus corresponds to the default of the first optional argument in the path.

`de_NumBuffers` sets the number of such buffers for each pipe and corresponds to the default of the second optional argument of the path.

The Kickstart ROM does not provide a startup value and thus mounts the Queue handler with its defaults parameters. However, within a custom mountlist, they can be provided as follows:

```
EHandler        = Queue-Handler
BlockSize       = 1024
Buffers         = 8
```

The `BLOCKSIZE` keyword sets the quantum size, and the `BUFFERS` keyword the number of quantum buffers.

## 11.6   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 sections 12.1 and following, with the exception of record locking. The RAM-Handler provides one extended feature, namely external links, also explaned in section 6.4. Such a link points to a target outside of the RAM disk; while such objects are initially empty, their contents will be made accessible by copying the objects within the link target to the RAM disk.

This feature is used within AmigaDOS to realize the `ENV:` assign which contains the system preferences. This allows to save RAM by only keeping those files in `RAM:` that are actually accessed.

The RAM handler does not interpret any arguments in its startup packet and cannot be configured through it.

## 11.7   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 hostadapters are typically able to load an updated version of this (and other) file

systems from the *rigid disk block* of the boot disk and make it available to the system. The *System-Startup* module will make such updated version then also available for all other devices.

The same code serves multiple flavours of the file system that differ in the structure and organization of on disk. Table 29 in section 7.1.3 provides an overview on the available flavours.

As a full hierarchical file system, the FFS supports all file-system relevant packets listed from section 12.1 onwards; packets specific to interactive handlers listed in section 12.8 are an exception and not implemented, and the `ACTION_SET_OWNER` packet is neither implemented and remains exclusive to to multi-user network file systems. The FFS does support record locking, soft- and hard links and notification requests. External links remain currently exclusive to the RAM-Handler and are neither supported. Latest versions of the FFS can also be shut down by `ACTION_DIE`.

### 11.7.1   Configuring the FFS

The FFS is customized through multiple sources. First of all, the `DosList`, the `FileSysStartupMsg` pointed to by this structure, and the environment vector in the latter. Section 7.1 provides a list of the configurable options and the corresponding entries in the mountlist.

In some situations, however, the FFS will override the parameters found in the above sources, and therefore parameters from 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 by a `TD_GETGEOMETRY` when validating an available medium. The same logic can also be enabled 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 7.1.3.

Upon detecting device parameters, data from the device provided `DriveGeometry` structure documented in `devices/trackdisk.h` are copied into the environment vector of the file system. The following algorithm is used to adjust the data in the environment vector:

```
void AdjustEnv(struct DosEnvec *env,
               const struct DriveGeometry *dg) {
    env->de_SizeBlock      = 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 though the mountlist or the rigid disk block. Ideally, of course, any memory type should be supported and this workaround should not be necessary.

The number of blocks kept in the file system cache is read from `de_NumBuffers` in the environment vector, but `ACTION_MORE_CACHE` can be used to adjust this parameter anytime.

### 11.7.2   The Boot Block

The FFS flavour is initially read from `de_DosType` element of the environment vector, see section 7.1.3. 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.

During formatting, the flavour is taken from `dp_Arg2` of the `ACTION_FORMAT` packet. If `dp_Arg2` is none of known FFS flavours from table 29, the currently active flavour remains in force and is used to

initialize the disk structure. If the FFS did not adjust its flavour from accessing a previous volume, this remains the value from `de_DosType`.

During disk insertion, or if a volume becomes accessible to the FFS by other means such as mounting it or uninhibiting a volume through `ACTION_INHIBIT`, the FFS reads the flavour 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

$$\text{BootBlock} = \text{de\_LowCyl} \times \text{de\_Surfaces} \times \text{de\_BlocksPerTrack}$$

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, at least for floppy disks, for bootblock booting which is described in mored detail in [8].

### 11.7.3 Disk Keys and Sectors

Except for the boot block, the FFS does not address sectors through their physical sector number, but due to their *key*, which addresses data on disk relative to the start of the partition in blocks, and not sectors. The relation between the physical sector $S$ and the key $K$ is given by the following equation:

$$S = \text{de\_LowCyl} \times \text{de\_Surfaces} \times \text{de\_BlocksPerTrack} + K \times \text{de\_SectorPerBlock}$$

In particular, the boot block has therefore the key 0. The sector number is used directly for the "direct SCSI" transfer which is enabled by the `ENVF_SCSIDIRECT` flag in the `de_Interleave` element of the environment vector.

For the default trackdisk-like transfer, exec device drivers expect a byte offset instead of a sector, which is computed by multiplying $S$ by $\text{de\_SizeBlock} \times 4$. The additional factor of 4 is because `de_SizeBlock` is in units of long-words 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 two here. Thus, any multiplication or division by these numbers are easily implementable as shifts.

### 11.7.4 The Root Block

The Root Block represents the root directory of a volume and contains the creation date and the volume name. It is placed in the middle of the volume at key

$$\left\lfloor \frac{\text{de\_Reserved} + \left\lfloor \frac{(\text{de\_HighCyl} - \text{de\_LowCyl} + 1) \times \text{de\_Surfaces} \times \text{de\_BlocksPerTrack}}{\text{de\_SectorsPerBlock}} \right\rfloor - 1}{2} \right\rfloor$$

Note that the element name `de_BlocksPerTrack` is actually misleading as this the size of a track in (physical) sectors and not in (logical) blocks or keys. The division by `de_SectorsPerBlock` computes from the sector index a key. The brackets $\lfloor \cdot \rfloor$ indiciate rounding down to the next integer.

The structure of the root block is as follows:

**Table 68: 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 | |
| L-22 | Mins | timestamp of last directory change |
| 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 | |
| L-9 | Mins | timestamp of last volume change |
| L-8 | Ticks | |
| L-7 | Days | |
| L-6 | Mins | volume creation time |
| 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 BPCL 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 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 element extends over multiple long-words. In particular, the list of hash-keys is a variably sized number of long-words large, it always extends over $L - 56$ longs.

The block type is indicated by `Type` and `SecType`. For the root block, these elements shall be 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 chosen 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 key* that is derived from its name. As multiple objects can have the same hash and thus hash conflicts may 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.

How the FFS computes the hash key from the name depends on the FFS flavour. 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)
{
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 element 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 characters in the latin-1 character 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. It therefore also works correctly for this encoding.

`BMFlag` is `DOSTRUE` in case the bitmap is valid. If the quick validation on disk insertion finds that this element is 0, then the disk status is set to non-validated and the bitmap is recomputed through a complete disk scan. The bitmap keeps information on which blocks of the volume are allocated and which are free.

`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 remain 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.

The elements at offsets $L - 23$ to $L - 21$ form a `DateStamp` structure as specified in section 3. The date and time there indicate the last change within the root directory.

`Name` is the volume name, encoded as BSTR; the first character is therefore the size of the name. This element is 8 long-words large and thus limits the volume name to 30 characters (one extra character is reserved, even though there would be sufficient space for 31 characters).

The elements at offsets $L - 10$ to $L - 8$ form a `DateStamp` structure that is updated on every change of the volume.

The elements 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.

`DCache` is the key of the directory cache list of blocks. It is only used for the flavours of the FFS that utilize such a cache.

Unlike user directories, the root block lacks a list of hard-links that points to it. This has the consequence that the FFS does not allow to create hard-links to the root directory of a volume.

### 11.7.5   The User Directory Block

The user header block represents a sub-directory of the volume root, or another user directory. 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 flavour of the FFS. Table 69 applies to all FFS variants except the long-file variants `DOS\06` and `DOS\07`.

**Table 69: FFS 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 6.1 |
| L-47 | 0 | reserved for future use |
| L-46 | Comment | directory comment as BSTR |
| L-26 | 0 | reserved for future use |
| L-23 | Days | |
| L-22 | Mins | timestamp of last directory change |
| 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, setthis 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 block number 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 element, is zero.

Hash contains for user directories all the hash keys of the file system objects contained within.

OwnerID is reserved for the owner ID of this directory that can be set with ACTION_SET_OWNER. As the FFS has no means to validate the directory owner, this element does not bear any practical meaning.

`PrtBits` are the protection bits of this directory. The FFS actually ignores the protection bits on the directory, but stores the value here anyhow.

`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 elements at offset $L - 23$ to $L - 21$ form a `DateStamp` structure 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 here, 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` flavour 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 for 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 then is converted from a link to a directory.

`NxtHash` is the key of the next object using the same hash key as this directory itself.

`Parent` is the key of the parent directory, or the key of the root block in case this directory is directly in the volume root.

`DCache` is the key of the first directory cache block. This key is only used for FFS flavours with directory caching enabled. Otherwise, it stays 0.

`SecType` along with `Type` identifies the type of this block. The value of this element shall be 2, identifying this as a user directory block.

For the long file-name enabled flavours of the FFS, namely `DOS\06` and `DOS\07`, this block looks somewhat different:

**Table 70: 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 6.1 |
| 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 | |
| L-14 | Mins | timestamp of last directory change |
| 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 | DCache | key of directory cache block if used |
| L-1 | SecType | shall be 2, setthis 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 field is 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 element 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 element, an additional comment block is created keeping only the comment, and this element then only keeps the file name.

CmtBlk is the key of the comment block, keeping the file comment if the NaC element is too short. If no comment block is needed, this element is 0.

### 11.7.6   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 flavour of the FFS. Table 71 applies to all FFS variants except the long-file variants DOS\06 and DOS\07.

**Table 71: 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 6.1 |
| 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 | |
| L-22 | Mins | timestamp of last file change |
| 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 | 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 block number 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 $L-49$. Probably the key at this offset was historically used for sequential access into the file, whereas the block list at offset $L-6$ following were 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 element, 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. $L-49$ contains the key of first data block, $L-50$ the second key and so on. If this array overflows, additional blocks are in one or more file extension blocks chained at $L-2$.

OwnerID is reserved for the owner ID of this file that can be set with ACTION_SET_OWNER. As the FFS has no means to validate the file owner, this element does not bear any practical meaning.

PrtBits are the protection bits of the file, encoded as in section 6.1. 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 elements at offset $L-23$ to $L-21$ form a DateStamp structure that identifies the timestamp of the last change of 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 flavour 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 key as this directory itself.

Parent is the key of the directory containing this file, or the key of the root block in case this file is directly 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 element shall be -3, identifying this as file header block.

For the long file-name enabled flavours of the FFS, namely `DOS\06` and `DOS\07`, this block looks somewhat different:

**Table 72: 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 6.1 |
| 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 | |
| L-14 | Mins | timestamp of last file change |
| 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 2, setthis is the BCPL constant ST.USERDIR |

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 field is 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 element 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 element, an additional comment block is created keeping only the comment, and this element then only keeps the file name.

`CmtBlk` is the key of the comment block, keeping the file comment if the `NaC` element is too short. If no comment block is needed, this element is 0.

### 11.7.7 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 flavour of the FFS. Table 73 applies to all FFS variants except the long-file variants DOS\06 and DOS\07.

**Table 73: 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 6.1 |
| L-47 | 0 | reserved for future use |
| L-46 | Comment | link comment as BSTR |
| L-26 | 0 | reserved for future use |
| L-23 | Days | |
| L-22 | Mins | timestamp of link creation |
| 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 block number 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 element, is zero.

Target is the path of the link target for soft-links. This is stored as NUL-terminated C-string, not as BSTR[3]. 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 this path is too long, though in a default floppy with 512 bytes per sector, this element leaves room for paths up to 287 characters.

---

[3] The information in [1] that this is a BSTR is incorrect

`OwnerID` is reserved for the owner ID of this link that can be set with `ACTION_SET_OWNER`. As the FFS has no means to validate the link owner, this element does not bear any practical meaning.

`PrtBits` are the protection bits of the link, encoded as in section 6.1. However, the practical value of these protection bits is 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 `ACTION_EXAMINE_NEXT` or `ACTION_EXAMINE_ALL`.

`Comment` contains a potential comment for this link. 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. Comments of links *can* be set with `ACTION_SET_COMMENT` as the FFS does not attempt to follow the link.

The elements 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 element bears no practical meaning as even an `ACTION_SET_DATE` will update the date of the link target and not 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 linked target. Thus, this date may potentially be seen when walking a directory, but it cannot be changed.

`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.

`NameX1` and `NameX2` are name extensions that are used by the `DOS\08` flavour of the FFS. They add additional 24 bytes of storage for the directory name. Again, one character is reserved, limiting the maximum name 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. For soft-links, this element 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`.

`NxtHash` is the key of the next object using the same hash key as this directory entry.

`Parent` is the key of the directory containing this link, or the key of the root block in case this file is directly in the volume root.

`SecType` along with `Type` identifies the type of this block. This can be either $-4$, corresponding to `ST.LINKFILE` for a hard-link to a file, or 4 (`ST.LINKDIR`) for a hard-link to a directory, or 3, corresponding to `ST.SOFTLINK` for soft-links.

For the long file-name enabled flavours of the FFS, namely `DOS\06` and `DOS\07`, this block looks somewhat different:

**Table 74: 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 6.1 |
| 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` | |
| L-14 | `Mins` | timestamp of last directory change |
| 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 field is replaced by a new field at offset $L - 46$.

`NaC` contains both the link name and file comment as two BSTRs, directly placed next to each other. The link name comes first, followed by the comment. This element 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 element, an additional comment block is created keeping only the comment, and this element then only keeps the file name.

`CmtBlk` is the key of the comment block, keeping the file comment if the `NaC` element is too short. If no comment block is needed, this element is 0.

### 11.7.8   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 75: 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 BPCL 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 occupied 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 element, 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. $L - 49$ contains the key of first data block referenced in this extension block, $L - 50$ the second key and so on. If this array overflows, additional blocks are in another file extension block chained at $L - 2$.

`Parent` is the key of the file header block of the file whose data block keys are extended by this block.

`FileExt` is the key of the next file extension block if this block is not sufficient to keep all data block keys. Otherwise, it is 0.

`SecType` along with `Type` identifies the type of this block. The value of this element shall be -3 (ST.FILE), identifying this block as belonging to a file.

### 11.7.9   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 76: Bitmap Extension Block**

| Long-word Offset | Content | Notes |
|---|---|---|
| 0 | BMKeys | additional $L - 1$ bitmap keys |
| L-1 | BMNext | key of another bitmap extension block |

`BMKeys` is an array of $L - 1$ keys, each of 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.

### 11.7.10   The Comment Block

This block keeps a file comment for the long filename FFS flavours `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 file comment.

**Table 77: 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 element, is zero.

Comment is the comment, stored as a BSTR with the first character containing its size. This element 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.

### 11.7.11  The Directory Cache Block

This block type is only used by the directory cache flavours 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 78: 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 79. 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.

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 element, 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 79: Directory Cache Entry**

| Size | 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 | |
| 16 | Mins | timestamp of last change |
| 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 elements of this structure are as follows:

Key is the key of the file header block, the user directory block or the link block, depending on the file system object to which this entry belongs.

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 ACTION_SET_OWNER packet. However, as the FFS has no means to verify access rights to an object, the purpose of this field remains opaque.

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 element. This is sufficient if the DateStamp() structure is normalized, i.e. each element is as small as possible.

SecType is a copy from the SecType element 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 element.

Name is a copy of the Name element 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 element 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, not a C string, and there is no NUL-termination.

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; that is, 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 element 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.

### 11.7.12   The Data Block

Data blocks contain the payload data of files. It comes in two variants: The "OFS" variants of the FFS, namely DOS\00, DOS\02, DOS\04 and DOS\06 keep a lot of redundant information within the data block that makes the file system structure very robust against media corruption; however, this information needs to be stripped off before the payload data is delivered to the FFS client, these variants are very 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. If the host adapter offers DMA, the CPU is not even 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 are generally recommended. The OFS flavours are therefore only useful for slow and unreliable data carriers.

The following table describes the structure of an OFS data block:

**Table 80: 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 BCPL 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 element, is zero.

Data is the actual payload data of the block. It consists of Size arbitrary bytes.

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 administrational 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 adminstration blocks that were, actually, data blocks allocated for a file.

### 11.7.13   The Bitmap Block

Bitmap blocks keep a bitmap — one bit per key — which keys are already occupied for administrational or payload data, and which keys are still free. Depending on the size of the volume, one or many bitmap blocks exist, sometimes even so many that bitmap extension blocks are needed as the 25 keys available in the root block.

The structure of a bitmap block is as follows:

**Table 81: Bitmap Block**

| Long-word Offset | Content | Notes |
|---|---|---|
| 0 | Chksum | LW sum over block is 0 |
| 1 | Bitmap | bitmap of avaiable blocks |

`Chksum` is the checksum over the entire block. It is chosen such that the long-word sum over the entire block, including this element, 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 flavour 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, 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);

/* 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;

  /* 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 + 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[BMExt]);
   /* Follow the link chain of extension
   ** blocks to find the right one
   */
   while(extension > 0) {
     block   = readKey(block[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 */
}
}
```

### 11.7.14  The Deleted Block

The FFS also mark 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 82: 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 12

# Packet Documentation

Packets are the mechanism by which the *dos.library* communicates with file systems and handlers. Their structure is specified in section 11.1. The activity a handler is requested to perform is defined through the packet type, contained in the `dp_Type` element of the handler.

This chapter documents all packets the AmigaDOS handlers and file systems support. Third party handlers may support additional packets. It is permissible to implement additional packet types as long as they do not conflict with existing packets. In particular, the range from 2050 to 2999 is available for third party handler implementations.

## 12.1 Packets for File Interactions

The packet types listed in this section are used to implement file-specific functions, such as those listed in chapter 4. The arguments of the packets typically follow the calling conventions of the *dos.library* functions closely, though are typically represented in their BCPL equivalents, i.e. BPTRs instead of regular pointers or BSTRs instead of `NUL`-terminated C strings.

### 12.1.1 Opening a File for Shared Access

The packet `ACTION_FINDINPUT` initializes a `FileHandle` structure for shared access to a file.

**Table 83:** `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 file name |
| `dp_Res1` | Boolean result code |
| `dp_Res2` | Error code |

This packet is used by the `Open()` function of the *dos.library* where the packet type is taken copied from the second argument, i.e. the packet type is identical to the `accessMode` argument of `Open()`, and thus `ACTION_FINDINPUT` is identical to `MODE_OLDFILE`.

`dp_Arg1` is initialized to 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 it operates on. This is particularly important to file systems that handle multiple files by a single process. In a typical implementation, this opaque value may be the (internal) lock to the opened file.

A handler or file system may also replace the `fh_Type` element of the `FileHandle` structure request that packets concerning this particular file shall be delivered to an alternative port. This element otherwise defaults to the process port of the handler.

`dp_Arg2` is initialized to the `BPTR` to a `FileLock` structure describing the directory within which the file to be opened is located. Typically, this corresponds to the current directory of the caller of `Open()`. 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. Flat file systems or handlers may ignore this argument.

`dp_Arg3` is initialized to a `BPTR` to a `BSTR` providing the path of the file relative to the directory provided by `dp_Arg1`.

All other arguments of the packet shall be ignored. The handler should locate the file indicated by `dp_Arg3` relative to `dp_Arg2`, and fill in a boolean success code in `dp_Res1`. It shall be set to `DOSTRUE` for success or `DOSFALSE` for error. `dp_Res2` shall be set to an error code from `dos/dos.h` or 0 for success.

The purpose of this packet is to prepare a `FileHandle` structure for shared access, allowing both reading and writing. If the file cannot be located, this packet *shall not* create it but rather fail with the error code `ERROR_OBJECT_NOT_FOUND`. This packet is used by the `Open()` function if its second argument is set to `MODE_OLDFILE`.

Handlers such that the Port-Handler or the CON-Handler may already open its resources as part of the startup-packet handling and thus may not perform a lot of activities here. Note that both handlers do not initialize `dol_Task` of the `DosList` structure and thus each opened file will launch a new process. The FFS, however, runs on a single process and thus distinguishes its files through fh_Arg1 in the `FileHandle` structure.

Note that both the Lock provided by `dp_Arg2` and the path `dp_Arg3` are required for locating a file on a hierarchical file system. The path from `dp_Arg3` is logically "appended" to the path implied 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 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 to locate a file that ends walking the directory tree whenever `dp_Arg3` ends.

### 12.1.2 Opening a File for Exclusive Access

The packet `ACTION_FINDOUTPUT` initializes a `FileHandle` structure for exclusive access to a file and potentially creates the file if it does not yet exist.

**Table 84:** `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 file name |
| dp_Res1 | Boolean result code |
| dp_Res2 | Error code |

This packet is used by the `Open()` function of the *dos.library* where the packet type is taken copied from the second argument, i.e. the packet type is identical to the `accessMode` argument of `Open()`, and thus `ACTION_FINDOUTPUT` is identical to `MODE_NEWFILE`.

The arguments of the packet are initialized as for `ACTION_FINDINPUT`, see section 12.1.1.

### 12.1.3 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 85:** `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* where the packet type is taken copied from the second argument, i.e. the packet type is identical to the `accessMode` argument of `Open()`, and thus `ACTION_FINDUPDATE` is identical to `MODE_READWRITE`.

The arguments of the packet are initialized as for `ACTION_FINDINPUT`, see section 12.1.1.

### 12.1.4 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 86:** `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 5.2.3. It opens a file from a lock. On a hierarchical file system, this is equivalent to an `ACTION_FINDINPUT` packet with `dp_Arg3` set to an empty string.

To uniquely identify the file handle and resources associated to it later on, the handler may place an identifier or a handle or pointer to internal resources in the `fh_Arg1` element of the `FileHandle` provided by `dp_Arg1`. 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, and to an error code from `dos/dos.h` otherwise.

### 12.1.5 Closing a File

The packet `ACTION_END` releases all *file system* internal resources of a file handle.

**Table 87:** `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 |

`dp_Arg1` is initialized to 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, for example any implicit lock on the file system object.

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.

The *dos.library* uses this packet to implement the `Close()` function. If it receives an error code, the library will not release the memory of the `FileHandle` structure, and as such, it remains available to the caller of `Close()` to perform other activities on the file.

### 12.1.6   Reading from a File

The packet `ACTION_READ` reads data from a *file system* or handler and advances the file pointer accordingly.

**Table 88:** `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*. The elements of the packet are initialized 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. Note that it is *not* the `FileHandle` itself; instead, the handler may create and insert such an identifier into the `FileHandle` when opening a file.

`dp_Arg2` is a pointer (not a BPTR) to the buffer to be filled.

`dp_Arg3` is the number of bytes to read.

Before replying this packet, the handler shall fill `dp_Res1` with the number of bytes that could be transferred into the buffer, or −1 for an error condition. This number may also be 0 if no data could be transferred, either because the end of file has been reached, or because currently no data is available on an interactive file, such as the console or the serial port. 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 4.7. 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.

### 12.1.7   Writing to a File

The packet `ACTION_WRITE` writes data to a file system or handler and advances the file pointer accordingly.

**Table 89:** `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*. The elements of the packet are initialized 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 contaiing the data to be transferred.

`dp_Arg3` is the number of bytes to write.

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 4.7. 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.

### 12.1.8  Adjusting the File Pointer

The packet `ACTION_SEEK` sets the file pointer relative to a base location.

**Table 90:** `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 11 |
| dp_Res1 | Previous file position |
| dp_Res2 | Error code |

This packet implements, to a major degree, the `Seek()` function of the *dos.library*. The elements of the packet are initialized 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` defines the new location of the file pointer relative to a position identified by `dp_Arg3`.

`dp_Arg3` defines the position to which `dp_Arg2` is relative. It is one of the modes in table 11 and is therefore identical to the third argument of `Seek()`.

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.

While this packet type mostly implement the `Seek()` function, the latter is also aware of the (caller-side) buffer of the `FileHandle` and performs a flush of this buffer. The packet cannot, of course, perform a flush.

### 12.1.9  Setting the File Size

The packet `ACTION_SET_FILE_SIZE` adjusts the size of a file, either truncating or extends it beyond its current end of file.

**Table 91:** `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 11 |
| dp_Res1 | Boolean result code |
| dp_Res2 | Error code |

This packet implements the `SetFileSize()` function of the *dos.library*. The elements of the packet are initialized 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` 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`.

`dp_Arg3` defines the position to which `dp_Arg2` is relative. It is one of the modes in table 11 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.

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.

Additional information on this packet is found in section 4.6.5 which describes the *dos.library* function that is based on it.

### 12.1.10   Locking a Record of a File

The packet `ACTION_LOCK_RECORD` locks a record of a file. Such a record lock neither provides reading nor writing to the locked region, it just prevents locking the same region with an exclusive lock, see section 4.10 for details how the protocol is supposed to use.

**Table 92:** `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 14 |
| 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. The file is identified by `dp_Arg1` which is a copy from `fh_Arg1` of the `FileHandle` structure[1]. There is no packet corresponding to `LockRecords()`. Instead, the latter locks records sequentially.

The record to lock is identified by `dp_Arg2` and `dp_Arg3` which correspond to the `offset` and `length` arguments of the `LockRecord()` function.

---

[1]The information on `dp_Arg1` in [1] is incorrect.

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 14, and more information on the modes is found in section 4.10.1.

If the mode from dp_Arg4 includes a timeout the file system should wait for the record to become available, it is provided by dp_Arg5. Otherwise, this argument is ignored, see table 14.

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 to obtain the record lock, dp_Res1 shall be set to DOSFALSE and dp_Res2 to an error code.

### 12.1.11  Release a Record of a File

The packet ACTION_FREE_RECORD releases a record lock on a portion of a file.

**Table 93:** 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.

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.

## 12.2  Packets for Interacting with Locks

The packets listed in this section implement the *dos.library* functions listed in section 5; they create, release or duplicate locks or create directories.

### 12.2.1  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. The object can later be used as active directory, to open a file from it or to retrieve metadata such as comments or protection bits associated to the identified object.

**Table 94:** 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 as a lock. As such, it implements the `Lock()` function of the *dos.library*.

`dp_Arg1` is a BPTR to a `FileLock` structure that represents the object to which the path given in `dp_Arg3` is relative. This is usually a directory, and the `Lock()` function fills it from the current directory of the calling process. This argument can be `ZERO` in which case the file system shall assume that `dp_Arg3` is relative to the root of the currently inserted volume handled by the file system.

`dp_Arg2` is the type of the lock to be created as defined in table 15. The value `SHARED_LOCK` requests a non-exclusive lock on an object; multiple of such locks can be held on the same object. The value `EXCLUSIVE_LOCK` requests an exclusive an object; only a single exclusive lock can be held on an object at once, and no other locks, including shared locks, are permissible. Attempting to exclusively lock an object that is already locked shall fail, and attempting to lock an object that is already exclusively locked shall as well. Unfortunately, some programs call `Lock()` with an invalid argument for the mode, and thus file systems should be prepared to handle invalid values for `dp_Arg2`. Such values should be considered equivalent to `SHARED_LOCK`.

`dp_Arg3` is a BPTR to a BSTR of an absolute or relative path of the object to be locked. Relative paths are relative to the lock in `dp_Arg1`, and if this argument is `ZERO`, relative to the root — the topmost directory — of the volume currently managed by this file system. If this string is empty, the identified object to be locked is identical to the one identified by `dp_Arg1`.

On success, `dp_Res1` shall be a BPTR to a `FileLock` structure. These objects are created and maintained by the file system. This structure shall be initialized as indicated in section 5.2.5:

`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 section 7.

`fl_Access` shall be filled by `dp_Arg2`, identifying the type of the lock.

`fl_Link` may be used to queue up locks on a volume that is currently not inserted. When a volume with locked objects is removed from the drive, the file system may store all these locks in a singly linked list starting at `dol_LockList` 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 manage them instead. 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`, finally, 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 9.2.9.

### 12.2.2 Duplicating a Lock

Despite its misguiding name, the packet `ACTION_COPY_DIR` makes a copy of a (shared) lock.

**Table 95:** 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* and attempts to replicate the lock passed in `dp_Arg1`. If this argument is `ZERO`, the packet is supposed to create a lock on the root directory of the currently mounted 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 [1] indicates that `dp_Res1` may be `ZERO` on an attempt to replicate the `ZERO` lock, this is not advisable as most users of this packet may misinterpret this result as failure case.

This packet is identical to `ACTION_LOCATE_OBJECT` with `dp_Arg2` set to `SHARED_LOCK` and `dp_Arg3` set to an empty string.

### 12.2.3 Finding the Parent of a Lock

The packet `ACTION_PARENT` obtains a shared lock on the directory containing a locked object.

**Table 96:** 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* and attempts to create a 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.

Otherwise, this function returns on success a BPTR to the `FileLock` representing the parent directory in `dp_Arg1` and sets `dp_Arg2` to 0. On error, `dp_Res1` is set to `ZERO` and `dp_Arg2` to an error code.

### 12.2.4 Duplicating a Lock from a File Handle

The packet `ACTION_COPY_DIR_FH` creates a shared lock from an object opened by a file handle; that is, it copies the implicit lock of the handle.

**Table 97:** 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*, and thus creates a copy of the lock that is implicit to an open file handle. This works only if the file is open in a non-exclusive mode, i.e. either `MODE_READWRITE` or `MODE_OLDFILE`.

`dp_Arg1` is a copy of the `fh_Arg1` element of the `FileHandle` structure and thus serves to identify the opened file.

On success, `dp_Res1` is filled by the BPTR to the `FileLock` created, namely a lock to the object opened by the file handle. In such a case, `dp_Res2` is set to 0. On error, `dp_Res1` is set to `ZERO` and `dp_Res2` to an error code.

### 12.2.5 Finding the Parent Directory of a File Handle

The packet `ACTION_PARENT_FH` creates a lock on the directory containing a file identified by a file handle.

**Table 98:** `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* and as such creates a lock on the directory that contains a given file handle. `dp_Arg1` is a copy of the `fh_Arg1` element of the `FileHandle` structure and thus identifies the handle.

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.

### 12.2.6 Creating a new Directory

The packet `ACTION_CREATE_DIR` creates a new directory and represents it by an exclusive lock on the directory created.

**Table 99:** `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* and creates a new directory of the name `dp_Arg2` within the directory identified by the lock `dp_Arg1`.

On success, it creates a new exclusive lock which is returned 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.

### 12.2.7 Comparing two Locks

The packet `ACTION_SAME_LOCK` compares two locks on the same file system and checks whether the two locks are on the same object.

**Table 100:** `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*, which, however, first checks whether the two locks to compare are on the same file system. Only if so, the file system corresponding the

two locks is contacted with the above packet to compare the two locks. For this, 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[2].

### 12.2.8 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 101:** ACTION_CHANGE_MODE

| DosPacket Element | Value |
|---|---|
| dp_Type | ACTION_CHANGE_MODE (1028) |
| dp_Arg1 | Object type from table 16 |
| 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 thus changes the access mode of files or locks to exclusive or shared access mode.

dp_Arg1 identifies the object whose mode is to be changed. This is a value from table 16, i.e. either CHANGE_LOCK to adjust the type of the 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[3]. 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. This packet shall then 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.

### 12.2.9 Releasing a Lock

The packet ACTION_FREE_LOCK releases a lock on a file system.

---

[2]The documentation in [1] on dp_Res1 is incorrect.
[3]The information in [1] is incorrect on this subject matter.

**Table 102:** `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()` 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.

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 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, but adjusts `IoErr()` according to `dp_Res2`.

## 12.3 Packets for Examining Objects

The packets in this section implement the functions of section 6.1 on packet level, i.e. they retrieve information from the file system on the metadata of objects and allow to scan all objects of a directory. Unfortunately, the packets in this section are the hardest to implement in a robust way as the directory contents can change while a directory scan is active, e.g. because files or directories are created or deleted.

### 12.3.1 Examining a Locked Object

The `ACTION_EXAMINE_OBJECT` packet retrieves metadata such as file name, comment and protection bits from a lock on an object on a file system and fills a `FileInfoBlock` with this data.

**Table 103:** `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* and retrieves metainformation on the object represented by the lock in `dp_Arg1`. This information is provided in a `FileInfoBlock` structure that is documented in section 6.1.

However, as the AmigaDOS handler design is due to historic reasons 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*.

The elements `fib_DirEntryType` and `fib_EntryType` shall be filled with the same value as some programs check one and others the other element. In particular, the value filled in shall be taken from table 21. The value 0 shall be avoided as some programs check for directories by testing against a positive value whereas some others check for a non-negative value, i.e. accept 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; they are probably an opaque value. `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.

This packet is also used to start a scan over the contents of a directory. In such a case, the examined object is the directory itself, i.e. the lock in `dp_Arg1` is on a directory. As such, the file system should prepare for a scan over this directory and potentially initialize an internal state machine.

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 inserted into the `FileInfoBlock` given by `dp_Arg2`. In that case, `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.

### 12.3.2   Scanning Directory Contents

The `ACTION_EXAMINE_NEXT` continues a scan over the directory contents and delivers meta-information on the 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 104:** `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`. Meta-information on the object itself is copied into the `FileInfoBlock` structure in the BPTR in `dp_Arg2`. Similar to `ACTION_EXAMINE_OBJECT` discussed in 12.3.1, 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*.

Unlike Unixoid 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. When parsing paths, the current directory is identified by the empty string, and the parent directory by the isolated slash ("/").

Unfortunately, this packet is one of the hardest to implement in a file system as the directory may change while the scan is active. 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_OJBECT`. 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 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`.

### 12.3.3 Examining Multiple Entries at once

The `ACTION_EXAMINE_ALL` packet scans a directory supplying multiple entries at once, potentially filtering entries through a pattern or through a hook.

**Table 105:** `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 struct ExAllControl |
| dp_Res1 | Continuation flag |
| dp_Res2 | Error code |

This packet implements the `ExAll()` function of the *dos.library* 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[4]. If the file system does not implement this packet, i.e. returns `ERROR_ACTION_NOT_KNOWN`, then the *dos.library* emulates it using `ACTION_EXAMINE_OBJECT` and `ACTION_EXAMINE_NEXT`.

`dp_Arg1` is a lock on the directory to be examined.

`dp_Arg2` is a pointer to the buffer to be filled with information on the objects found in the directory, it is *not* a BPTR. This buffer is filled with a singly linked list of `ExAllData` structures, see section 6.1.4 for the definition of this structure. Only the elements requested by `dp_Arg4` are filled, remaining entries are undefined.

`dp_Arg3` is the size of the buffer provided in `dp_Arg2` in bytes.

`dp_Arg4` defines which elements of the `ExAllData` are filled. The encoding of this argument is given by table 23 in section 6.1.4.

`dp_Arg5` is a pointer to a `ExAllControl` structure, also defined in section 6.1.4. A detailed description of this structure is also provided in section 6.1.4. Note that this is really a regular pointer, not a BPTR.

---

[4]The information on `dp_Arg5` in [1] is incorrect, it is really a pointer and not a BPTR

The file system shall provide in the `eac_Entries` element of this structure the number of `ExAllData` structures it could fit into the target buffer provided by `dp_Arg2`.

It 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. It is zero-initialized before the first packet, and thus the file system may assume that a value of 0 for the key identifies the start of the scan.

If non-`NULL`, `eac_MatchString` is a parsed pattern as generated by the `ParsePatternNoCase()` function, see section 8.2.2. 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 elements are filled into the target buffer; it supplies a `Hook` structure whose `h_Entry` function is called for each candidate directory entry. If the hook returns non-zero, the `ExAllData` copied to the buffer shall be considered accepted and shall remain in the buffer; if it returns zero, this candidate entry is rejected and will be removed from the final output. Calling conventions of this hook are also described in section 6.1.4.

The same precautions as for `ACTION_EXAMINE_NEXT` hold for this packet, too. In particular, the file system shall be prepared for the directory to get modified between scans, by either adding, removing, renaming or moving objects out or into the directory. Using a disk block or a pointer to a structure representing a specific object as `eac_LastKey` is therefore discouraged. Similarly, `dp_Arg1` will be a lock to the same directory, though may not necessarily be the identical lock as used in the previous iteration over the same directory, only the directory that is locked may assumed to be the same. In other words, `eac_DiskKey` shall hold sufficient state information to reconstruct the point from which the scan is to be continued. What is, however, is ensured is that the pointer to the `ExAllControl` structure stored in `dp_Arg5` did not change and the file system may depend on the pointer having the same value as on a the previous iteration over a 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.

This 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 aborted by an `ACTION_EXAMINE_ALL_END` packet.

In case the end of the directory has reached and all directory entries could be supplied, `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.

### 12.3.4 Aborting a Directory Scan

The packet `ACTION_EXAMINE_ALL_END` aborts a directory scan started with `ACTION_EXAMINE_ALL` that returned with `dp_Res1` non-zero and thus indicated that the end of the directory has not yet been reached.

**Table 106:** `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* 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.

`dp_Arg1` is the lock on the directory on which a partial directory 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, will not be touched and no data will be deposited there; as this buffer is not necessarily the same as the one for which the scan has been started, this information is likely not very useful, but it is provided here anyhow.

`dp_Arg4` is a type information which would define the information that would go into the buffer provided by `dp_Arg2`; however, as this packet does not request to add any data to the buffer, it is probably not very helpful and file systems should probably ignore it as no useful decisions can be derived from what the requested information actually was.

`dp_Arg5` is a pointer to a `ExAllControl` structure, and the pointer provided here is identical to the pointer provided to `ACTION_EXAMINE_ALL`. Thus, file systems may use this pointer value and specifically the `eac_DiskKey` element therein to release any resources related to this scan.

Before replying this packet, the file system shall set `dp_Res1` to a boolean success code whether it could abort the scan. If `dp_Res1` is set to `DOSTRUE` indicating that the scan was aborted successfully, then `dp_Res2` shall also be set 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 overruning the buffer. However, due to a defect in the *dos.library* up to the latest release, the (non-matching) pattern is a literal pattern and not a pre-parsed pattern and thus, actually, could match an entry in the directory. Thus, it is advised 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.

### 12.3.5   Examining from a File Handle

The `ACTION_EXAMINE_FH` packet provides meta data on an object identified by a file handle rather than a lock; that is, it uses the (implied) lock of the opened file to retrieve information.

**Table 107:** `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* and retrieves information on an object from a file handle rather than a lock.

`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 to be examined[5].

---

[5]Note, again, that the information in [1] on `dp_Arg1` is incorrect.

dp_Arg2 is a BPTR to a `FileInfoBlock` structure as documented in section 6.1, and which shall be filled by the file system with the information on the object identified by dp_Arg1. As for the packet type `ACTION_EXAMINE_OBJECT` specified in section 12.3.1, the `FileInfoBlock` passed out to the caller of the *dos.library* and what the packet shall deliver differ. In specific, fib_FileName and fib_Comment shall be BSTRs with the length of the string in the zeroth element, and not NUL-terminated C strings. The conversion to C strings happens within 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.

## 12.4 Packets for Working with Links

The packets in this section interact with links; they create them, or in case of soft-links, resolve them and update a path given the link. Section 6.4 provides background information on links and on the interface within the *dos.library* for links.

### 12.4.1 Creating Links

The `ACTION_MAKE_LINK` packet creates a link in a file system.

**Table 108:** `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 link to create |
| dp_Arg3 | BPTR to `FileLock` or APTR to C-string |
| dp_Arg4 | Type of the link, from table 25 |
| dp_Res1 | Boolean result code |
| dp_Res2 | Error code |

This packet implements the `MakeLink()` function of the *dos.library* and as such creates soft- or hardlinks within the file system.

dp_Arg1 is a BPTR to a `FileLock` structure that identifies the directory within which the object is to be created.

dp_Arg2 is a BPTR to a BSTR that provides the 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. If this `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 count 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.

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 this link. The FFS follows (mostly for backwards compatibility) this latter approach.

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 in the directory identified by `dp_Arg1`, 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 and all objects within the linked object if this object is a directory.

If the target of such a link is deleted or an object within the linked target, the copy within the file system containing the link remains accessible as a regular file or directory. If no copy has been created so far, 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. The external link pulls only those preferences settings into the RAM disk that are actually required, at the time they are required.

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 interpreted as a path name relative to the location of the link source at the time the link is resolved[6].

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 24.

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 returned, 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 6.4.2. This function then again contacts the file system containing the link with a `ACTION_READ_LINK` which will construct from the link an updated path to the object.

From this follows that a file system is not aware whether the target of a soft link actually exists as it can only provide the path to the object, but not necessarily the object itself; in particular, soft links may 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 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.

### 12.4.2  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 109:** `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, see section 6.4.2, of the *dos.library* and is used there to resolve soft links and obtain the link target. Implementing this packet correctly requires, however, some

---

[6]This is quite unusual as AmigaDOS otherwise depends on BSTR representation for packets.

care as the soft link may be in the middle of the path provided by `dp_Arg2` and not just its last component, and the soft link itself may be an absolute or relative path. This packet needs to 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`.

`dp_Arg1` is a lock to a directory in the file system to which this packet has been send and relative to which the path in `dp_Arg2` shall be interpreted. This *need not* to be the directory containing the link.

`dp_Arg2` is a pointer to a path name containing a soft link on this file system. This 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 section 4.3, until it finds a soft link. That is, colons (":") indicate that the scan should continue at the root directory of the volume, 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 may not only consist of an isolated file or directory name, but may 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 of the path stored in the soft link is then concatenated to this device or volume specification. 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 is then 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 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 15 links.

`dp_Arg3` is the pointer (not a BPTR) to a target buffer into which the newly constructed path, with the soft link resolved, shall be provided 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[7]. In case of error, `dp_Res2` shall also be set to an error code. In particular, if the target buffer size `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 consistutes 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

---

[7]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.

is the right choice — it helps an algorithm for soft link resolution 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 6.4.2.

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.

The following example is a skeleton code implementing the above algorithm:

```
/*
** 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;
  /* Pointer to DosList of this handler, needed
  ** for the name of the volume.
  */
  extern struct DosList *volumenode;

  /*
  ** 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.
  */
  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);

/*
** 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.
*/
```

```
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->dl_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);
}
```

## 12.5  Packets for Adjusting Metadata

Packets in this section change metadata associated with file system objects, such as name, comment, creation date or protection flags.

### 12.5.1  Renaming or Moving Objects

The `ACTION_RENAME_OBJECT` changes the name of an object such as a file, directory or a link, and relocates it within the directory tree of a volume.

**Table 110:** `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* and relocates and renames objects within the directory structure of a volume. This packet cannot relocate 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.

`dp_Arg1` represents the directory to which the source `dp_Arg2` is relative to.

`dp_Arg2` is the path of the original object location. 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 needs to 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.

`dp_Arg4` is the target path of the object, relative to `dp_Arg3`.

Several cases for renaming exist the file system shall consider: If the last component does not yet exist within the file system, the object identified by `dp_Arg1` and `dp_Arg2` shall be moved into the directory represented by the second to last component of of `dp_Arg4` starting from `dp_Arg3`, or directly into `dp_Arg3` if `dp_Arg4` only consists of a single component, and shall be renamed to the last component of the path in `dp_Arg4`.

If the object identified by `dp_Arg3` and the path in `dp_Arg4` does already exist and represents a directory, the object shall be moved *into* this directory, retaining its original name.

If the last component does already exist and is a file, and this file is different from the source object, an error shall be generated. Renaming a file or directory to itself, or to a file name that only differs in case shall, however, be accepted, and shall adjust its case (i.e. changing letters in its name to lower or upper case is permissible).

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 object into itself, i.e. into one of the subdirectories of its own. As this would render the directory unreachable, an error 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.

### 12.5.2  Deleting an Object

The `ACTION_DELETE_OBJECT` removes an object from a directory, releasing the storage it occupies.

**Table 111:** `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*, 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 with the current directory of the calling process.

`dp_Arg2` is the path to the object to be deleted. The path can contain multiple components, in which case the file system need to 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.

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.

### 12.5.3  Changing the Protection Bits

The `ACTION_SET_PROTECT` changes the protection bits of a file system object.

**Table 112:** `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*; 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 6.1, table 22. 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; 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 which the path in `dp_Arg3` is relative to[8].

`dp_Arg3` is a path relative to `dp_Arg2` whose last component is the object whose protection bits are to be changed.

`dp_Arg4` are the new protection bits as specified in table 22. Note that the first four bits are shown inverted by most system tools such as the `List` command.

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.

### 12.5.4 Setting the Comment to an Object

The `ACTION_SET_COMMENT` packet changes, adds or removes a comment on a file system object such as a directory, file or link.

**Table 113:** `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* 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.

`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 will be removed.

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.

### 12.5.5 Setting the Creation Date of an Object

The `ACTION_SET_DATE` packet sets the creation date of an object on a file system.

**Table 114:** `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 |

---

[8]This not a typo, `dp_Arg1` is really unused

This packet implements the `SetFileDate()` function of the *dos.library* 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[9].

`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 3 and defines the target date to install for the object.

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.

### 12.5.6  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 115:** `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* and is supposed to set a group and user ID of an object in a file system; the actual encoding of this information is specific to the file system and not defined by the *dos.library* or the packet. `dp_Arg4` is a literal copy of the second argument of `SetOwner()`. Note further that most if not all AmigaDOS file systems do not implement this packet, and as AmigaDOS is not a multi-user system, the value of this packet is questionable.

`dp_Arg2` represents a directory relative to which the path in `dp_Arg3` is interpreted.

`dp_Arg3` is the path of the object whose owner information is to be changed. This path shall be interpreted relative to `dp_Arg2`.

`dp_Arg4` is a file system specific owner information; the *dos.library* does not define its meaning. This argument is a verbatim copy of the second argument of `SetOwner()`.

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.

## 12.6  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. Notification requests are explained in section 6.5.

---

[9]The information in [1] is incorrect, and `dp_Arg1` is, indeed, unused.

### 12.6.1 Registering a Notification Request

The `ACTION_ADD_NOTIFY` packet registers a notification request at a file system.

<p align="center"><strong>Table 116:</strong> <code>ACTION_ADD_NOTIFY</code></p>

| 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*, registering a request at a file system to get informed on changes within the file system.

`dp_Arg1` is a C pointer[10] to a `NotifyRequest` structure that defines the object to be monitored, and the task, signal or message port to be triggered on such changes. This structure is specified in section 6.5.1.

The `nr_FullName` element of the `NotifyRequest` structure is already filled by an absolute path to the object to be monitored which may or may not yet exist. This path is filled in 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 may depend on this path.

The file system should store this request in its internal structures, and then monitor the object identified by `nr_FullName` for changes. The following table lists the packets which shall or may trigger a notification request upon a monitored object:

<p align="center"><strong>Table 117: Packets triggering a Notification</strong></p>

| 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_DELETE_OBJECT` | Immediately |
| `ACTION_SET_DATE` | Immediately |
| `ACTION_RENAME` | Immediately |
| `ACTION_SET_COMMENT` | Optionally |
| `ACTION_SET_PROTECT` | Optionally |
| `ACTION_SET_OWNER` | Optionally |

If the `NRF_NOTIFY_INITIAL` flag is set in `nr_Flags`, the file system shall send a notification immediately after receiving 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 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.

A notification event is either signalled through `Signal()` or by sending a message to a `MsgPort`:

If `NRF_SEND_MESSAGE` is set in `nr_Flags`, then a `NotifyMessage` is send to the port indicated in `nr_Msg.nr_Port`. This structure is also specified in section 6.5.1. 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 should trigger a signal bit on the target task if the monitored object changes by

---

[10]In [1] this argument is documented to be a BPTR, though this information incorrect.

```
    Signal(nr->nr_Signal.nr_Task,1UL<<nr->nr_Signal.nr_SignalNum);
```

For this method, `NRF_WAIT_REPLY` is, of course, non-functional.

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.

### 12.6.2   Canceling a Notification Request

The `ACTION_REMOVE_NOTIFY` packet cancels a notification request and aborts monitoring the object in the `NotifyRequest` structure.

**Table 118:** `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* and terminates monitoring the object in `nr_FullName` element of the `NotifyRequest` structure.

`nr_Arg1` is a C-pointer to the structure that indicates which notification request shall be canceled. The `NotifyRequest` structure is specified in section 6.5.1.

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 may come back because the client is still working on them. In particular, the memory associated to such messages shall not be released upon receiving the `ACTION_REMOVE_NOTIFY` packet, but 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.

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.

## 12.7   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.

### 12.7.1   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 119:** `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. If an error is generated by a function taking a `FileHandle` as argument, this function 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 handle is provided, a BPTR to the `DosList` of the volume the handle is located on shall be provided in `dp_Res1`. In such case, the packet is safe as the volume entry cannot go away as long as the lock implicit to the handle uses it[11].

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 provided 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, and should copy the name before the list is unlocked again, i.e.

```
char name[256];
struct DosList *dl;

  LockDosList(LDF_VOLUMES|LDF_READ);
  dl = (struct DosList *)BADDR(DoPkt0(port,ACTION_CURRENT_VOLUME));
  if (dl) {
    /* dol_Name is always a 0-terminated BSTR */
    strncpy(name,dl->dol_Name + 1,sizeof(name));
    name[sizeof(name)-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 this handler. 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.

### 12.7.2 Retrieving Information on a Volume from a Lock

The `ACTION_INFO` packet retrieves information on a volume, given a lock on an object on the volume, and provides it in an `InfoData` structure.

**Table 120:** `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* and provides information on the volume on which the locked object is located.

`dp_Arg1` is a BPTR to a lock, used to identify a volume from which information is to be requested. If the volume is not inserted, not validated or the lock is invalid, this packet will fail.

---

[11]In [1], `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 go away during its lifetime.

`dp_Arg2` is a BPTR to an `InfoData` structure as defined in section 5.2.4 into which information is provided 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 are found in section 5.2.4.

Upon replying this packet, `dp_Res1` shall be set to a boolean success indicator that includes the result of the validity test of the lock. If the lock is valid and 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.

### 12.7.3  Retrieving Information on the Currently Inserted Volume

The `ACTION_DISK_INFO` packet retrieves information on the volume currently mounted by the file system (if any) and provides this information in an `InfoData` structure.

**Table 121:** `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. It is almost identical to `ACTION_INFO`, except that it does not take a lock as argument. Therefore, it always refers to the currently inserted volume, and not to the volume the locked object is located on. Otherwise, the information provided by this packet and `ACTION_INFO` is identical.

This packet is also used to retrieve console specific information from the CON-Handler, and is documented as such again in section 11.3. 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 5.2.4 into which information is provided 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 5.2.4.

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.

### 12.7.4  Relabeling a Volume

The `ACTION_RENAME_DISK` packet changes the volume name of a medium.

**Table 122:** `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* and changes the volume name of a 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. Unfortunately, there is no guarantee that the volume that is inserted in the drive or partition managed by the file system when this packet is issued by the client is identical to the volume in the drive when the file system receives it.

`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.

### 12.7.5 Initializing a New File System

The `ACTION_FORMAT` packet initializes a new file system on a disk, writing all administration information on it represting a blank volume. It therefore deletes all information stored previously on the medium, drive or partition.

**Table 123:** `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*. It performs a soft-initialization of the drive, medium or partition currently inserted into the drive. Note, however, that it does not perform a low-level formatting, such as creating sectors on a floppy disk or issuing a low-level SCSI command. If this is intended, it must be performed by a client application, e.g. the `Format` command of the Workbench disk. This packet only writes administrative information on a volume representing it in empty state. This packet is probably the only one that should be used while a file system is inhibited and would usually refuse to alter any information on disk, see also section 12.9.2.

`dp_Arg1` is a BPTR to a BSTR of the volume name the medium, partition or drive shall be given[12]. Typically, this packet will be issued while the drive is inhibited, and thus the `DosList` structure representing the volume will be created or updated at the time the file system is unhibited. Otherwise, if the handler also allows an `ACTION_FORMAT` while not being inhibited, it should update the `DosList` immediately, in particular its `dol_Name`, to the name given by this argument.

`dp_Arg2` is file system specific information that may be used to define its flavour. For example, for the FFS `dp_Arg2` carries the `DosType`, which shall be one of the values documented in table 29 in section 7.1.3. If such a change of the `DosType` also happens, and the handler allows formatting while not being inhibited, the de_DosType in the environment vector linked through `dol_Dostype` shall be updated as well. Otherwise, this update will happen when the file system is uninhibited. 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.

### 12.7.6 Make a Copied Disk Unique

The `ACTION_SERIALIZE_DISK` packet serializes a disk, that is, ensures that the disk is unique and distinguishable from other media available to the system.

---

[12]The information in [1] that the new volume name is in `dp_Arg2` and the `DosType` is in `dp_Arg3` is incorrect.

**Table 124:** `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()`, needs to 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 should be issued while the file system is inhibited, see sections 7.7.4 and 12.9.2.

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 performing its job to ensure that the copy of a disk 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.

### 12.7.7 Write Protecting a Volume

The `ACTION_WRITE_PROTECT` packet enables or disables a software write-protection on the currently inserted medium, thus disallowing any write access through the file system on it.

**Table 125:** `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 media. 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, device 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 the stored password hash is not equal to the original hash, attempting to release the protection shall fail with `ERROR_INVALID_COMPONENT_NAME`.

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.

## 12.8 Packets for Interactive Handlers

The packet types documented in this section are specific for 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.

### 12.8.1 Waiting for Input Becoming Available

The packet `ACTION_WAIT_CHAR` waits for characters becoming available on an interactive handler.

**Table 126:** `ACTION_WAIT_CHAR`

| DosPacket Element | Value |
|---|---|
| dp_Type | ACTION_WAIT_CHAR (20) |
| dp_Arg1 | Timeout in ticks |
| dp_Res1 | Boolean result code |
| dp_Res2 | Error code |

This packet implements the `WaitForChar()` function of the *dos.library*. 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 specified which `ACTION_READ` packet will receive the available data.

`dp_Arg1` provides the timeout in ticks, where a tick is 20ms long, see also section 3.

This packet instructs the handler to wait at most `dp_Arg1` ticks. 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, `dp_Res1` shall be set to `DOSTRUE`. If the handler implements line buffering such as the CON-Handler, then `dp_Res2` shall be set to the number of lines available in the output buffer. This feature is, for example, used by the ARexx interpreter. If it does not implement line buffering, `dp_Res2` should be set to 0.

In case of an error, `dp_Res1` shall be set to `DOSFALSE` and `dp_Res2` to an error code.

### 12.8.2 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 127:** `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*. `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:` device, all of which are incarnations of the same handler configured differently; the CON-Handler described in section 11.3.

In particular, `CON:` and `RAW:` interact with the user through a window of the graphical user interface; they are just two modes of the in total three modes of the 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 exists.

The same type of mode switch can also be performed for the serial terminal represented by the `AUX:` window, which also exists in three modes in total, though usually only one of them is exposed to the user as `AUX:` device. This packet makes two additional configurations 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 13 in section 4.9.3.

Only values between 0 and 2 shall be provided in `dp_Arg1`; all other values are reserved for future use and may or may not correspond to modes that are currently not documented. The console does not generate an error if an invalid mode is requested, but may interpret this mode change request in an unexpected way.

Before replying this packet, the console shall set `dp_Res1` to DOSTRUE and `dp_Res2` to 0 in case the mode switch could be performed. Otherwise, if the mode switch is not possible, the console shall set `dp_Res1` to DOSFALSE and `dp_Res2` to an error code.

### 12.8.3 Retrieving IORequest and the Window Pointer from the Console

The `ACTION_DISK_INFO` packet retrieves from a console pointers to its underlying resources.

**Table 128:** `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 12.7.3 and takes the same parameters. However, when applied to consoles, it returns additional information and is thus discussed here again.

The `id_DiskType` element of the `InfoData` structure (see section 5.2.4) pointed to by `dp_Arg1` provides information on the mode the console is in:

**Table 129:** `id_DiskType`

| Value of id_DiskType | Console Mode |
|---|---|
| 'CON\0' | Cooked or medium mode |
| 'RAW\0' | Raw mode |

Section 11.3 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 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 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 12.8.4.

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 over which the console communicates with the user, and not 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. 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 independent 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(LONG *x,LONG *y)
{
  ParsePosition(stdOut,'R',x,y);
}

/*
** Maximum time to wait for the console
** to respond in ticks. May require
** adjustment on 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 request to the stream */
  if (answer == 'R') {
    Write(file,"\033[6n",4);
  } else {
    Write(file,"\033[0 q",5);
  }

  /* Now parse an incoming string */
  for(;;) {
    if (WaitForChar(file,MAX_DELAY) == FALSE) {
      success = FALSE;
      break;
    }

    if (Read(file,&in,1) != 1) {
      success = FALSE;
      break;
    }

    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.

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.

### 12.8.4  Releasing Console Resources

The `ACTION_UNDISK_INFO` packet releases any resources obtained by `ACTION_DISK_INFO`.

**Table 130:** `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 needs to be issued through the `DoPkt()` function.

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 be 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 is closed, either by iconification or because it is an `AUTO` window that was 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.

### 12.8.5 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 131:** `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 needs to 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`.

Lines from this buffer are provided to clients on `ACTION_READ` packets 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 user input.

Lines injected into the console output buffer are not echoed on the screen. Arexx uses this packet to employ 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.

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.

### 12.8.6 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 132:** `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 needs to 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`.

Lines from this buffer are provided to clients on `ACTION_READ` packets 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 user input.

Lines injected into the console output buffer are not echoed on the screen. ARexx uses this packet to employ the console as "external stack". This packet places a line at the end of this stack, i.e. queues them up.

`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.

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.

### 12.8.7 Force Characters into the Input Buffer

The `ACTION_FORCE` packet injects characters into the keyboard input buffer of the console, as if the user typed them.

**Table 133:** `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 needs to be issued through the packet interface, e.g. `DoPkt()`. This packet injects characters into the input 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. E.g, this packet allows also to move the cursor left or right, or erase the current line by emulating the corresponding keystrokes.

Note that the keyboard input buffer is different from the line output buffer; lines entering the console through `ACTION_FORCE` qualify as keyboard input[13]. The main user of this packet is the Shell which 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.

`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.

---

[13]Even though the V40 CON-Handler already supported this packet, it is there implemented incorrectly and does not impact the keyboard buffer. This was fixed in V47.

Upon replying this packet, `dp_Res1` shall be set to the number of characters that could be injected into the keyboard input 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.

## 12.8.8 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 134:** `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 |

This packet is not exposed by the *dos.library*, it rather needs to be issued through the packet interface, e.g. `DoPkt()`. This packet removes any lines injected by `ACTION_STACK` and `ACTION_QUEUE` from the output buffer of the console and thus reverts their results. User keyboard inputs remain unaffected. Thus, this packet resets the line stack in the console. The intended purpose this packet is ARexx which, however, at the time of writing does not use it.

`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.

## 12.8.9 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 135:** `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. This packet brings the window associated 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.

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.

### 12.8.10 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 136:** `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 a port to which break signals as generated by the `Ctrl-C` to `Cltr-F` key combinations are send. Actually, this *break port* is not used as a target of a message, but only its `mp_Task` element to which the signal is send.

Despite this port, the console also sends break signals to the last process that issued an `ACTION_WRITE` request, provided this process is not a Shell process running in background. To avoid race conditions of applications forgetting to reset this port, the console resets this port whenever an `ACTION_CLOSE` is received from the *break port*, i.e. whenever the process to which this port belongs closes its connection to the console. Furthermore, the console also tests the validity of the *break port* by testing whether its `mp_Task` field is known to the exec scheduler.

This packet is, for example, used when a Shell is started in a console window, or a `NewShell` command creates another in an already open console window. 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 the Shell then running in the background. This packet is *not* send for executables placed by the `Run` command in the background. Even though `Run` also goes through `System()`, it uses different parameters.

`dp_Arg1` is a copy of the `fh_Arg1` element of a `FileHandle` structure that is opened to the console.

`dp_Arg2` is a pointer to a `MsgPort` to whose task break signals will be send. In addition to this port, the console also sends break signals to the last issuer of an `ACTION_WRITE` packet, provided it is not a background Shell process.

Before replying this packet, `dp_Res1` shall be set to `DOSTRUE` on success, and `dp_Res2` the port that was previously installed as *break port*, or `NULL` if no such port was configured. On error, `dp_Res1` shall be set to `DOSFALSE` and `dp_Res2` to an error code.

## 12.9 Packets Controlling the Handler in Total

The packets in this section impacts file systems and handlers at global. They do not apply to a particular volume.

### 12.9.1 Adjusting the File System Cache

The `ACTION_MORE_CACHE` packet increases or decreases the number of file system buffers.

**Table 137:** ACTION_MORE_CACHE

| DosPacket Element | Value |
|---|---|
| dp_Type | ACTION_MORE_CACHE (18) |
| dp_Arg1 | Buffer increment |
| dp_Res1 | Boolean result code |
| dp_Res2 | Buffer count or error code |

This packet implements the AddBuffers() function of the *dos.library*. In increases or decreases the number of buffers the file system may use to cache data. How exactly a file 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 administrative data such as blocks describing the directory structure or information on 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 7.1.3.

dp_Arg1 is the increment (or decrement, if negative) of the number of buffers to 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.

If this packet succeeds, the file system shall set dp_Res1 to DOSTRUE and dp_Res2 to the number of buffers currently allocated. 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.

### 12.9.2 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 138:** 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*. An inhibited file system is blocked from accessing its underlying medium for read and write access, with the exception of ACTION_FORMAT and ACTION_SERIALIZE_DISK which remain operational. An inhibited file system changes its disk type as reported by ACTION_INFO or ACTION_DISK_INFO in the InfoData structure to 'BUSY', see table 20 in section 5.2.4.

Once the file system is unhibited again, it shall perform a validation of the medium, as if the medium was been re-inserted. This is necessary because an application bypassing the file system to access it directly could have changeed the file system structure, the volume name or the date, or may have even written a completely new file system structure to it. This check thus implies checking the flavour 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 inhbiting and uninhibiting a file system is equivalent to simulating a medium change; in fact, the DiskChange command enforces a medium change by first inhibiting and then unhibiting the file system.

`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 unhibited and a file structure check shall be performed. In all other cases, the file system shall be inhibited and, with the exception of `ACTION_FORMAT`, should refrain from accessing the medium, partition or device.

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.

### 12.9.3   Check if a Handler is a File System

The `ACTION_IS_FILESYSTEM` tests whether a handler is a file system.

**Table 139:** `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* and tests whether a particular handler provide sufficient services to operate as a file system. 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. A file system may be either a flat or a hierarchical file system, i.e. a file system is not required to support multiple directories on a storage medium, device or partition.

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 identified 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` on the file system root. If such a lock can be obtained, then this heuristic assumes that the handler is, in fact, a file system.

### 12.9.4   Write out all Pending Modifications

The `ACTION_FLUSH` packet instructs the file system to write all pending or cached modifications out to disk[14].

**Table 140:** `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 any write operations an arbitrary amount of time. This packet instructs the file system to write out all modifications immediately.

---

[14]Users of unixoid systems may consider this as the AmigaDOS equivalent of `sync`.

This packet shall not be replied before all changes have been written and, if applicable, the drive motor has been turned off. However, FFS versions of V40 and before replied `ACTION_FLUSH` immediately and thus only established a write barrier. This was fixed in V43.

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[15]. In case of error, file systems should set `dp_Res1` to `DOSFALSE` and `dp_Res2` to an error code.

### 12.9.5   Shutdown a Handler

The packet `ACTION_DIE` requests a file system or handler to unmount its volumes and terminate.

**Table 141:** `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 needs to be send through the packet interface, i.e. through `DoPkt()`. This packet requests the file system to release all resources, set the `dol_Task` elements of the `DosList` structures in the device list to `NULL`, and then terminate the process.

A file system may not exit as long as client programs cache copies of its `MsgPort`; this port is typically the `pr_MsgPort` of the handler process which is released by AmigaDOS as soon as the process exits, and it will thus become unusable. At least all file handles and locks need to indicate that the file system is no longer usable. Either, the file system may refuse to exit if some file handles using it are still open, or some locks are not yet unlocked. Alternatively, the file system may zero out the `fh_Type` element of all files open to it, thus essentially turning them into `NIL:` file handles, also zero out the `fl_Task` elements of all active locks, and link them in `dol_LockList` of the `DosList` of their corresponding volume as if the volume had been ejected, see also section 7.

As next step, references to the file system port(s) shall be removed from the device list. To gain access to the device list, 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 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 from AmigaDOS to launch a new file system from the `dol_SegList` segment as soon as a client requests again access it through its device name.

Unfortunately, even these precautions are not fully sufficient for safely unmounting a device because a client program can request a handler message port via `GetDeviceProc()` or `DeviceProc()` any time for direct communication and buffer the returned port outside of the classical AmigaDOS objects. Thus, a handler has no direct control of where 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`. In case termination is seemingly possible, the file system shall check for any pointers to its `MsgPort`(s) in the device list and replace them by `NULL`, thus requesting from the *dos.library* to create a new instance of the file system if needed. Then, 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 may still contain packets that were submitted after `ACTION_DIE` has been

---

[15]The official reference [1] 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.

received. In order to avoid a deadlock or a crash, the file system still needs to reply these packets, for example by the default result codes from table 63, see also section 11.2.3.

Some handlers do not initialize the `dol_Task` element of its `DosList` entrie(s) with an address of their port(s); handlers that are not file systems such as the CON-Handler are typical examples. Such handlers shall terminate themselves without requiring an `ACTION_DIE` when the last resource opened on 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 notification requests (if supported) had been canceled.

### 12.9.6  Do Nothing

The `ACTION_NIL` packet performs no activity.

**Table 142:** `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 11.2.1.

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.

## 12.10  Handler Internal Packets

The packets in this section shall never be sent to a handler or filesystem 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 instead 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`.

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 to process 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.

### 12.10.1  Receive a Returning Read

The `ACTION_READ_RETURN` packet is part of a returning `CMD_READ` or similar read `IORequest` sent to an exec device and indicates that reading data completed.

**Table 143:** `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.

### 12.10.2   Receive a Returning Write

The `ACTION_WRITE_RETURN` packet is part of a returning `CMD_WRITE` or similar write `IORequest` sent to an exec device and indicates that writing data completed.

**Table 144:** `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 writing data completed. As such, this packet shall never be replied.

### 12.10.3   Receive a Returning Timer Request

The `ACTION_TIMER` packet is part of a returning `TR_ADDREQUEST`, a `timer.device` request to wait for a time period.

**Table 145:** `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`.

## 12.11   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 146: 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_FINDREADONLY(1009) | open a file for exclusive reading |
| ACTION_FINDONEWRITER(1010) | open a file for exclusive writing |
| ACTION_DIRECT_READ(1900) | used by the CDTV |
| ACTION_DOUBLE(2000) | create two file handles for a pipe |
| ACTION_PEEK(2005) | retrieve console input line |

| | |
|---|---|
| `ACTION_REPLACE(2006)` | replace console input line |
| `ACTION_GET_HISTORY(2007)` | retrieve 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 13

# The AmigaDOS Shell

The Shell is the command line interpreter of AmigaDOS and is a simple language of itself. User applications can use services of the shell by requesting it to interpet 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 ROM of the kickstart, 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.

## 13.1 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 upfront and its arguments, where the arguments amongst each other and the command are are separated by spaces, tabulators or by equals-signs ("= ") which separates keywords from its values. The command refers to a file within the *path*, a list of directories that can be adjusted by the user. It always contains the current directory and the `C:` assign, though additional directories can be added or removed with the `path` command any time.

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, backtick expansion and additional redirection operators.

### 13.1.1 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 "*". Optionally, they can 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.

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 a 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 indicated.

*<<ind* uses the current console or the shell script itself from which the command is run as input, line by line, until a line starting with `ind` is found. The indicator can be choosen arbitrarily. A typical choice is `EOF`. This avoids creating temporary files.

*<>path* redirects both the standard input and the standard output of the command to a path. The file must exist already, otherwise an error is indicated.

*\*>path* redirects the error output of the command to the given path. If this path identifies an interactive stream (see section 4.5.1), such as a console, then the console process of the command is set to it as well. This will also redirect output of commands that prints errors to the `*` file that is contacted for this special file name.

*\*>>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.

*\*<>* creates a standard error stream and redirects it to the standard output. 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 provide error output to the standard output as if an implicit `*<>` would be present.

Where the redirection is placed within a command line does typically not matter, e.g. wether 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. The same applies if the shell variable `oldredirect` is set to `on`. In the above cases, 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 back will redirect its streams rather than the streams of the `Run` command, and for `Alias`, the redirection will become part of the alias and will become active when the alias is expanded. The `oldredirect` variable will do likewise for all other commands. However, they will typically not interpret redirection requests in a useful way and will instead take the angle brackets as regular arguments.

### 13.1.2 Compound Commands and Binary Operators

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 argument start to the right. As for arguments, these operators need to be separated by arguments and commands with blank spaces or tabulators:

| 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, which is mounted by the Kickstart during the system startup.

|| Two vertical bars concatenate the output of two commands together into one common stream which can then redirected to a common output[1].

&& Sequences two commands, first executes the one to the left, and if its return value is below the `FailAt` value, continues to execute the command to its right.

_____

[1]This syntax is different from the `bash` operator that looks similar.

The brackets " (" and ") " group commands and execute the grouped command in a separate process in a sub-shell. For this, both the opening and closing bracket shall be separated by commands and arguments by at least a single space or tabulator as otherwise the shell interprets them as part of the command or arguments. This provides logical grouping of commands into compound commands.

### 13.1.3 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 ampersand 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 of course redirected explicitly to other files. 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 extended CON-Handlers use this information for job control and hold the output of the background process until made explicitly active.

+ 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 4.8.2. This sub-shell will thus receive a shell script as input which consists of multiple lines, and will execute the commands one after another.

; The semicolon ends a command line and starts a commend. All characters beyond the semicolon are ignored.

### 13.1.4 Quoting and Escaping

To allow spaces, tabulators, equals-signs and the above operators as parts of command names and arguments, e.g. to handle file names containing such characters, the Shell allows quoting. Blank spaces, tabulators, semicola and equals-signs within double quotes ("") become part of the argument, and are not interpreted by the shell. A quote only starts a quoted argument or command and is thus a functional element of the Shell if it is preceeded by a blank space or a tabulator, quotes *within* an otherwise unquoted argument are literals and stand for itself. 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 recgonized 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 can also appear within the command line of the executed program, which is then responsible to perform the above substitutions correctly itself. In particular, the above 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.

Proper interpretation and escape sequence substitution is ensured if the command uses the `ReadArgs()` or `ReadItem()` functions of the *dos.library*. 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 handle them[2]. 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 13.1.5.

*` The literal backtick. The non-escaped backtick is a syntactical element for command output substitution, see section 13.1.6.

*[ The literal opening square bracket. Square brackets are part of the alias syntax, see section 13.1.7, 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.

### 13.1.5  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 lst 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 are allowed as components of the name as well.

Local variables are represented as entries in the `pr_LocalVars` list that is part of the process structure that represents the Shell, see section 9. 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

---

[2]This inconsistency can probably be only understood in historical contents as variable expansion and backtick substitution were later Amiga specific extensions to the original Tripos Shell.

contains spaces, tabulators or equals-signs and it is not included in double quotes, it will be seen 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) asterik as escape character to ensure the resulting argument is represented appropriately.

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 commands and their argument parser.

### 13.1.6   Backtick Expansion

After variable expansion, the Shell checks the command line for backticks, ("`"). The characters within two terminating backticks 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 backticks. As this output may include line feeds, these are furthermore substituted with blank spaces.

Note that the result of the executed commands may even contain functional syntax elements of the Shell, e.g. angle brackets as output of the command in backticks will become shell stream redirection operators which are then further interpreted by the shell. Needless to say, this can cause bad surprises.

The backticked 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 backticked 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 backticked 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 backticked command, same as in variable substitution.

### 13.1.7   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 its arguments. The leftmost element of each compound command can potentially be an alias; the remaining arguments right to it do not go through alias expansion.

Aliases are always local to the executing Shell, AmigaDOS keeps them in the same list as local Shell variables. If the leftmost element fits a defined alias, this element 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 same component of the compound command. The result of alias expansion is another command that undergoes alias expansion again, except that the same alias cannot be used twice to avoid endless recursion.

### 13.1.8   Command Execution

After alias expansion, the Shell takes the leftmost element of each component of a compound command as a command to be executed. The Shell attempts to locate this command first on the list of resident commands, see section 13.6, but only if the command is not quoted. If no match is found there, or the command is

quoted, then the Shell keeps looking in the current directory, and if it is not found there, in all elements of the *path*, as set by the `path` command. If no matching file or directory is found there, the Shell finally looks into the `C:` directory[3].

Once a matching file system object has been found, the Shell attempts to "execute" it and for that checks its type and contents. If this "command" turns out to be a directory, the Shell performs an implicit change to this directory by inserting a `CD` command upfront, and then triggers iteratively command execution again with this replacement.

If the `s` protection bit is set, see section 6.1, 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 command 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 injected into the Shell command line upfront the script file, and command interpretation starts again.

If none of the above conditions hold true and the `s` protection bit is set, the Shell assumes that this file is a Shell script prepends an `Execute` command upfront the script file name, thus interpretiing it through the Shell itself — more on the Execute command in section 13.1.9.

If the `e` protection bit is set indicating an executable file, the Shell attempts to load the file through `LoadSeg()` and, after performing input, output and error stream redirection, transfers program execution to this binary through `RunCommand()`, passing it the remaining command line arguments.

If the `e` protection bit is *not* set, but the shell variable `$VIEWER` is set, then the Shell attempts to check whether the system *datatypes.library* knows anything about the file. If so, the Shell prepends the contents of this variable to the command line and then continues execution. By setting `$VIEWER` to the system program `MultiView`, the Shell can therefore display any system-known datatype by just typing its name.

Finally, if none of the above conditions is true, or `LoadSeg()` failed, the Shell indicates an error that the file is not executable.

### 13.1.9   The Execute Command

Unlike its name suggests, the `Execute` command does not actually perform interpretation Shell scripts and does not "execute" them directly. Instead, it performs argument substitution within an existing shell script through a simple interpreter, and then leaves the execution of the resulting script to the Shell itself 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 itself[4].

Argument substitution and its 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 a 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 none of these pseudo-commands are present, `Execute` bypasses argument substitution and feeds the script unaltered into the Shell, without further processing.

The following pseudo-commands are supported by the `Execute`:

`.dot` takes a single argument and changes the character by which all (following) peudo-command start. This is by default a dot ("`.`"), and for the sake of simplicity, the following pseudo-commands are all shown with this default.

---

[3]Note that unlike under unixoid systems, the current directory is always an implicit first member of the path and cannot be removed from it.

[4]This is probably another historical accident from Tripos legacy

`.key` or `.k` defines which arguments a shell script takes. The argument is a `ReadArgs()` type template, defining the type of the command line arguments the shell script takes, and also the names of the formal parameters which will be substituted with the arguments provided during invocation of the script. Clearly, only a single `.key` pseudo-command shall be present within a script.

`.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 can be either separated by blank spaces, tabulators, or an equals-sign "(=)". If no default value is provided, an empty string is used. One `.default` peudo-command can be provided per formal parameter. Another mechanism to provide default parameters is through the "$" character.

`.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 one 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` 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 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, this character is the dollar sign ("$") An optional "$" sign, or its replacement defined through this pseudo-command, separates the formal parameter from its default value. The formal name, the "$" sign, and its default are all enclosed in this order in the angle brackets, or the characters defined by `.bra` and `.ket`. Two "$" 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. 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 path as variable expansion, 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 suitable 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 13.5.1), then `Execute` also expands it as multiple parameter in scripts.

The generated script after substitution of formal parameters is placed into the `T:` assign, its name in `cli_CommandFile` and the command input of the invoking Shell, namely `cli_CurrentInput` (see section 13.3), is redirected to a file handle to this temporary script. Thus, the shell continues execution of commands, but rather takes its input from the temporary file rather than its current input. 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 execution a "clean" environment[6] that allows redirection of its input. Once `cli_CurrentInput` exhausted, the Shell terminates execution there, closes the file handle stored there, 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`. The `System()` function with a

---

[5]This is another historical accident, likely.

[6]This is a major difference compared to previous versions of AmigaShell which instead resolved this situation by appending 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.

non-`NULL` `command` argument will terminate after `cli_CurrentInput` runs out of data, otherwise the shell switches over to `cli_StandardInput`, see section 13.7.1.

## 13.2    Shell Process Control

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.

### 13.2.1    Execute Shell Scripts

The `System()` function creates a new Shell, and potentially executes a Shell script. 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 3 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, possibly only a single command, and additional parameters encoded in a `TagItem` array as defined in `utility/tagitem.h`.

The `SystemTags()` function also receives this string, but provides the `TagItems` explicitly as one or multiple exra 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)
D0                              D1        D2

LONG SystemTagList(STRPTR, struct TagItem *)

error = System(command, tags)
D0                      D1        D2

LONG System(STRPTR, struct TagItem *)

error = SystemTags(command, Tag1, ...)

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, and multiple commands separated by newline characters may be provided that will be executed one after another. Argument substitutions as by the `Execute` command (see section 13.1.9) are *not* performed, but otherwise the entire Shell syntax, including variables shell, compound commands and pipes is available. If such mechanisms are necessary, `Execute` shall be called directly, receiving its input from a temorary file.

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.

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 section 9 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 and provide suitable tags.

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 to the console through the * file name.

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 shell exists, i.e. `pr_CLI` 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> " 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 13.6. Other shells can be provided by their name, and the list of resident segments will be scanned for a match then. Section 13.7 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.

`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.

`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 done, regardless of other tags. This tag is mutually exclusive to `SYS_Input`.

`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* than 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 this stream is explicitly set to ZERO, then AmigaDOS will attempt to re-open a console through the "*" file name. If an input stream is present and interactive, and `command` is non-NULL or `SYS_Asynch` is non-zero, then the handler of the input stream is used to open the replacement output stream. 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`.

`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 and `command` is NULL, 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. Thus, the configuration of providing a SYS_CmdStream and setting `SYS_Asynch` is equivalent to the `NewShell` command.

`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.

SYS_Asynch If the argument to this tag is non-zero, the shell is detached from the calling process and executes concurrently with the calling process; setting this tag also implies that the streams provided by SYS_Input and SYS_Output are always closed. 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 tag.

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. This 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 13.7.

SYS_CustomShell this tag provides the name of a custom shell to be used instead. The AmigaDOS list of resident segments is scanned for a fitting name, and the fitting segment is then used as shell.

The following tags defined for CreateNewProc(), see section 9.1.1, are also recognized:

NP_StackSize defines the stack size in bytes for the shell to be created and 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".

NP_Priority is the priority of the shell process to be created. The default priority is 0.

NP_ConsoleTask provides a pointer to the MsgPort of the console handler that is used for opening the * and CONSOLE: files, which will be copied to pr_ConsoleTask. This tag is only used if the input stream of the new shell is not interactive, and otherwise overridden by the process MsgPort port of the input file handle.

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 a linked list of directory locks that establish the path of the newly created shell. The structure of this path is specified in section 13.3. The path provided by this tag *is not* copied for the new shell, but directly used *and released* by it when it exits. Thus, callers should make a copy of the path upfront as it will be released by the created shell.

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 9.1.1.

The return code of the System() and related functions 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.

This function 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 commands and the Shell icon on the Workbench. The initial CLI interpreting S:Startup-Sequence is also created through the System() function.

The Run command creates a new shell process by the following:

```
System(NULL,SYS_Input,instream,SYS_Output,Output(),
       SYS_UserShell,TRUE,TAG_DONE);
```

where the input stream instream is a string stream (see section 4.8.2) containing the commands to be executed; they are taken from the input arguments of the command.

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 uses `System()` as follows:

```
System(NULL,SYS_InName,window_arg,
       SYS_CmdStream,Open(from_arg,MODE_OLDFILE),
       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 `NewShell` of the same names. 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.

The `Execute()` function described in section 13.2.2 is equivalent to

```
System(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 4.8.2) 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 only the result code, but internally run into the same code.

## 13.2.2 Execute Shell Scripts (Legacy)

The `Execute()` function creates a new AmigaDOS shell, which then executes commands from a string, and 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 superceeded by `System()`, see section 13.2.1.

The first argument is a string containing a shell script. 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. The output of the commands go to the stream provided as third argument if non-`ZERO`, or to a console cloned by opening "`*`" from the console task of caller if possible, or to a `NIL:` handle if this fails.

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, switches over to the stream provided by `input`. If this does not exist, it returns at that point. Otherwise, it continues reading from `input` until this stream reaches its EOF, or an `EndCLI` command. 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 if it is interactive, or otherwise disregards all output by providing a `NIL:` handle as output.

Unlike `System()`, this function returns `DOSTRUE` on success and `DOSFALSE` in case creating the shell failed. 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 13.1.9. 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 1.3 and below, `Execute()` depended on the `Run` command, but this dependency has been removed long since. As of the latest version of AmigaDOS, `Run` depends on the `System()` function.

### 13.2.3   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)
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 first segment of the executable to start, for example as returned by `LoadSeg()` for disk-based commands, or `FindSegment()` for resident commands.

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 13.3, and multiplies it by 4. It additionally searches the `seglist` for the stack cookie, see section 10.5.2, and from that potentially increases the size.

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 section 9, 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 4.8.1. Thus, buffered I/O operations as those listed in section 4.7 will retrieve the arguments. This step is necessary to make the arguments available to `ReadArgs()`, 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 `SetProgramName()` upfront, more on this function in section 13.3.4. 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.

### 13.2.4   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)
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 `Ctrl-C` through `Ctrl-F` console key combinations. The function returns a mask of all signals set in the process, and clears those signals that are set in the mask. A typical example is

```
if (CheckSignal(SIGBREAKF_CTRL_C)) {
  break; /* abort the command */
}
```

which checks whether the user pressed `Ctrl-C`, and also clears this bit. This function does not alter `IoErr()`.

### 13.2.5  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)
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 `ULONG`s, 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 newest end of the AmigaDOS Shell history and make it accessible to the user through the cursor keys. This method takes a single tag, namely SHELL_ADDH_LINE. The argument of this tag is a UBYTE * to a NUL-terminated string, namely the entry to add to the history. The Shell copies the provided string, and may strip initial or trailing spaces. It may also limit the size of the history buffer by releasing the oldest entry or entries. 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 the latter provides elementary line editing functions of the console, but does not have access to Shell internal states such as the path of the caller requird for TAB expansion[7].

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 or multiple paths relatively 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 quesion 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 on 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.

### 13.2.6 Find a Shell Process by Task Number

The FindCliProc() function finds a process by its task number.

```
proc = FindCliProc(num)
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 section 9.

This function returns a pointer to the process structure, see section 9 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 or an attempt is made to access its process structure. 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*

---

[7]Readers beware: TAB expansion and the history are *not* console features, but Shell features.

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.

### 13.2.7 Retrieve the Size of the Process Table

The `MaxCli()` function returns the size of the process table.

```
number = MaxCli()
D0

LONG MaxCli(void)
```

This function returns the number of entries the process table may 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 feed of the caller. Thus, this function should be avoided.

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 value is only valid as long as the task switching remains disabled.

## 13.3 The CLI Structure

The `CommandLineInterface` structure is the public interface of an AmigaDOS shell. Every command started from a shell has access to this structure through the `pr_CLI` BPTR in its process structure, see section 9, which is actually the process structure of the shell calling the program, see section 13.2.3.

The functions listed in this section provide accessor functions to this structure which should be preferred to access and alter properties of the calling shell.

### 13.3.1 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 instead.

```
success = GetCurrentDirName(buf, len)
D0                          D1   D2

BOOL GetCurrentDirName(STRPTR, LONG)
```

This function checks whether the caller is a shell command. If so, it copies what the shell believes to be its current directory into the supplied buffer. That is, the function then copies `cli_SetName` into `buf`, see section 13.3. 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, returns `DOSFALSE` and sets `IoErr()` to `ERROR_LINE_TOO_LONG`, but still returns `DOSTRUE`.

The shell itself does not update `cli_SetName`, namely the field this function depends upon, itself. Rather, the CD, `SwapCD`, `PushCD` and `PopCD` commands attempt to keep it consistent. 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 6.3.1, 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 then.

### 13.3.2  Set the Current Directory Name

The `SetCurrentDirName()` sets the buffer within which the shell keeps the string representing the current directory. It does not update the current directory itself.

```
success = SetCurrentDirName(name)
D0                               D1

BOOL SetCurrentDirName(STRPTR)
```

This function updates `cli_SetName` of the shell the caller is part of, if such a shell exists. In such a case, it copies the supplied string into `cli_SetName` and potentially truncates it to the size of the shell-internal buffer. Even if the string is truncated, the function returns `DOSTRUE`. It does not update `IoErr()` in either case. For some legacy reasons, the size of this buffer is rather small, and thus may not reflect the full directory name supplied.

If the caller is not a shell command, this function returns `DOSFALSE` without setting an error code.

This function does not attempt to synchronize `pr_CurrentDir` of the calling process to the supplied directory, 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* call `CurrentDir()` to make this change consistent to the Shell.

### 13.3.3  Obtaining the Current Program Name

The `GetProgramName()` function copies the name of the currently executed program into a buffer.

```
success = GetProgramName(buf, len)
D0                            D1   D2

BOOL GetProgramName(STRPTR, LONG)
```

This function fills `buf` with the what the Shell assumes to be the name of the currently executed command. This name is taken from `cli_CommandName`, see section 13.3, 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 and returns `DOSFALSE` and sets `IoErr()` to `ERROR_OBJECT_WRONG_TYPE`.

The Shell buffer that keeps the current command name is unfortunately due to legacy reasons quite limited in size, and at present shorter than the 106 character limit imposed by the `FileInfoBlock` structure, see section 6.1. Even though the Shell uses internally a longer buffer and thus is not limited in the file name size of commands it executes, the ability of the Shell to communicate such long command names to the caller is restricted, and thus the command name retrieved from this function may not reflect the correct file name.

### 13.3.4   Set the Current Program Name

The `SetProgramName()` sets the name the shell assumes the currently executing program has.

```
success = SetProgramName(name)
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`, section 13.3, and returns `DOSTRUE`. If the supplied string does not fit into the Shell internal buffer, it is truncated without this function indicating failure.

If this function is not called from a shell command, it returns `DOSFALSE`. This function does not change `IoErr()` in any case.

### 13.3.5   Obtaining the Shell Prompt

The `GetPrompt()` function copies the prompt format string with all formatting instructions into a caller supplied function.

```
success = GetPrompt(buf, len)
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 unexpanded prompt including format strings as it is provided by the `Prompt` command, and not the expanded prompt currently printed by the shell.

### 13.3.6   Setting the Shell Prompt

The `SetPrompt()` sets the Shell prompt format string.

```
success = SetPrompt(name)
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 13.3, such as `%S` for the current path, or Shell variables and backticks to construct a prompt dynamically. The shell-internal buffer size for the prompt is unfortunately quite limited. If longer prompts are required, they could be placed in a local shell variable which is expanded in the prompt.

### 13.3.7 Retrieving the CLI Structure

The `Cli()` function returns a pointer to the `CommandLineInterface` structure describing the shell within which the calling process is executing, or `NULL` in case the process is not run from a shell.

```
cli_ptr = Cli()
D0

struct CommandLineInterface *Cli(void)
```

This function returns a pointer to the `CommandLineInterface` structure that describes properties of the shell the calling process runs within. 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;
};
```

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 this value 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.

`cli_CommandDir` contains a linked list of directories that are scanned for commands. 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 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 linked 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 name of the currently executing command. It is placed here by the shell before calling in.

`cli_FailLevel` contains the threshold at which executed commands will cause an abortion of its 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 print the command prompt on interactive shells. The AmigaDOS Shell recognizes the following strings:

%S  Replaced by the path of the current directory, namely the contents of `cli_SetName`.

%N  Substituted with the CLI number, which is the closest analogon of a process ID AmigaDOS has to offer. This is taken from `pr_TaskNum` of the process running the shell, see also section 9.

%R  Represents the return code of the last executed command as contained in `cli_ReturnCode`.

%%  The percent ("%") sign itself.

The AmigaDOS Shell also expands variables as described in section 13.1.5 in the prompt, executes commands backticks, see section 13.1.6, and inejcts its output into the printed prompt. Any "%" sign included in expanded variables or backticks is *not* a formatting command but stands for itself.

`cli_StandardInput` is a BPTR to a file handle that represents the primary source of commands. 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 4.8.2) containing the command or commands to be 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 exhaused, the Shell terminates.

`cli_CurrentInput` is a BPTR to a file handle from which the shell is currently executing commands. This stream is either comming from the `SYS_CmdStream` or `SYS_CmdIn` tags of the `System()` function, or a string stream constructed from its first argument. Also, the `Execute` command, see section 13.1.9, places here the file handle of the original or processed shell script that is to be executed. Once this file is exhaused, the Shell will close it. This happens, for example, if a script reaches its end of file or is aborted by the `Exit` command. Depending on how the shell was created and configured, see section 13.7.1, the shell then sets this BPTR to `cli_StandardInput` and continues execution, 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` actually works see section 13.1.9.

`cli_Interactive` is a boolean flag that indicates whether the Shell is interactive, i.e. requesting data from the console. If this boolean is non-zero and `cli_Background` is `DOSFALSE`, a prompt is printed before attempting to read a command from `cli_CurrentInput`. If the shell is executing a script, this element is `DOSFALSE` and the Shell then checks for a `Ctrl-D` signal to potentially abort a running script.

`cli_Background` is a boolean that indicates whether the Shell runs in background. This flag is set for shells that are started asynchronously or are equipped with a non-interactive (non-console) output stream. If this flag is cleared, then the shell also prints a message when its input stream reaches its EOF, indicating that the shell terminates.

`cli_CurrentOutput` is currently not used by the Shell. It is currently 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 the command for a stack cookie, see section 10.5.2, 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 for its clients.

`cli_StandardOutput` is the file handle the shell provides as default output handle to its clients, and to which it prints output. This handle is copied from the `SYS_Output` or `SYS_OutName` tags of the `System()` function if they are provided.

`cli_Module` is a BPTR to the first segment of the command currently executed. The Shell also uses this BPTR to release loaded, non-resident commands. This BPTR is either filled with the segment provided by `LoadSeg()` in section 10.3.1, or an element of the list of resident segments, see section 13.6. A common technique for load and stay resident commands is to set this BPTR to `ZERO` to prevent the shell from releasing the loaded program and thus keep it in memory.

Even though the `CommandLineInterface` structure is typically constructed by AmigaDOS, e.g. through the `System()` function, it is sometimes necessary to create it manually. Allocation of this structure shall happen only through `AllocDosObject()` as this structure contains some Shell-internal elements for extended Shell features. These extended features are instead accessible through the `DoShellMethod()` function specified in section 13.2.5.

This function does not change `IoErr()`.

## 13.4   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.

### 13.4.1   Read 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 )
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 on file systems. It is not case-sensitive. For global variables, this hierarchy is mapped to subdirectories 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, but if no matching variable is found, it checks for a global variable of the name provided.

The `buffer` and `size` arguments specify a buffer and its length into which the contents of the variable, if it exists, is copied. It is advisable to store only printable characters in variables, in which case any `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, but 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. It is, however, `NUL` terminated, unless `GVF_DONT_NULL_TERM` is set.

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`.

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, 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()`.

### 13.4.2 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 )
D0                 D1      D2     D3    D4

BOOL SetVar(STRPTR, STRPTR, LONG, ULONG )
```

This function assigns the value in `buffer` to the variable 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. subdirectories for global variables, 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 internally. 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 this flag 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 this flag 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.

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`.

### 13.4.3   Finding a Shell Variable

The `FindVar()` locates a local shell variable or alias of the calling process.

```
var = FindVar( name, type )
D0                 D1    D2

struct LocalVar * FindVar(STRPTR, ULONG )
```

This function finds the adminstration 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;
};
```

This structure shall never be allocated by the caller as it may grow in future releases. Instead, creation of 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 section 9. The `lv_Node.ln_Name` element is the full path to 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`.

### 13.4.4 Deleting a Shell Variable

The `DeleteVar()` deletes a global or local shell variable or alias.

```
success = DeleteVar( name, flags )
D0                    D1    D2

BOOL DeleteVar(STRPTR, ULONG )
```

This function removes a shell variable or alias from the local or global pool of shell variables. This function is equivalent to `SetVar(name,NULL,0,flags)`.

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 separates. This allows variables to form a hierarchy similar to 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 variable 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.

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.

## 13.5 Command Line Argument Parsing

The functions in this section support applications in parsing command line arguments. While programs are free to use custom algorithms instead, 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 13.1.4 and integrate a simple help system for command line tools.

### 13.5.1 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)
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 streams. Currently, this function requires that the command line is terminated by a newline character (`0x0a`).

The `template` parameter specifies the arguments the parser attempts to pull from its input. It describes their name and type, whether they are optional or mandatory, and whether the name is required on the command line to fill the argument, or whether it can also be matched by position.

Each command line argument consists of a keyword, and an optional abbreviation that is separated from the full keyword by an equals-sign ("="), e.g. `QUICK=Q`. Additional options may follow the keyword, and are separated from it and additional options by a forwards slash ("/"). Such options describe wether a particular argument is mandatory, optional, numerical, whether the keyword is required to match an argument or whether arguments are filled by position left to right, etc. For example, `QUICK=Q/S` would indicate a boolean switch matching the keyword `QUICK` or `Q` on the command line, the `/S` indicates that it is a switch. The keywords are separated by commas (",") from each other.

This function performs parsing by matching keywords on the command line to keywords in the template; the keyword either stands for itself for boolean tags, 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 assign to the keyword `FROM` in the template the value `org`. If no keyword is present, then remaining arguments that do not explicitly require the keyword are filled left to right from the command line.

Upon return, each argument is placed in one of the elements of `array`, the first argument into the first element, the second into the second, in the order they appear in the template. While formally this an array of `LONG`s, the actual type of an element depends on the nature of the argument as specified through the options, and may 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 the function, and it shall be large enough to hold one entry per command line argument 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 toogle. The corresponding entry in `array` is set to 0 if the keyword does not appear, and changes from 0 to non-zero and back each time the keyword is detected on the command line. That is, if the keyword appears an odd number of times, the element in `array` is non-zero, otherwise it becomes 0.

/K This argument is only filled in if its keyword appears on the command line, arguments without /K 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 non-specified command line arguments.

/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 a, the second to b, the third to c and the forth 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 from 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 explictly appearing on the command line.

/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 provides a source of the arguments, and is also used for internal resource management of `ReadArgs()`. If NULL is passed in here, this function will allocate and initialize it, 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`.

For a custom command line source, an RDArgs structure as documented in `dos/rdargs.h` shall be allocated through `AllocDosObject()`. This structure looks 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 following:

```
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 element 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.

The `RDA_DAList` element of the `RDArgs` structure is private to `ReadArgs()` and shall not be read or written to. This element is initialzed to `NULL` by `AllocDosObject()`.

The `RDA_Buffer` element contains a buffer which will be used by `ReadArgs()` to keep parsed-off 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 that is printed over the output stream of the calling process if the command line to be parsed consists of a single question mark. If not present, the `template` parameter of `ReadArgs()` is printed. Following displaying the help string, `ReadArgs()` then attempts to retrieve another command line from the shell through the input stream. This feature allows users to request help on the called command.

The `RDA_Flags` element allows callers to further refine the functionality of `ReadArgs()`. The available flags are also defined in `dos/readargs.h` and are as follows:

`RDAF_STDIN`: This flag 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, this function 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 accquired during argument parsing, it shall be released by `FreeArgs()` once all arguments have been interpreted and worked on, regardless of whether the `RDArgs` structure was 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 as `rdargs` parameter, then this structure is *not* released; instead `FreeDosObject()` shall be called to dispose a user provided structure.

Amongst errors related to input and output operations, the following additional error codes can be returned in `IoErr()`:

`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.

## 13.5.2   Releasing Argument Parser Resources

The `FreeArgs()` function releases all resources allocated by `ReadArgs()`.

```
FreeArgs(rdargs)
         D1

void FreeArgs(struct RDArgs *)
```

This function releases all resources accquired by the AmigaDOS argument parser given an `RDArgs` structure. This includes temporary buffers, and — unless a custom `RDArgs` structure was passed into `ReadArgs()` — also the `RDArgs` structure itself. If a custom allocated `RDArgs` structure was provided to `ReadArgs()`, then this structure shall be disposed manually by `FreeDosObject()`, it is not released by `FreeArgs()`.

The `rdargs` argument is a pointer to a `RDArgs` structure whose resources, including potentially itself, shall be released. It is safe to pass in `NULL`, in which case this function does nothing.

This function shall be called once all arguments of a `ReadArgs()` call have been processed. Its only argument is either the result of `ReadArgs()`, or the `rdargs` structure passed into it. This function potentially releases all elements of the `array` argument of `ReadArgs()`, and thus its contents is no longer usable after calling `FreeArgs()`.

### 13.5.3   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)
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 13.5.1.

An argument is delimted by spaces, tabulators, or equals-signs ("="), unless they are enclosed in double quotes. A newline, semicolon (";") or EOF always terminates an argument. Section 13.1 provides more details on the shell syntax. The `ReadItem()` function handles some escaping rules, namely the first class of escape sequences listed in section 13.1.4. This includes the sequences `*"`, `*E`, `*N` and `**`. Once an argument delimiter is found, `ReadItem()` terminates, but places the terminator back into its source such that the caller is able to identify it.

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, this function 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 explained in section 13.5.1.

This function returns the following result codes defined in `dos/dos.h`:

`ITEM_EQUAL`: an equals-sign ("=") was found as terminator.

`ITEM_ERROR`: the supplied user buffer is too small or the source run out of data before matching a closing 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, semiolon or EOF.

`ITEM_UNQUOTED`: an unquoted item was found and retrieved from the command line.

`ITEM_QUOTED`: a quoted item was found and retrieved.

This function does not set `IoErr()`.

### 13.5.4 Find an Argument in a Template

The `FindArg()` function finds the index of a keyword in a `ReadArgs()` compatible template.

```
index = FindArg(template, keyword)
D0                      D1          D2

LONG FindArg(STRPTR, STRPTR)
```

This function scans a template as those provided to `ReadArgs()` for a specific keyword and returns the (zero-based) count of the keyword within the template. See section 13.5.1 for how templates look like.

The `template` argument a pointer to a `NUL`-terminated C string that contains the template; the syntax of the template is specfied in section 13.5.1.

The `keyword` argument provides a keyword that is to be found in the template. It is provided as a pointer to a `NUL`-terminated C string. Searching is case-insensitive.

This function returns a zero-based index which argument in the template matches the supplied keyword matches, i.e. $\tilde{0}$ for the first keyword, 1 for the second and so on. It returns −1 if no matching argument is found in the template. `IoErr()` is not touched.

## 13.6 Resident Segments

AmigaDOS keeps a list of resident segments that that are available immediately without requiring them to load from disk. This list of segments include resident commands, but also ROM-resident system segments that have been placed there during initialization of AmigaDOS through the System-startup module.

This 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[8].

Segments enter this list in multiple ways: First, commands whose p protection bit (see section 6.1) 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 but only depends on this bit. The AmigaDOS Shell scans the list of resident commands, and if the requested command is not found there, searches the *path* instead to load the command from disk.

Second, if a command has the p and h protection bits set, it is automatically added to the list of resident segments. The h stands for "hold", meaning that a command will be held resident once used.

Third, the AmigaDOS Shell adds its built-in commands during its initialization to this list. This includes elementary commands such as `CD` or `Execute` that do not need to be loaded from disk.

Fourth, some segments are added to the list during initialization of AmigaDOS through System-Startup. These include the CON-Handler, the RAM-Handler, the FFS which is named "FileHandler" there, and — finally – the Shell.

The Shell 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 make it available by default to the entire system. 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. 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 13.2.1.

A resident segment is described by the `Segment` structure defined in `dos/dos.h`:

_____

[8]This duplication of structures is again a historic accident due to the origin of AmigaDOS.

```
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. Its elements have the following function:

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. This BPTR shall not be modified by the user.

The `seg_UC` element is a use counter. It is incremented every time the segment is used, e.g. when the shell locates a command in the resident segments, 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 [9].

Despite these regular entries used for disk-based commands made resident, 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 avaiable 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 `CMD_SYSTEM` are not commands that could be executed, but rather segments of system components. The RAM-Handler, the CON-Handler and the FFS aka 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 rather marked as temporarily disabled. This type is created if a built-in command or a system segment is "removed" by the `Resident` command.

The `seg_Seg` element is a BPTR to the actual segment list corresponding to the loaded or resident command or system segment. It is a segment list as those loaded through `LoadSeg()`.

The `seg_Name` is, actually, not only 4 characters long as indicated, but a variable length field. It contains the name of the command or system segment as BSTR.

### 13.6.1 Finding a Resident Segment

The `FindSegment()` locates a resident segment by name.

```
segment = FindSegment(name, start, system)
D0                     D1     D2     D3

struct Segment *FindSegment(STRPTR, struct Segment *, LONG)
```

This function locates the segment named `name` in the list of resident segments, scanning either the entire list if `start` is `NULL`, or at all segments behind 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.

The `system` flag indicates whether the function scans for regular segments or system/internal segments. If `system` is `DOSFALSE`, only regular entres 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,

---

[9]The `Resident` command prints `seg_UC` - 1 instead of `seg_UC`.

the above function does not attempt to filter out disabled segments; if required, this must 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 should be called while a `Forbid()` is active, and the caller shall increment `seg_UC` for non-system segments it attempts to use them 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, but it sets it to `ERROR_OBJECT_NOT_FOUND` in case no matching segment could be found.

### 13.6.2  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)
D0                    D1       D2       D3

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

This function adds the segment `seglist`, e.g. as returned by `LoadSeg()`, under the given `name` to the list of resident segments. The `seg_UC` type and use counter is initialized to the value `type`.

The `name` is copied and may be released 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 that are made resident and by that available to other processors, `type` shall be set to $1$[10].

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 use. 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`.

### 13.6.3  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)
D0                    D1

BOOL RemSegment(struct Segment *)
```

---

[10]The official autodocs and [1] are wrong in this regard, they suggest 0, but this value is incorrect and not even valid.

This function attempts to remove the resident segment provided as argument from the AmigaDOS list of resident commands. 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 is left 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 accquired 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 should stop multitasking with `Forbid()`, then locate the segment with `FindSegment()`, and if this function succeeds, unload the found segment with `RemSegment()`. Then, finally, `Permit()` should be called to re-enable multitasking.

If segment passed in cannot be removed because it is in use or a system segment, this function returns `DOSFALSE` and sets `IoErr()` to `ERROR_OBJECT_IN_USE`. On success, it returns `DOSTRUE`[11].

## 13.7   Writing Custom Shells

AmigaDOS allows adding custom shells to the system that may optionally also replace the AmigaDOS default shell, the "boot shell". A shell is a resident system segment, see section 13.6, similar to the CON-Handler or the RAM-Handler on the same list. When launching a shell with the `System()` function, 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 the necessary steps:

```
resident shell MyShell replace system
```

where `MyShell` is the file name of your shell. However, a shell is not a regular program otherwise and shall be implemented according to the following guidelines:

First of all, AmigaDOS supports a BCPL shell under the name "`CLI`" which will be initialized as BCPL programs using the BCPL runtime binding protocol explained in section 10.5.3. 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 C or 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 10.5.3 provides a minimal interface for the BCPL runtime binding, and is also safe to use for C shells. For the following example code, it is assumed that this startup code is the first segment of the shell code, and that its `main()` function is called through this minimal interface. AmigaDOS, in its current incarnation, initializes the shell startup packet as follows:

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 type that still exists. If, deriving from the above, `dp_Type` is zero, then this indicated the creation of the "Initial CLI", and if `dp_Res1` is non-zero, this used to indicate the creation of a shell through `NewShell`. These two startup types no longer exist and are no longer in use. All other startup packet types are not used by AmigaDOS.

In addition, the current AmigaDOS Shell launches instances of itself as new processes, for example to implement pipes or the run-back operator ("`&`"), see sections 13.1.2 and 13.1.3, and to indicate this private startup mechanism, the AmigaDOS Shell uses an even different combination of values in these three elements. 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 AmigaDOS startup packet.

---

[11]Due to a defect in the current version of the *dos.library* it unfortunately *also* sets `IoErr()` to the same error code.

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`, and then reply the packet by sending it back to its origin.

If this first initialization succeeds, the segment array of the calling process in `pr_SegList` needs to be reorganized. The regular process startup code of AmigaDOS places the segment list of the process 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, see section 13.3.7.

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[12] 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 the further handling on the startup packet, namely when exactly to send it back to the caller. More on this insection 13.7.1. At this point, the shell is initialized as far as AmigaDOS is concerned and can start its work.

The following code implements minimal shell startup handling:

```
void __asm main(register __a0 struct DosPacket *pkt)
{
  struct Library *DOSBase;
  struct Library *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 */
      /* In worst case, run into Alert()              */
      ...
  } else {
   LONG fn;
   BPTR segs;
   /* Perform shell initialization, here an example */
   DOSBase = OpenLibrary("dos.library",47);
   if (DOSBase == NULL) {
     /* Initialization failed, return a meaningful  */
     /* error by replying the packet                */
```

---

[12]This communication protocol is surely needlessly bizarre, and this case a legacy of Kickstart 2.0 and not part of the Tripos shell startup.

```
    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(process->pr_SegList);
  /* move shell's seg list to the next slot */
  if (segarray[4] == NULL) {
    segarray[4] = segarray[3];
    segarray[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(DOSBase);
    /* The Packet is already replied       */
    return;
  }
  /* Main function continues               */
  ...
}
```

When the shell replies the startup packet and when it terminates depends on the flags returned by `CliInitRun()`. This is described in the following section.

### 13.7.1   Initializing a new Shell

The `CliInitRun()` function initializes a process and its `CommandLineInterface` structure from a shell startup package.

```
flags = CliInitRun( packet )
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 packet are intentionally not documented here, and AmigaDOS may change or extend the packet in the future.

This function 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, names for these bits are suggested here:

**Table 147: CliInitRun() flags**

| Acronym | Bit Number | Notes |
|---------|-----------|-------|

| FN_VALID | 31 | indiates 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 accquired 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 executed by the shell, 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 the `cli_StandardInput` file handles of the `CommandLineInterface` structure. This type of startup is typically used when the shell is instructed to execute commands in the background, e.g. by the `Run` command, and if `System()` is called with `NULL` as its first argument, see section 13.2.1.

The reason for delaying the startup packet after execution 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 command that executes 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 this 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, and 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()`, or `System()` with a `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.

Historically, two additional functions existed to initialize a shell. `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.

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()`.

# Chapter 14

# Miscellaneous Functions

In this section, miscellaneous functions are specified that logically do not belong into any other section. These are constructur and destructor functions for AmigaDOS objects, checking for signals or retrieving error strings and generating error requesters.

## 14.1 Object Constructors and Destructors

AmigaDOS objects should not be allocated manually through `AllocMem()` or other generic memory allocation functions as they do not initialize the objects correctly nor are they aware of undocumented internal extensions to such objects. Thus, the `sizeof` operator of the C language does not necessarily reflect the number of bytes to be allocated as AmigaDOS placed undocumented elements at the end of the officially documented structures.

### 14.1.1 Allocating a DOS Object

The `AllocDosObject()` function constructs instances of various types used by the *dos.library* and its components. 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. They all correspond to the same entry in the *dos.library* and only differ in the calling convention.

The `AllocDosObject()` and `AllocDosObjectTags()` functions are identically and provide 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. The compiler takes these arguments, and passes it as a tag list into the entry point of the *dos.library*.

```
ptr = AllocDosObject(type, tags)
D0                     D1    D2

void *AllocDosObject(ULONG, struct TagItem *)

ptr = AllocDosObjectTagList(type, tags)
D0                            D1    D2

void *AllocDosObjectTagList(ULONG, struct TagItem *)

ptr = AllocDosObjectTags(type, Tag1, ...)
```

```
void *AllocDosObjectTags(ULONG, ULONG, ...)
```

The above functions all create objects used by the *dos.library* and initializes them according to additional arguments. The `type` argument provides the type of the object to be constructed; types are defined in `dos/dos.h`:

**Table 148:** **AllocDosObject() type IDs**

| Type | Description |
|------|-------------|
| DOS_FILEHANDLE | construct a `FileHandle` structure, as in 4.8.1 |
| DOS_EXALLCONTROL | construct a `ExAllControl` structure, see 6.1.4 |
| DOS_FIB | construct a `FileInfoBlock` structure, see 6.1 |
| DOS_STDPKT | construct a `StandardPacket` structure, as in 11.1.2 |
| DOS_CLI | construct a `CommandLineInterface` structure, see 13.3 |
| DOS_RDARGS | construct a `RDArgs` structure, as defined in section 13.5.1 |

The `DOS_FILEHANDLE` type creates a `FileHandle` structure. This structure describes an open file and shall only be created through this function as it contains some hidden elements. This type takes 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 10 which corresponds to the second argument of the `Open()` function specified in section 4.4.1. It defaults to `ACTION_FINDINPUT`, i.e. a file that is opened nonexclusively for reading.

The `DOS_EXALLCONTROL` type creates an `ExAllControl` structure used to iterate over directory contents, see section 6.1.4; no additional tags are required.

The `DOS_FIB` type creates a `FileInfoBlock` structure as defined in section 6.1. 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. The macro `D_S()` in section 2.4 should be used if the memory for this structure is to be taken from the stack. This macro ensures the required alignment. No further tags are necessary.

The `DOS_STDPKT` 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. There is no need to use exactly this structure for handler communication provided the `DosPacket` is aligned to a long-word boundary and linked to a `Message`, see section 11.1.2. This function 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` structure creates a `CommandLineInterface` structure as defined in section 13.3. No additional tags are recgonized. While it was possible to construct these structures manually in the past, this is no longer the case with the current version of AmigaDOS as this structure has been extended. A shell which is equipped with a manually constructed `CommandLineInterface` structure will not offer the full functionality of a regular AmigaDOS Shell.

The `DOS_RDARGS` structure creates a `RDArgs` structure used by the `ReadArgs()` function for command line argument parsing, see section 13.5.1. This constructor is also called internally by the *dos.library*, but if a custom `RDArgs` structure is required to parse, 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.

This function returns on succes the pointer to the constructed object. Note that for `DOS_STDPKT` this is a pointer to a `DosPacket` structure and not to the `StandardPacket` structure. It does not touch `IoErr()` on success. On error, this function returns `NULL` and sets `IoErr()` to `ERROR_NO_FREE_STORE`.

### 14.1.2  Releasing a DOS Object

The `FreeDosObject()` function destroys an AmigaDOS object and releases the resources associated to it.

```
FreeDosObject(type, ptr)
               D1    D2

void FreeDosObject(ULONG, void *)
```

This function destroys an object created by `AllocDosObject()`. The `type` argument is one of the types from table 148 in section 14.1.1, and `ptr` is a pointer to the object to be destroyed. In case type is `DOS_STDPKT`, the `ptr` argument is a pointer to the `DosPacket` element of the `StandardPacket` aggregate.

Passing `NULL` as `ptr` performs nothing, the function exits in this case without performing any action. This function does not set `IoErr()`.

## 14.2   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 to be printed on the console, or by creating a requester with a message on a screen.

### 14.2.1   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 a the response of the user who may either abort or retry an activity.

```
status = ErrorReport(code, type, arg1, device)
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 cancelation of the activity is desired. This function is typically called by AmigaDOS handlers attempting to generate an error requester, or the *dos.library* itself. However, user code may also use this function to generate an AmigaDOS standard requester.

The `code` argument is an error identifier returned by `IoErr()`. Only a subset of the error identifies listed in section 9.2.9 are supported by this function; they are defined in the include files `dos/dos.h` and `dos/dosextends.h`. Table 149 lists them:

**Table 149: 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 full |
| 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 a drive |
| ABORT_DISK_ERROR | Reports that a medium has a physical read/write error |
| ABORT_BUSY | Requests to insert a particular medium into a drive |

The `type` argument provides 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 150: Error sources supported by ErrorReport()**

| Type | Description |
|---|---|
| REPORT_STREAM | `arg1` is a BPTR to a `FileHandle` structure |
| REPORT_LOCK | `arg1` is a BPTR to a `FileLock` structure |
| REPORT_VOLUME | `arg1` is a BPTR[a] to a `DosList` structure |
| REPORT_INSERT | `arg1` is a *regular* pointer to a path separated at ":" |

[a]Unlike what the official documentation says, this is a BPTR, not a regular pointer.

The `REPORT_INSERT` type shall be a pointer to an absolute path name containing a colon (":") from which a volume or device name is extracted.

The `device` argument is a (regular) pointer to a `MsgPort` structure that is only used if `type` is `REPORT_LOCK` and `arg1` is `ZERO`. This port is assumed to be a `MsgPort` of a file handler that is contacted to learn about the name of the currently inserted volume. This could be the `pr_FileSystemTask` of the calling process which contains the `MsgPort` of the file handler responsible for the `ZERO` lock.

This function returns a boolean indicator whether the currently ongoing operation should be aborted or retried. If the result code is `DOSFALSE`, the operation should be retried. On return, this function also sets `IoErr()` to the error identifier in `code`.

### 14.2.2 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, len)
D0          D1      D2      D3      D4

LONG Fault(LONG, STRPTR, STRPTR, LONG)
```

This function fills the `buffer` that is `len` bytes long with an optional `header`, followed by a colon (":") followed by a string generated from the error `code` given as first argument.

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 the `buffer`. If no localized error message is available, the numeric value of the error is inserted instead.

The `len` of `buffer` shall not be 0. It includes the `NUL` byte terminating the string created in `buffer`.

Unlike documented, the return value of this function is *not* the number of characters inserted into the buffer and is thus not usable. As a workaround, to find the length of the inserted string, use `strlen()` on the buffer if `code` is non-zero:

```
if (code) {
  Fault(code,header,buffer,len);
  len = strlen(buffer);
} else {
  len = 0;
}
```

Note that `buffer` remains unchanged for a 0 argument of `code`, and thus `strlen()` does not return a reliable result in this case.

### 14.2.3 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 channel of the calling process.

```
success = PrintFault(code, header)
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. If that stream does not exist, the error is reported over the standard output stream `pr_COS` of the caller.

If `header` is non-NULL, this string is printed first, followed by a colon and a space ("： "). If a textual description of the error code `code` is available, it is printed behind it. Otherwise, just a generic error message with the error value is printed.

If `code` is 0, nothing is printed at all. Otherwise, this function sets `IoErr()` to `code`.

This function returns a boolean success code. It currently is, however, always non-zero indicating success, and thus not particularly helpful.

### 14.2.4 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)
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` of the caller.

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.

# Chapter 15

# 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 their elements.

## 15.1    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;
    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 as it is defined in `exec/libraries.h`.

The `dl_Root` pointer points to the `RootNode` structure documented in section 15.2. Most useful information can be reached from there.

The `dl_GV` is the pointer to the (fake) BCPL *global vector* which contains a table of (BCPL compiled) 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.

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 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 subroutine" function.

The `dl_Errors` pointer provides (localized) strings for all messages the *dos.library* generates, including other localized messages. This element is private and shall not be accessed. Instead, localized error messages can be retrieved by the `Fault()` function of section 14.2.2 which accesses this string.

The `dl_TimeReq` element is a private pointer to a `timerequest` structure the *dos.library* uses for functions such as `Delay()` in section 3.1.3. It shall not be used by clients.

The `dl_UtilityBase` element provides a pointer to the *utility.library* the *dos.library* uses for parsing tag lists.

The `dl_IntuitionBase` is the pointer to the *intuition.library* that is used to generate requesters.

## 15.2   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;
    struct MinList rn_CliList;
    struct MsgPort *rn_BootProc;
    BPTR    rn_ShellSegment;
    LONG    rn_Flags;
};
```

The `rn_TaskArray` element is a BPTR the (legacy) Tripos process table, and is part of the complete table reachable through `rn_CliList`. Instead of going through these elements, the `FindCliProc()` function specified in section 13.2.6 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 are pointers to `Process` structures (see section 9) or `NULL`. This array is indexed by the CLI number, though only processes that are associated to a shell are listed here.

The `rn_ConsoleSegment` is the pointer to the BCPL entry point of the shell. Even though this pointer is initialized, it is no longer used. Instead, the shell is located on the resident segments with `FindSegment()` of section 13.6.1. The corresponding resident segment has the name "CLI".

The `rn_Time` contains the current time of the system, it is updated every time `DateStamp()` (see section 3.1.1 is called, and this is also the recommended way of obtaining the time instead of copying it from here.

The `rn_RestartSeg` is obsolete and no longer in used. This used to be a BCPL program that performed validation of inserted disks and restarted the file system. Today, this function is integrated into the file system itself.

The `rn_Info` element is a BPTR to the `DosInfo` structure described in section 15.3. It contains further global information.

The `rn_FileHandlerSegment` element is the BPTR to the segment of the ROM default AmigaDOS file system. This element is no longer in use today. This element is filled from the resident segment "File-Handler", and is also updated if a newer version of the FFS is found in the RDB of the booting disk. Today, file systems are found through the "FileSystem.resource" and not through this BPTR.

The `rn_CliList` contains the full process table containing all shell processes accessible through `FindCliProc()`. This element is a `MinList` structure as defined in `exec/lists.h`. Each node on this list keeps a consecutive number of shells 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 creating 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 kept in `cpl_Array[1]`.

The `cpl_Array` element contains pointers to the `Process` structures of the shell, except that the `cpl_Array[0]` entry is the number of shell processes adminstrated in this node. The actual process pointers start from the second entry onwards.

The `cpl_Array` of the first `CliProcList` structure is also accessible through rn_TaskArray, and the nodes kept here are part of an extension that overcomes the limitation of at most 20 shells in AmigaDOS 1.3 and before.

The `rn_BootProc` element is a pointer to the `MsgPort` of the file system that booted the system. This is not necessary the file system whose segment is recorded in `rn_FileHandlerSegment`. It is different if the booting file system is not the ROM-based FFS 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 entry point of the shell. This pointer is also not used anymore, but still intialized. 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. Today, 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 section 8 supports "*" 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.

## 15.3   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 section 7 and the resident segments from section 13.6.

```
struct DosInfo {
    BPTR    di_McName;
    BPTR    di_DevInfo;
    BPTR    di_Devices;
    BPTR    di_Handlers;
    APTR    di_NetHand;
    struct  SignalSemaphore di_DevLock;
    struct  SignalSemaphore di_EntryLock;
    struct  SignalSemaphore di_DeleteLock;
};
```

Unlike what the official includes suggest, `di_McName` is *not* the list of resident segments. It is instead unused.

The `di_DevInfo` element is a BPTR to the first entry of the device list introduced in section 7. 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 7.3.

The `di_Devices` and `di_Handlers` elements are Tripos legacies and not in use.

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 13.6, 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 13.6.

The `di_DevLock`, `di_EntryLock` and `di_DeleteLock` semaphores are private to the *dos.library*. They are used by the `LockDosList()` function of section 7.3.1.

# Bibliography

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

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

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

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

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

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

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

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

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

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

# Index

* (file name), 19, *19*, 83, 98, 99, 103, 118, 129, 134, 144, 195, 197, 198, 273–276, 281–283, 317
.BRA, *279*
.DEF, *279*
.DEFAULT, *279*
.DOL, *279*
.DOLLAR, *279*
.DOT, *278*
.K, 279
.KET, 279, *279*
`ACTION_EXAMINE_ALL`, 240

ABORT_BUSY, 311
ABORT_DISK_ERROR, 311
AbortPkt(), *192*
ACCESS_READ, *48*
AChain, 119, *119*, 120
ACION_DISK_INFO, 258
ACTION_ADD_NOTIFY, *251*
ACTION_CHANGE_MODE, *235*
ACTION_CHANGE_SIGNAL, 266, *266*
ACTION_CLOSE, 266
ACTION_COPY_DIR, *232*
ACTION_COPY_DIR_FH, *233*
ACTION_CREATE_DIR, *234*
ACTION_CURRENT_VOLUME, *252*
ACTION_DELETE_OBJECT, *248*, 251
ACTION_DIE, 197, 206, 269, *269*, 270
ACTION_DISK_INFO, 55, 56, 199, *254*, 259, 262, 265, 267
ACTION_DROP, *265*
ACTION_END, 196, *227*
ACTION_EXAMINE, 237
ACTION_EXAMINE_ALL, *238*, 239, 240
ACTION_EXAMINE_ALL_END, 239, *239*, 240
ACTION_EXAMINE_FH, *240*
ACTION_EXAMINE_NEXT, *237*, 238, 239
ACTION_EXAMINE_OBJECT, *236*, 237, 238, 241
ACTION_EXAMINE_OJBECT, 237

ACTION_FH_FROM_LOCK, *227*
ACTION_FINDINPUT, 196, 204, 225, *225*, 226, 227, 235, 310
ACTION_FINDOUTPUT, 196, 204, 226, *226*, 235, 251
ACTION_FINDUPDATE, 196, 204, 227, *227*, 235
ACTION_FLUSH, *268*, 269
ACTION_FORCE, 201, 264, *264*, 265
ACTION_FORMAT, 206, 255, *255*, 256, 267, 268
ACTION_FREE_LOCK, *235*
ACTION_FREE_RECORD, *231*
ACTION_INFO, *253*, 254, 267
ACTION_INHIBIT, 114, 207, *267*
ACTION_IS_FILESYSTEM, *268*
ACTION_KEY, *228*
ACTION_LOCATE_OBJECT, *231*, 233, 268
ACTION_LOCK_RECORD, *230*
ACTION_MAKE_LINK, *241*
ACTION_MORE_CACHE, 206, *266*
ACTION_NIL, 194, 270, *270*
ACTION_PARENT, *233*
ACTION_PARENT_FH, *234*
ACTION_QUEUE, 263, *263*, 264, 265
ACTION_READ, 199, 257, 263, 264
ACTION_READ_LINK, 76, 242, *242*, 243
ACTION_READ_RETURN, *270*
ACTION_REMOVE_NOTIFY, 252, *252*
ACTION_RENAME, 251
ACTION_RENAME_DISK, *254*
ACTION_RENAME_OBJECT, *247*
ACTION_SAME_LOCK, *234*
ACTION_SCREEN_MODE, *257*
ACTION_SEEK, *229*
ACTION_SERIALIZE_DISK, *255*, 267
ACTION_SET_COMMENT, *249*, 251
ACTION_SET_DATE, *249*, 251
ACTION_SET_FILE_SIZE, *229*, 251
ACTION_SET_OWNER, 206, *250*, 251
ACTION_SET_PROTECT, *248*, 251