

# AmigaDOS Packet Interface Specification

by John Toebe

AmigaDOS communicates with file systems and other DOS handlers by sending and receiving *packets*. Opening and closing file handles (including console file handles), creating directories, and renaming disks all require DOS to tell a handler to perform these actions through sending a packet. The particular action a handler performs depends on the type of packet it receives.

This article documents the standard AmigaDOS packet types. For information on how to use packets to communicate with handlers see the *AmigaDOS Manual*.

Packets sent to a file system or handler can be divided into several basic categories:

- **Basic Input/Output**

These actions deal with transferring data to and from objects controlled by the handler.

- **File/Directory Manipulation/Information**

These actions are used to gain access to and manipulate the high level structures of the file system.

- **Volume Manipulation/Information**

These actions allow access to the specific volume controlled by the file system.

- **Handler Maintenance and Control**

These allow control over the handler/file system itself, independent of the actual volume or structure underneath.

- **Handler Internal**

These actions are never sent to the handler directly. Instead they are generally responses to IO requests made by the handler. The handler makes these responses look like packets in order to simplify processing.

- **Obsolete Packets**

These packets are no longer valid for use by handlers and file systems.

- **Console Only Packets**

These packets are specific to console handlers. File Systems can ignore these packets.

Much of this information can be extracted from Developer Conference notes, *The AmigaDOS Manual*, and various Fred Fish disks. However, because there is no single complete reference to these packet types, a consolidated view of all the packets is presented here. Several structures are referenced here which can be found by looking at the include files *<dos/dos.h>* and *<dos/dosextens.h>*. (If you are using the 1.3 version of the include files, these are in the *libraries* directory instead of the *dos* directory). Before attempting to work with a file handler you should first become familiar with these files.

Each packet type documented in this article is listed with its action name, its corresponding number, any AmigaDOS routines which uses this packet, and the list of parameters that the packets uses. The C variable types for the packet parameters are one of the following types:

**BPTR** This is BCPL pointer (the address of the given object shifted right by 2).  
Note: this means that the object must be aligned on a longword boundary.

**LOCK** This is a BPTR to a FileLock structure returned by a previous `ACTION_LOCATE_OBJECT`. A lock of 0 is legal, *indicating the root of the volume for the handler*.

**BSTR** This is a BPTR to a string where the first byte indicates the number of characters in the string. This length byte is unsigned but because it is stored in a byte, the strings are limited to 255 characters in length.

**BOOL** A 32-bit boolean value either containing `DOSTRUE` (-1) or `DOSFALSE` (0).  
Note: equality comparisons with `DOSTRUE` should be avoided.

**CODE** A 32 bit error code as defined in the *dos/dos.h* include file. Handlers should not return error codes besides those defined in *dos/dos.h*.

**ARG1** The FileHandle->fh\_Arg1 field.

**LONG** A 32 bit integer value.

## Basic Input/Output

The Basic Input/Output actions are supported by both handlers and file systems. In this way, the application can get a stream level access to both devices and files. One difference that arises between the two is that a handler will not necessarily support an `ACTION_SEEK` while it is generally expected for a file system to do so.

These actions work based on a FileHandle which is filled in by one of the three forms of opens:

```
ACTION_FINDINPUT      1005      Open(..., MODE_OLDFILE)
ACTION_FINDOUTPUT     1006      Open(..., MODE_NEWFILE)
ACTION_FINDUPDATE     1004      Open(..., MODE_READWRITE)
ARG1: BPTR FileHandle to fill in
ARG2: LOCK Lock on directory that ARG3 is relative to
ARG3: BSTR Name of file to be opened (relative to ARG1)
```

```
RES1: BOOL Success/Failure (DOSTRUE/DOSFALSE)
RES2: CODE Failure code if RES1 is DOSFALSE
```

All three actions use the lock (ARG2) as a base directory location from which to open the file. If this lock is `NULL`, then the file name (ARG3) is relative to the root of the current volume. Because of this, file names are not limited to a single file name but instead can include a volume name (followed by a colon) and multiple slashes allowing the file system to fully resolve the name. This eliminates the need for AmigaDOS or the application to parse names before sending them to the file system. Note that the lock in ARG2 must be associated with the file system in question. It is illegal to use a lock from another file system.

The calling program owns the file handle (ARG1). The program must initialize the file handle before trying to open anything (in the case of a call to `Open()`, AmigaDOS allocates the file handle automatically and then frees it in `Close()`). All fields must be zero except the fh\_Pos and fh\_End fields which should be set to -1. The `Open()` function fills in the fh\_Type field with a pointer to the MsgPort of the handler process. Lastly, the handler must initialize fh\_Arg1 with something that allows the handler to uniquely locate the object being opened (normally a file). This value is implementation specific. This field is passed to the `READ/WRITE/SEEK/ END/TRUNCATE` operations and not the file handle itself.

`FINDINPUT` and `FINDUPDATE` are similar in that they only succeed if the file already exists. `FINDINPUT` will open with a shared lock while `FINDUPDATE` will open it with a shared lock but if the file doesn't exist, `FINDUPDATE` will create the file. `FINDOUTPUT` will always open the file (deleting any existing one) with an exclusive lock.

```
ACTION_READ           'R'      Read(...)
ARG1: ARG1 fh Arg1 field of the opened FileHandle
ARG2: APTR Buffer to put data into
ARG3: LONG Number of bytes to read
```

```
RES1: LONG Number of bytes read.
      0 indicates EOF.
```

-1 indicates ERROR  
RES2: CODE Failure code if RES1 is -1

This action extracts data from the file (or input channel) at the current position. If fewer bytes

remain in the file than requested, only those bytes remaining will be returned with the number of bytes stored in RES1. The handler indicates an error is indicated by placing a -1 in RES1 and the error code in RES2. If the read fails, the current file position remains unchanged. Note that a handler may return a smaller number of bytes than requested, even if not at the end of a file. This happens with interactive type file handles which may return one line at a time as the user hits return, for example the console handler, CON:.

**ACTION\_WRITE**                    'W'            **Write(...)**  
ARG1: BPTR fh\_Arg1 field of the opened file handle  
ARG2: APTR Buffer to write to the file handle  
ARG3: LONG Number of bytes to write  
  
RES1: LONG Number of bytes written.  
RES2: CODE Failure code if RES1 not the same as ARG3

This action copies data into the file (or output channel) at the current position. The file is automatically extended if the write passes the end of the file. The handler indicates failure by returning a byte count in RES1 that differs from the number of bytes requested in ARG3. In the case of a failure, the handler does not update the current file position (although the file may have been extended and some data overwritten) so that an application can safely retry the operation.

**ACTION\_SEEK**                    1008            **Seek(...)**  
ARG1: BPTR fh\_Arg1 field of the opened FileHandle  
ARG2: LONG New Position  
ARG3: LONG Mode: OFFSET\_BEGINNING, OFFSET\_END, or OFFSET\_CURRENT  
  
RES1: LONG Old Position. -1 indicates an error  
RES2: CODE Failure code if RES1 = -1

This packet sets the current file position. The new position (ARG2) is relative to either the beginning of the file (OFFSET\_BEGINNING), the end of the file (OFFSET\_END), or the current file position (OFFSET\_CURRENT), depending on the mode set in ARG3. Note that ARG2 can be negative. The handler returns the previous file position in RES1. Any attempt to seek past the end of the file will result in an error and will leave the current file position in an unknown location.

**ACTION\_END**                    1007            **Close(...)**  
ARG1: BPTR fh\_Arg1 field of the opened FileHandle  
  
RES1: LONG DOSTRUE

2.0 only

This packet closes an open file handle. This function generally returns a DOSTRUE as there is little the application can do to recover from a file closing failure. If an error is returned under 2.0, DOS will not deallocate the file handle. Under 1.3, it does not check the result.

**ACTION\_LOCK\_RECORD**            2008            **LockRecord(fh,pos,len,mod,tim)**  
ARG1: BPTR FileHandle to lock record in  
ARG2: LONG Start position (in bytes) of record in the file  
ARG3: LONG Length (in bytes) of record to be locked  
ARG4: LONG Mode  
          0 = Exclusive  
          1 = Immediate Exclusive (timeout is ignored)  
          2 = Shared  
          3 = Immediate Shared (timeout is ignored)  
ARG5: LONG Timeout period in AmigaDOS ticks (0 is legal)  
  
RES1: BOOL Success/Failure (DOSTRUE/DOSFALSE)  
RES2: CODE Failure code if RES1 is DOSFALSE

This function locks an area of a file in either a sharable (indicating read-only) or exclusive (indicating read/write) mode. Several sharable record locks from different file handles can exist simultaneously on a particular file area but only one file handle can have exclusive record locks on a particular area at a time. The "exclusivity" of an exclusive file lock only applies to record locks from other file handles, not to record locks within the file handle. One file handle can have any number of overlapping exclusive record locks. In the event of overlapping lock ranges, the entire range must be lockable before the request can succeed. The timeout period (ARG5) is the number of AmigaDOS ticks (1/50 second) to wait for success before failing the operation.

**ACTION\_FREE\_RECORD**            2009            **FreeRecord(file,pos,len)**  
2.0 only    ARG1: BPTR FileHandle to unlock record in  
          ARG2: LONG Start position (in bytes) of record in the file  
          ARG3: LONG Length of record (in bytes) to be unlocked  
  
RES1: BOOL Success/Failure (DOSTRUE/DOSFALSE)  
RES2: CODE Failure code if RES1 is DOSFALSE

This function unlocks any previous record lock. If the given range does not represent one that is currently locked in the file, ACTION\_FREE\_RECORD returns an error. In the event of multiple locks on a given area, only one lock is freed.

**ACTION\_SET\_FILE\_SIZE**            1022            **SetFileSize(file,off,mode)**  
ARG1: BPTR FileHandle of opened file to modify  
ARG2: LONG New end of file location based on mode  
ARG3: LONG Mode. One of OFFSET\_CURRENT, OFFSET\_BEGIN, or OFFSET\_END  
  
RES1: BOOL Success/Failure (DOSTRUE/DOSFALSE)  
RES2: CODE Failure code if RES1 is DOSFALSE

This function is used to change the physical size of an opened file. ARG2, the new end-of-file position, is relative to either the current file position (OFFSET\_CURRENT), the beginning of the file

(`OFFSET_BEGIN`), or the end of the file (`OFFSET_END`), depending on the mode set in `ARG3`. The current file position will not change unless the current file position is past the new end-of-file position. In this case, the new file position will move to the new end of the file. If there are other open file handles on this file, `ACTION_SET_FILE_SIZE` sets the end-of-file for these alternate file handles to either their respective current file position or to the new end-of-file position of the file handle in `ARG1`, whichever makes the file appear longer.

## Directory/File Manipulation/Information

The directory/file actions permits an application to make queries about and modifications to handler objects. These packets perform functions such as creating subdirectories, resolving links, and filling in `FileInfoBlock` structures for specific files.

```
ACTION_LOCATE_OBJECT      8      Lock (...)
ARG1:  LOCK      Lock on directory to which ARG2 is relative
ARG2:  BSTR      Name (possibly with a path) of object to lock
ARG3:  LONG      Mode:  ACCESS_READ/SHARED_LOCK, ACCESS_WRITE/EXCLUSIVE_LOCK

RES1:  LOCK      Lock on requested object or 0 to indicate failure
RES2:  CODE      Failure code if RES1 = 0
```

The AmigaDOS function `Lock()` uses this action to create its locks. Given a name for the object, which may include a path, (`ARG2`) and a lock on a directory from which to look for the name (and path), `ACTION_LOCATE_OBJECT` will locate the object within the file system and create a `FileLock` structure associated with the object. If the directory lock in `ARG1` is `NULL`, the name is relative to the root of the file handler's volume (a.k.a. ``:'). The memory for the `FileLock` structure returned in `RES1` is maintained by the handler and freed by an `ACTION_FREE_LOCK`. Although it's not a requirement, if an handler expects to support the pre-1.3 *Format* command, it must accept any illegal mode as `ACCESS_READ`.

A handler can create an exclusive lock only if there are no other outstanding locks on the given object. Once created, an exclusive lock prevents any other locks from being created for that object. In general, a handler uses the `FileLock->fl_Key` field to uniquely identify an object. Note that some applications rely on this (although a handler is not required to implement this packet).

The `fl_Volume` field of the returned `FileLock` structure should point to the DOS device list's volume entry for the volume on which the lock exists. In addition, there are several diagnostic programs that expect all locks for a volume to be chained together off the `dl_LockList` field in the volume entry. *Note that relying on this chaining is not safe, and can cause serious problems including a system crash. No application should use it.*

```
ACTION_COPY_DIR           19      DupLock (...)
ARG1:  LOCK      Lock to duplicate

RES1:  LOCK      Duplicated Lock or 0 to indicate failure
RES2:  CODE      Failure code if RES1 = 0
```

This action's name is misleading as it does not manipulate directories. Instead, it creates a copy of a *shared* lock. The copy is subsequently freed with an `ACTION_FREE_LOCK`. Note that it is valid to pass a `NULL` lock. Currently, the `DupLock()` call always returns 0 if passed a 0, although a handler is not required to return a 0.

```
ACTION_FREE_LOCK          15      UnLock (...)
ARG1:  LOCK      Lock to free

RES1:  BOOL      TRUE
```

This action frees the lock passed to it. The AmigaDOS function `Unlock()` uses this packet. If passed a `NULL` lock, the handler should return success.

```
ACTION_EXAMINE_OBJECT     23      Examine (...)
ARG1:  LOCK      Lock of object to examine
ARG2:  BPTR      FileInfoBlock to fill in

RES1:  BOOL      Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE      Failure code if RES1 = DOSFALSE
```

This action fills in the `FileInfoBlock` with information about the locked object. The `Examine()` function uses this packet. This packet is actually used for two different types of operations. It is called to obtain information about a given object while in other cases, it is called to prepare for a sequence of `EXAMINE_NEXT` operations in order to traverse a directory.

This seemingly simple operation is not without its quirks. One in particular is the `FileInfoBlock->fib_Comment` field. This field used to be 116 bytes long, but was changed to 80 bytes in release 1.2. The extra 36 bytes lie in the `fib_Reserved` field. Another quirk of this packet is that both the `fib_EntryType` and the `fib_DirEntryType` fields must be set to the same value, as some programs look at one field while other programs look at the other.

File systems should use the same values for `fib_DirEntryType` as the ROM file system and ram-handler do. These are as follows:

```
ST_ROOT          1
ST_USERDIR       2
ST_SOFTLINK      3 NOTE: this Shows up as a directory unless checked for
explicitly
ST_LINKDIR       4
ST_FILE          -3
ST_LINKFILE      -4
```

Also note that for directories, handlers *must* use numbers greater than 0, since some programs test to see if `fib_DirEntryType` is greater than zero, ignoring the case where `fib_DirEntryType` equals 0. Handlers should avoid using 0 because it is not interpreted consistently.

```
ACTION_EXAMINE_NEXT       24      ExNext (...)
ARG1:  LOCK      Lock on directory being examined
ARG2:  BPTR      BPTR FileInfoBlock
```



```
RES1:  BOOL    Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE    Failure code if RES1 = DOSFALSE
```

The `ExNext()` function uses this packet to obtain information on all the objects in a directory. `ACTION_EXAMINE` fills in a `FileInfoBlock` structure describing the first file or directory stored in the directory referred to in the lock in `ARG1`. `ACTION_EXAMINE_NEXT` is used to find out about the rest of the files and directories stored in the `ARG1` directory. `ARG2` contains a pointer to a valid `FileInfoBlock` field that was filled in by either an `ACTION_EXAMINE` or a previous `ACTION_EXAMINE_NEXT` call. It uses this structure to find the next entry in the directory. This packets writes over the old `FileInfoBlock` with information on the next file or directory in the `ARG2` directory. `ACTION_EXAMINE_NEXT` returns a failure code of `ERROR_NO_MORE_ENTRIES` when there are no more files or directories left to be examined. Unfortunately, like `ACTION_EXAMINE`, this packet has its own peculiarities. Among the quirks that `ACTION_EXAMINE_NEXT` must account for are:

- The situation where an application calls `ACTION_EXAMINE_NEXT` one or more times and then stops invoking it before encountering the end of the directory.

- The situation where a `FileInfoBlock` passed to `ACTION_EXAMINE_NEXT` is not the same as the one passed to `ACTION_EXAMINE` or even the previous `EXAMINE_NEXT` operation. Instead, it is a copy of the `FileInfoBlock` with only the `fib_DiskKey` and the first 30 bytes of the `fib_FileName` fields copied over. *This is now considered to be illegal and will not work in the future. Any new code should not be written in this manner.*

- Because a handler can receive other packet types between `ACTION_EXAMINE_NEXT` operations, the `ACTION_EXAMINE_NEXT` function must handle any special cases that may result.

- The `LOCK` passed to `ACTION_EXAMINE_NEXT` is not always the same lock used in previous operations. It is however a lock on the same object.

Because of these problems, `ACTION_EXAMINE_NEXT` is probably the trickiest action to write in any handler. Failure to handle any of the above cases can be quite disastrous.

```
ACTION_CREATE_DIR      22      CreateDir(...)
ARG1:  LOCK            Lock to which ARG2 is relative
ARG2:  BSTR            Name of new directory (relative to ARG1)

RES1:  LOCK            Lock on new directory
RES2:  CODE            Failure code if RES1 = DOSFALSE
```

```
ACTION_DELETE_OBJECT   16      DeleteFile(...)
ARG1:  LOCK            Lock to which ARG2 is relative
ARG2:  BSTR            Name of object to delete (relative to ARG1)

RES1:  BOOL            Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE            Failure code if RES1 = DOSFALSE
```

```
ACTION_RENAME_OBJECT   17      Rename(...)
ARG1:  LOCK            Lock to which ARG2 is relative
ARG2:  BSTR            Name of object to rename (relative to ARG1)
ARG3:  LOCK            Lock associated with target directory
ARG4:  BSTR            Requested new name for the object

RES1:  BOOL            Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE            Failure code if RES1 = DOSFALSE
```

```
ACTION_SET_PROTECT     21      SetProtection(...)
ARG1:  Unused
ARG2:  LOCK            Lock to which ARG3 is relative
ARG3:  BSTR            Name of object (relative to ARG2)
ARG4:  LONG            Mask of new protection bits
```

```
RES1:  BOOL            Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE            Failure code if RES1 = DOSFALSE
```

This action allows an application to modify the protection bits of an object. The 4 lowest order bits (RWED) are a bit peculiar. If their respective bit is set, that operation is not allowed (i.e. if a file's delete bit is set the file is *not* deleteable). By default, files are created with the RWED bits set and all others cleared. Additionally, any action which modifies a file is required to clear the A (archive) bit. See the *dos/dos.h* include file for the definitions of the bit fields.

```
ACTION_SET_COMMENT     28      SetComment(...)
ARG1:  Unused
ARG2:  LOCK            Lock to which ARG3 is relative
ARG3:  BSTR            Name of object (relative to ARG2)
ARG4:  BSTR            New Comment string
```

```
RES1:  BOOL            Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE            Failure code if RES1 = DOSFALSE
```

This action allows an application to set the comment string of an object. If the object does not exist then `DOSFALSE` will be returned in `RES1` with the failure code in `RES2`. The comment string is limited to 79 characters.

```
ACTION_SET_DATE        34      SetFileDate(...) in 2.0
ARG1:  Unused
ARG2:  LOCK            Lock to which ARG3 is relative
ARG3:  BSTR            Name of Object (relative to ARG2)
ARG4:  CPTR            DateStamp

RES1:  BOOL            Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE            Failure code if RES1 = DOSFALSE
```

This action allows an application to set an object's creation date.

2.0 only

```
ACTION_FH_FROM_LOCK    1026    OpenFromLock(lock)
ARG1:  BPTR            BPTR to file handle to fill in
ARG2:  LOCK            Lock of file to open
```

```
RES1:  BOOL            Success/failure (DOSTRUE/DOSFALSE)
RES2:  CODE            Failure code if RES1 = NULL
```

This action open a file from a given lock. If this action is successful, the file system will essentially steal the lock so a program should not use it anymore. If `ACTION_FH_FROM_LOCK` fails, the lock is still usable by an application.

2.0 only

```
ACTION_SAME_LOCK       40      SameLock(lock1,lock2)
ARG1:  BPTR            Lock 1 to compare
ARG2:  BPTR            Lock 2 to compare
```

```
RES1:  LONG            Result of comparison, one of
DOSTRUE                if locks are for the same object
DOSFALSE                if locks are on different objects
RES2:  CODE            Failure code if RES1 is LOCK_DIFFERENT
```

This action compares the targets of two locks. If they point to the same object,

ACTION\_SAME\_LOCK should return LOCK\_SAME.

2.0 only

```
ACTION_MAKE_LINK          1021    MakeLink(name,targ,mode)
ARG1:  BPTR      Lock on directory ARG2 is relative to
ARG2:  BSTR      Name of the link to be created (relative to ARG1)
ARG3:  BPTR      Lock on target object or name (for soft links).
ARG4:  LONG      Mode of link, either LINK_SOFT or LINK_HARD

RES1:   BOOL      Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE      Failure code if RES1 is DOSFALSE
```

This packet causes the file system to create a link to an already existing file or directory. There are two kinds of links, hard links and soft links. The basic difference between them is that a file system resolves a hard link itself, while the file system passes a string back to DOS telling it where to find a soft linked file or directory. To the packet level programmer, there is essentially no difference between referencing a file by its original name or by its hard link name. In the case of a hard link, ARG3 is a lock on the file or directory that the link is ``linked'' to, while in a soft link, ARG3 is a pointer (CPTR) to a C-style string.

In an over-simplified model of the ROM file system, when asked to locate a file, the system scans a disk looking for a file header with a specific (file) name. That file header points to the actual file data somewhere on the disk. With hard links, more than one file header can point to the same file data, so data can be referenced by more than one name. When the user tries to delete a hard link to a file, the system first checks to see if there are any other hard links to the file. If there are, only the hard link is deleted, the actual file data the hard link used to reference remains, so the existing hard links can still use it. In the case where the original link (not a hard or soft link) to a file is deleted, the file system will make one of its hard links the new ``real'' link to the file. Hard links can exist on directories as well. Because hard links ``link'' directly to the underlying media, hard links in one file system cannot reference objects in another file system.

Soft links are resolved through DOS calls. When the file system scans a disk for a file or directory name and finds that the name is a soft link, it returns an error code (ERROR\_IS\_SOFT\_LINK). If this happens, the application must ask the file system to tell it what the link the link refers to by calling ACTION\_READ\_LINK. Soft Links are stored on the media, but instead of pointing directly to data on the disk, a soft link contains a path to its object. This path can be relative to the lock in ARG1, relative to the volume (where the string will be prepended by a colon ':'), or an absolute path. An absolute path contains the name of another volume, so a soft link can reference files and directories on other disks.

2.0 only

```
ACTION_READ_LINK          1024    ReadLink(port,lck,nam,buf,len)
ARG1:  BPTR      Lock on directory that ARG2 is relative to
ARG2:  CPTR      Path and name of link (relative to ARG1). NOTE: This is a C
string not a BSTR
ARG3:  APTR      Buffer for new path string
ARG4:  LONG      Size of buffer in bytes

RES1:   LONG      Actual length of returned string, -2 if there isn't enough
space in buffer, or -1 for other errors
RES2:   CODE      Failure code
```

2.0 only

This action reads a link and returns a path name to the link's object. The link's name (plus any necessary path) is passed as a CPTR (ARG2) which points to a C-style string, *not a BSTR*. ACTION\_READ\_LINK returns the path name in ARG3. The length of the target string is returned in RES1 (or a -1 indicating an error).

```
ACTION_CHANGE_MODE        1028    ChangeMode(type,obj,mode)
ARG1:  LONG      Type of object to change - either CHANGE_FH or CHANGE_LOCK
ARG2:  BPTR      object to be changed
ARG3:  LONG      New mode for object - see ACTION_FINDINPUT, and
ACTION_LOCATE_OBJECT
```

```
RES1:   BOOL      Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE      Failure code if RES1 is DOSFALSE
```

2.0 only

This action requests that the handler change the mode of the given file handle or lock to the mode in ARG3. This request should fail if the handler can't change the mode as requested (for example an exclusive request for an object that has multiple users).

```
ACTION_COPY_DIR_FH        1030    DupLockFromFH(fh)
ARG1:  LONG      fh_Arg1 of file handle
```

```
RES1:   BPTR      Lock associated with file handle or NULL
RES2:   CODE      Failure code if RES1 = NULL
```

2.0 only

This action requests that the handler return a lock associated with the currently opened file handle. The request may fail for any restriction imposed by the file system (for example when the file handle is not opened in a shared mode). The file handle is still usable after this call, unlike the lock in ACTION\_FH\_FROM\_LOCK.

```
ACTION_PARENT_FH          1031    ParentOfFH(fh)
ARG1:  LONG      fh_Arg1 of File handle to get parent of
```

```
RES1:   BPTR      Lock on parent of a file handle
RES2:   CODE      Failure code if RES1 = NULL
```

This action obtains a lock on the parent directory (or root of the volume if at the top level) for a currently opened file handle. The lock is returned as a shared lock and must be freed. Note that unlike ACTION\_COPY\_DIR\_FH, the mode of the file handle is unimportant. For an open file, ACTION\_PARENT\_FH should return a lock under all circumstances.

2.0 only

```
ACTION_EXAMINE_ALL        1033    ExAll(lock,buff,size,type,ctl)
ARG1:  BPTR      Lock on directory to examine
ARG2:  APTR      Buffer to store results
ARG3:  LONG      Length (in bytes) of buffer (ARG2)
ARG4:  LONG      Type of request - one of the following:
```

```
ED_NAME Return only file names
ED_TYPE Return above plus file type
ED_SIZE Return above plus file size
ED_PROTECTION Return above plus file protection
ED_DATE Return above plus 3 longwords of date
ED_COMMENT Return above plus comment or NULL
```

```
ARG5:  BPTR      Control structure to store state information. The control
structure must be allocated with AllocDosObject()!
```

```
RES1:   LONG      Continuation flag - DOSFALSE indicates termination
RES2:   CODE      Failure code if RES1 is DOSFALSE
```

This action allows an application to obtain information on multiple directory entries. It is particularly useful for applications that need to obtain information on a large number of files and directories.

This action fills the buffer (ARG2) with partial or whole ExAllData structures. The size of the ExAllData structure depends on the type of request. If the request type field (ARG4) is set to ED\_NAME, only the ed\_Name field is filled in. Instead of copying the unused fields of the ExAllData structure into the buffer, ACTION\_EXAMINE\_ALL truncates the unused fields. This effect is cumulative, so requests to fill in other fields in the ExAllData structure causes all fields that appear in the structure *before* the requested field will be filled in as well. Like the ED\_NAME case mentioned above, any field that appears after the requested field will be truncated (see the ExAllData structure below). For example, if the request field is set to ED\_COMMENT, ACTION\_EXAMINE\_ALL fills in all the fields of the ExAllData structure, because the ed\_Comment field is last. This is the only case where the packet returns entire ExAllData structures.

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

Each ExAllData structure entry has an ead\_Next field which points to the next ExAllData structure. Using these links, a program can easily chain through the ExAllData structures without having to worry about how large the structure is. *Do not examine the fields beyond those requested* as they certainly will not be initialized (and will probably overlay the next entry).

The most important part of this action is the ExAllControl structure. It *must* be allocated and freed through AllocDosObject()/FreeDosObject(). This allows the structure to grow if necessary with future revisions of the operating and file systems. Currently, ExAllControl contains four fields:

**Entries** - This field is maintained by the file system and indicates the actual number of entries present in the buffer after the action is complete. Note that a value of zero is possible here as no entries may match the match string.

**LastKey** - This field *must* be initialized to 0 by the calling application before using this packet for the first time. This field is maintained by the file system as a state indicator of the current place in the list of entries to be examined. The file system may test this field to determine if this is the first or a subsequent call to this action.

**MatchString** - This field points to a pattern matching string parsed by ParsePattern() or ParsePatternNoCase(). The string controls which directory entries are returned. If

this field is NULL, then all entries are returned. Otherwise, this string is used to pattern match the names of all directory entries before putting them into the buffer. The default AmigaDOS pattern match routine is used unless MatchFunc is not NULL (see below). Note that it is not acceptable for the application to change this field between subsequent calls to this action for the same directory.

**MatchFunc** - This field contains a pointer to an alternate pattern matching routine to validate entries. If it is NULL then the standard AmigaDOS wild card routines will be used. Otherwise, MatchFunc points to a hook function that is called in the following manner:

```
2.0 only
BOOL = MatchFunc(hookptr, data, typeptr)
                        A0      A1      A2
hookptr  Pointer to hook being called
data     Pointer to (partially) filled in ExAllData for item being checked.
typeptr  Pointer to longword indicating the type of the ExAll request (ARG4).
```

This function is expected to return DOSTRUE if the entry is accepted and DOSFALSE if it is to be discarded.

```
2.0 only
ACTION_EXAMINE_FH      1034      ExamineFH(fh, fib)
ARG1:  BPTR      File handle on open file
ARG2:  BPTR      FileInfoBlock to fill in

RES1:  BOOL      Success/Failure (DOSTRUE/DOSFALSE)
RES2:  CODE      Failure code if RES1 is DOSFALSE
```

This function examines a file handle and fills in the FileInfoBlock (found in ARG2) with information about the current state of the file. This routine is analogous to the ACTION\_EXAMINE\_OBJECT action for locks. Because it is not always possible to provide an accurate file size (for example when buffers have not been flushed or two processes are writing to a file), the fib\_Size field (see *dos/dos.h*) may be inaccurate.

```
ACTION_ADD_NOTIFY      4097      StartNotify(NotifyRequest)
ARG1:  BPTR      NotifyRequest structure

RES1:  BOOL      Success/Failure (DOSTRUE/DOSFALSE)
RES2:  CODE      Failure code if RES1 is DOSFALSE
```

This action asks a file system to notify the calling program if a particular file is altered. A file system notifies a program either by sending a message or by signaling a task.

```
struct NotifyRequest {
    UBYTE *nr_Name;
    UBYTE *nr_FullName;          /* set by dos - don't touch */
    ULONG nr_UserData;          /* for applications use */
    ULONG nr_Flags;

    union {

        struct {
            struct MsgPort *nr_Port;          /* for SEND_MESSAGE */
        } nr_Msg;

        struct {
            struct Task *nr_Task;             /* for SEND_SIGNAL */
        } nr_Task;
    };
};
```



```
        UBYTE nr_SignalNum;          /* for SEND_SIGNAL */
        UBYTE nr_pad[3];
    } nr_Signal;
} nr_stuff;

ULONG nr_Reserved[4];               /* leave 0 for now */

/* internal use by handlers */
ULONG nr_MsgCount;                  /* # of outstanding msgs */
struct MsgPort *nr_Handler;         /* handler sent to (for EndNotify)

*/
};
```

To use this packet, an application needs to allocate and initialize a `NotifyRequest` structure (see above). As of this writing, `NotifyRequest` structures are not allocated by `AllocDosObject()`, but this may change in the future. The handler gets the watched file's name from the `nr_FullName` field. The current file system does not currently support wild cards in this field, although there is nothing to prevent other handlers from doing so.

The string in `nr_FullName` must be an absolute path, including the name of the root volume (no assigns). The absolute path is necessary because the file or its parent directories do not have to exist when the notification is set up. This allows notification on files in directories that do not yet exist. Notification will not occur until the directories and file are created.

An application that uses the `StartNotify()` DOS call does *not* fill in the `NotifyRequest`'s `nr_FullName` field, but instead fills in the `nr_Name` field. `StartNotify()` takes the name from the `nr_Name` field and uses `GetDeviceProc()` and `NameFromLock()` to expand any assigns (such as `ENV:`), storing the result in `nr_FullName`. Any application utilizing the packet level interface instead of `StartNotify()` must expand their own assigns. Handlers must not count on `nr_Name` being correct.

The notification type depends on which bit is set in the `NotifyRequest.nr_Flags` field. If the `NRF_SEND_MESSAGE` bit is set, an application receives notification of changes to the file through a message (see `NotifyMessage` from *dos/notify.h*). In this case, the `nr_Port` field must point to the message port that will receive the notifying message. If the `nr_Flags NRF_SEND_SIGNAL` bit is set, the file system will signal a task instead of sending a message. In this case, `nr_Task` points to the task and `nr_SignalNum` is the signal number. *Only one of these two bits should be set!*

When an application wants to limit the number of `NotifyMessages` an handler can send per `NotifyRequest`, the application sets the `NRF_WAIT_REPLY` bit in the `nr_Flags` field. This bit tells the handler not to send new `NotifyMessages` to a `NotifyRequest`'s message port if the application has not returned a previous `NotifyMessage`. This pertains only to a specific `NotifyRequest`--if other `NotifyRequests` exist on the same file (or directory) the handler will still send `NotifyMessages` to the other `NotifyRequest`'s message ports. The `NRF_WAIT_REPLY` bit only applies to message notification.

If an application needs to know if a file or directory exists at the time the application sets up notification on that file or directory, the application can set the `NRF_NOTIFY_INITIAL` bit in the `nr_Flags` field. If the file or directory exists, the handler sends an initial message or gives an initial signal.

## Volume Manipulation/Information

The Volume Manipulation and Information actions are used to allow access to the underlying volume currently being manipulated by the file system.

```
ACTION_CURRENT_VOLUME      7      <sendpkt only>
RES1:  BPTR      Pointer to volume node of current volume
```

This action returns a pointer to the volume node (from the DOS device list) associated with the file system. As the volume node may be removed from the device list when the file system mounts a different volume (such as when directed to by an `ACTION_INHIBIT`) there is no guarantee that this pointer will remain valid for any amount of time. This action is generally used by AmigaDOS to provide the volume line of a requester.

```
ACTION_DISK_INFO           25      Info(...)
ARG1:  BPTR      Pointer to an InfoData structure to fill in

RES1:  BOOL      Success/Failure (DOSTRUE/DOSFALSE)
```

```
ACTION_INFO                 26      <sendpkt only>
ARG1:  LOCK      Lock
ARG2:  BPTR      Pointer to a InfoData Structure to fill in

RES1:  BOOL      Success/Failure (DOSTRUE/DOSFALSE)
```

These actions are used to get information about the device and status of the file handler. `ACTION_DISK_INFO` is used by the *info* command to report the status of the volume currently in the drive. It fills in an `InfoData` structure about the volume the file system currently controls. `ACTION_INFO` fills in an `InfoData` structure for the volume the lock (ARG1) is on instead of the volume currently in the drive. These actions are generally expected to return `DOSTRUE`.

The `ACTION_DISK_INFO` packet has a special meaning for console style handlers. When presented with this packet, a console style handler should return a pointer to the window associated with the open handle.

```
ACTION_RENAME_DISK          9      Relabel(...) in 2.0
ARG1:  BSTR      New disk name

RES1:  BOOL      Success/Failure (DOSTRUE/DOSFALSE)
```

This action allows an application to change the name of the current volume. A file system implementing this function must also change the name stored in the volume node of the DOS device list.



2.0 only

```
ACTION_FORMAT          1020      Format(fs,vol,type)
ARG1:  _ BSTR           Name of device (with trailing ':')
ARG2:  _ BSTR           Name for volume (if supported)
ARG3:  _ LONG           Type of format (file system specific)

RES1:  _ BOOL           Success/Failure (DOSTRUE/DOSFALSE)
RES2:  _ CODE           Failure code if RES1 is DOSFALSE
```

This packet tells a file system to perform any device or file system specific formatting on any newly initialized media. Upon receiving this action, a file system can assume that the media has already been low level formatted and should proceed to write out any high level disk structure necessary to create an empty volume.

## Handler Maintenance and Control

A number of packets are defined to give an application some control over a file system:

```
ACTION_DIE              5          <sendpkt only>
RES1:  _ BOOL            DOSTRUE
```

As its name implies, the ACTION\_DIE packet tells a handler to quit. All new handlers are expected to implement this packet. Because of outstanding locks and the fact that the handler address is returned by the DeviceProc() routine, it is unlikely that the handler can disappear completely, but instead will have to release as many resources as possible and simply return an error on all packets sent to it.

In the future, the system may be able to determine if there are any outstanding DeviceProc() references to a handler, and therefore make it is safe to shut down completely.

```
ACTION_FLUSH            27          <sendpkt only>
RES1:  _ BOOL            DOSTRUE
```

This action causes the file system to flush out all buffers to disk *before* returning this packet. If any writes are pending, they must be processed before responding to this packet. This packet allows an application to make sure that the data that is supposed to be on the disk is actually written to the disk instead of waiting in a buffer.

```
ACTION_MORE_CACHE       18          AddBuffers(...) in 2.0
ARG1:  _ LONG            Number of buffers

RES1:  _ BOOL            DOSTRUE
RES2:  _ LONG            New number of buffers
```

This action allows an application to change the number of internal buffers used by the file system for caching. Note that a positive number increases the number of buffers while a negative number decreases the number of buffers. In all cases, the number of current buffers are returned in RES2. This allows an application to inquire the number of buffers by sending in a value of 0 (resulting in no change). Note that the OFS and FFS in 1.3 do not accept a negative number of buffers.

```
ACTION_INHIBIT          31          Inhibit(...) in 2.0
ARG1:  _ BOOL            DOSTRUE = inhibit,      DOSFALSE = uninhibit

RES1:  _ BOOL            Success/failure (DOSTRUE/DOSFALSE)
```

This action is probably one of the most dangerous that a file system has to handle. When inhibited (ARG1 = DOSTRUE), the file system must not access any underlying media and return an error code on all attempts to access the device. Once uninhibited (ARG1 = DOSFALSE), the file system must *assume* that the medium has been changed. The file system must flush the buffers before the ACTION\_INHIBIT , popping up a requester demanding that the user put back the current disk, if necessary. The handler

may choose to reject an inhibit request if any objects are open for writing.

Although it's not required, a handler should nest inhibits. Prior to 2.0, the system handlers did not keep a nesting count and were subject to some obscure race conditions. The 2.0 ROM filing system introduced a nesting count.

```
ACTION_WRITE_PROTECT      1023    <sendpkt only>
ARG1:  BOOL      DOSTRUE/DOSFALSE (write protect/un-write protect)
ARG2:   LONG      32 Bit pass key

RES1:   BOOL      DOSTRUE/DOSFALSE
```

This is a new packet defined for the Fast File System. This packet allows an application to change the write protect flag of a disk (if possible - applications cannot write to floppies that have their write-protect tabs set). This packet is primarily intended to allow write-protecting non-removable media such as hard disks. The value in ARG1 toggles the write status. The 32-bit passkey allows a program to prevent other programs from unwrite-protecting a disk. To unlock a disk, ARG2 must match the passkey of the packet that locked the disk, unless the disk was locked with a passkey of 0. In this case, no passkey is necessary to unlock the disk.

2.0 only

```
ACTION_IS_FILESYSTEM      1027      IsFileSystem(devname)

RES1:   BOOL      Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE      Failure code if RES1 is DOSFALSE
```

Through this function, a handler can indicates whether or not it is a file system (whether or not it can support separate files for storing information). Programs will assume a handler can create multiple, distinct files through calls to Open() if the handler returns this packet with a DOSTRUE value. A handler does *not* need to support directories and subdirectories in order to qualify as a file system. It does have to support the Examine()/ExNext() calls.

Note that the AmigaDOS routine IsFileSystem() will attempt to use Lock(":",SHARED\_ACCESS) if this packet returns ERROR\_ACTION\_NOT\_KNOWN.

## Handler Internal

There are several actions that are generally used by handlers to allow messages returning from requested services (typically an Exec device) to look like incoming request packets. This allows the handler to request an asynchronous operation but be notified of the completion. For example, a handler sends the *serial.device* a request for a read, but instead of sending a plain IO request, it sends a DOS packet disguised as an IO request. The *serial.device* treats the packet like a normal IO request, returning it to the handler when it is finished. When the handler gets back its disguised DOS packet, it knows that the read has completed.

```
ACTION_NIL                0          <internal>
```

Although not specifically an action, many returns look like this value because the action field has not been filled in.

```
ACTION_READ_RETURN        1001      <internal>
```

Generally used to indicate the completion of an asynchronous read request.

```
ACTION_WRITE_RETURN       1002      <internal>
```

Generally used to indicate the completion of an asynchronous write request.

```
ACTION_TIMER              30        <internal>
```

Used to indicate the passage of a time interval. Many handlers have a steady stream of ACTION\_TIMER packets so that they can schedule house keeping and flush buffers when no activity has occurred for a given time interval.

## Obsolete Packets

There are several packet types that are documented within the system include files that are obsolete. A file system is not expected to handle these packets and any program which sends these packets can not expect them to work:

<code>ACTION_DISK_CHANGE</code>	33	<Obsolete>
<code>ACTION_DISK_TYPE</code>	32	<Obsolete>
<code>ACTION_EVENT</code>	6	<Obsolete>
<code>ACTION_GET_BLOCK</code>	2	<Obsolete>
<code>ACTION_SET_MAP</code>	4	<Obsolete>

Of particular note here is `ACTION_DISK_CHANGE`. The *DiskChange* command uses the `ACTION_INHIBIT` packet to accomplish its task.

## Console Only Packets

The remaining packets are only used for console handlers and do not need to be implemented by a file system.

<code>ACTION_SCREEN_MODE</code>	994	<sendpkt only>
<code>ARG1:</code>	<code>LONG</code>	Mode (zero or one)
<code>RES1:</code>	<code>BOOL</code>	Success/Failure (DOSTRUE/DOSFALSE)
<code>RES2:</code>	<code>CODE</code>	Failure code if RES1 is DOSFALSE

Switch the console to and from RAW mode. An ARG1 of one indicates the unprocessed, raw mode while an ARG1 of zero indicates the processed, ``cooked'' mode.

<code>ACTION_WAIT_CHAR</code>	20	<code>WaitForChar()</code>
<code>ARG1:</code>	<code>ULONG</code>	Timeout in microseconds
<code>RES1:</code>	<code>BOOL</code>	Success/Failure (DOSTRUE/DOSFALSE)
<code>RES2:</code>	<code>CODE</code>	Failure code if RES1 is DOSFALSE

Performs a timed read of a character. The `WaitForChar()` function uses this packet.

## Summary of Defined Packet Numbers

This is a listing of all the DOS packets defined by Commodore. Packets 0-1999 are reserved for use by Commodore. Unless otherwise noted, packets 2050-2999 are reserved for use by third party developers (see chart below). The remaining packets are reserved for future expansion (Note: packets 2008, 2009, 4097, and 4098 are in use by Commodore).

Decimal	Hex	Action #define
0	0x0000	<code>ACTION_NIL</code>
1		<Reserved by Commodore>
2	0x0002	<code>ACTION_GET_BLOCK</code>
3		<Reserved by Commodore>
4	0x0004	<code>ACTION_SET_MAP</code>
5	0x0005	<code>ACTION_DIE</code>
6	0x0006	<code>ACTION_EVENT</code>
7	0x0007	<code>ACTION_CURRENT_VOLUME</code>
8	0x0008	<code>ACTION_LOCATE_OBJECT</code>
9	0x0009	<code>ACTION_RENAME_DISK</code>
10-14		<Reserved by Commodore>
15	0x000F	<code>ACTION_FREE_LOCK</code>
16	0x0010	<code>ACTION_DELETE_OBJECT</code>
17	0x0011	<code>ACTION_RENAME_OBJECT</code>
18	0x0012	<code>ACTION_MORE_CACHE</code>
19	0x0013	<code>ACTION_COPY_DIR</code>
20	0x0014	<code>ACTION_WAIT_CHAR</code>
21	0x0015	<code>ACTION_SET_PROTECT</code>
22	0x0016	<code>ACTION_CREATE_DIR</code>
23	0x0017	<code>ACTION_EXAMINE_OBJECT</code>
24	0x0018	<code>ACTION_EXAMINE_NEXT</code>
25	0x0019	<code>ACTION_DISK_INFO</code>
26	0x001A	<code>ACTION_INFO</code>
27	0x001B	<code>ACTION_FLUSH</code>
28	0x001C	<code>ACTION_SET_COMMENT</code>
29	0x001D	<code>ACTION_PARENT</code>
30	0x001E	<code>ACTION_TIMER</code>
31	0x001F	<code>ACTION_INHIBIT</code>
32	0x0020	<code>ACTION_DISK_TYPE</code>
33	0x0021	<code>ACTION_DISK_CHANGE</code>
34	0x0022	<code>ACTION_SET_DATE</code>
35-39		<Reserved by Commodore>
40	0x0028	<code>ACTION_SAME_LOCK</code>
41-81		<Reserved by Commodore>
82	0x0052	<code>ACTION_READ</code>
83-86		<Reserved by Commodore>
87	0x0057	<code>ACTION_WRITE</code>
88-993		<Reserved by Commodore>
994	0x03E2	<code>ACTION_SCREEN_MODE</code>

995-1000		<Reserved by Commodore>
1001	0x03E9	ACTION_READ_RETURN
1002	0x03EA	ACTION_WRITE_RETURN
1003		<Reserved by Commodore>
1004	0x03EC	ACTION_FINDUPDATE
1005	0x03ED	ACTION_FINDINPUT
1006	0x03EE	ACTION_FINDOUTPUT
1007	0x03EF	ACTION_END
1008	0x03F0	ACTION_SEEK
1009-1019		<Reserved by Commodore>
1020	0x03FC	ACTION_FORMAT
1021	0x03FD	ACTION_MAKE_LINK
1022	0x03FE	ACTION_SET_FILE_SIZE
1023	0x03FF	ACTION_WRITE_PROTECT
1024	0x0400	ACTION_READ_LINK
1025		<Reserved by Commodore>
1026	0x0402	ACTION_FH_FROM_LOCK
1027	0x0403	ACTION_IS_FILESYSTEM
1028	0x0404	ACTION_CHANGE_MODE
1029		<Reserved by Commodore>
1030	0x0406	ACTION_COPY_DIR_FH
1031	0x0407	ACTION_PARENT_FH
1032		<Reserved by Commodore>
1033	0x0409	ACTION_EXAMINE_ALL
1034	0x040A	ACTION_EXAMINE_FH
1035-2007		<Reserved by Commodore>
2008	0x07D8	ACTION_LOCK_RECORD
2009	0x07D9	ACTION_FREE_RECORD
2010-2049		<Reserved by Commodore>
2050-2999		<Reserved for 3rd Party Handlers>
4097	0x1001	ACTION_ADD_NOTIFY
4098	0x1002	ACTION_REMOVE_NOTIFY
4099-		<Reserved by Commodore for Future Expansion>







