```
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
```

These three actions perform most of the work behind the AmigaDOS commands *MakeDir*, *Delete*, and *Rename* (for single files). These packets take as their parameters a lock describing where the file is and a name relative to that lock. It is the responsibility of the file system to ensure that the operation is not going to cause adverse effects. In particular, the RENAME_OBJECT action allows moving files across directory bounds and as such must ensure that it doesn't create hidden directory loops by renaming a directory into a child of itself.

For Directory objects, the DELETE_OBJECT action must ensure that the directory is empty before allowing the operation.

```
ACTION_PARENT            29      Parent(...)
ARG1:   LOCK    Lock on object to get the parent of

RES1:   LOCK    Parent Lock
RES2:   CODE    Failure code if RES1 = 0
```

This action receives a lock on an object and creates a shared lock on the object's parent. If the original object has no parent, then a lock of 0 is returned. Note that this operation is typically used in the process of constructing the absolute path name of a given object.

```
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
  LOCK_SAME          (0) if locks are for the same object
  LOCK_SAME_HANDLER  (1) if locks are on different objects of same handler
  LOCK_DIFFERENT    (-1) otherwise
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
```

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).

2.0 only

```
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
```

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).

2.0 only

```
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
```

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.

2.0 only

```
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 to control 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: