```
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
```

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 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:

```
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.

2.0 only

```
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 */
                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.

Handlers should only perform a notification when the actual contents of the file have changed. This includes `ACTION_WRITE`, `ACTION_SET_DATE`, `ACTION_DELETE`, `ACTION_RENAME_OBJECT`, `ACTION_FINDUPDATE`, `ACTION_FINDINPUT`, and `ACTION_FINDOUTPUT`. It may also include other actions such as `ACTION_SET_COMMENT` or `ACTION_SET_PROTECT`, but this is not required (and may not be expected by the application as there is no need to reread the data).

2.0 only

```
ACTION_REMOVE_NOTIFY     4098    EndNotify(NotifyRequest)
ARG1:   BPTR    Pointer to previously added notify request

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

This action cancels a notification (see `ACTION_ADD_NOTIFY`) . ARG1 is the NotifyRequest structure used to initiate the notification. The handler should abandon any pending notification messages. Note that it is possible for a file system to receive a reply from a previously sent notification message even after the notification has been terminated. It should accept these messages silently and throw them away.