Using the File Manager

Techniques for use

This section provides specific techniques for using the new features of the **File Manager**.

- You can pass **FSSpec** records that your application receives from the **Standard File Package** and the Finder directly to a set of high-level **File Manager** functions.
- You can exchange the contents of two files when updating a stored file, using the **FSpExchangeFiles** function.
- You can search a volume's catalog quickly, looking for files or directories that meet the criteria you specify, using the **PBCatSearch** function.
- You can track files within a volume by file number, using the set of functions that create and manipulate file ID references.
- You can mount a remote volume programmatically, without going through the Chooser, using the remote mounting functions.
- You can read and change privilege information in foreign file systems using the **PBGetForeignPrivs** and **PBSetForeignPrivs** functions.

Some of the new **File Manager** features depend on the system software; others depend on the characteristics of the volume. Before using any of the new **File Manager** features, check for availability by calling either the **Gestalt** function or the **PBHGetVolParms** function, depending on whether the feature's presence depends on the system software or the characteristics of the volume.

You can use **Gestalt** to determine whether or not you can call the functions that accept and support **FSSpec** records. Call **Gestalt** with the gestaltFSAttr selector to check for **File Manager** features. The response parameter has two relevant bits.

Constant	Meaning
gestaltFullExtFSDispatching	All of the routines selected through the _HFSDispatch trap macro are available to external file systems.
gestaltHasFSSpecCalls	The operating environment provides the file system specification versions of the basic file manipulation functions, plus the **FSMakeFSSpec** function.

For a complete description of the **Gestalt** function, see the section, **Compatibility Guidelines**.

To test for the availability of the features that depend on the volume, you call the low-level function **PBHGetVolParms**. **PBHGetVolParms** returns the volume description in an attributes buffer, defined in system software version 7.0 as the **GetVolParmsInfoBuffer** record.

Using FSSpec Records

The system software now recognizes the **file system specification (FSSpec) record,** which provides a simple, standard way to specify the name and location of a file or directory.

Your application typically receives **FSSpec** records from the **Standard File Package** or the Finder and passes those records on to the **File Manager**. For example, the following sample code fragments illustrate how your application might call the **Standard File Package** and then the **File Manager** when the user chooses **Open** from the **File** menu.

```
// Opening a document using the FSSpec record

// Assuming inclusion of MacHeaders


// prototype for error checking function
void myErrCheck(OSErr);

void myErrCheck (OSErr theErr)
{
    if (theErr) {
        // Handle error condition and exit program
    }
    return;
}

main()

{
    StandardFileReply mySFR;    // reply record
    SFTypeList myTypeList;  // list of types to display
    OSErr fsopenErr; // error returned by open function
    short fsRefNum;         // path reference number


    // Make sure to initialize the toolbox, if you don't, StandardGetFile will
    // not work properly
    InitGraf(&thePort);
    InitFonts();
    FlushEvents(everyEvent, 0);
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0L);
    InitCursor();

    // setup list of types
    myTypeList[0] = 'TEXT';
    myTypeList[1] = 'RAYS';

    StandardGetFile(nil, 2, myTypeList, &mySFR);
    if ( mySFR.sfGood ) {
        fsopenErr = FSpOpenDF(&mySFR.sfFile, fsCurPerm, &fsRefNum);
        myErrCheck(fsopenErr);      // check for errors
```

```
        // display document, or whatever else your application does
    }
    else {
        // if the user cancels the open, do whatever cleanup is necessary
    }
}
```

If you need to convert a file specification into an **FSSpec** record, call the new **FSMakeFSSpec** function. Do not fill in the fields of an **FSSpec** record yourself.

Three of the parameters to **FSMakeFSSpec** represent the volume, directory, and file specifications of the target object. You can provide this information in any of the four combinations described earlier in **File Specification Strategies** under the section entitled **Identifying Files, Directories, and Volumes**.

The following table details the ways your application can identify the name and location of a file or directory in a call to **FSMakeFSSpec** .

Table How **FSMakeFSSpec** interprets file specifications

vRefNum	dirID	fileName	Interpretation
Ignored	Ignored	Full pathname	Full pathname overrides any other information
Volume reference number or drive number	Directory ID	Partial pathname	Partial pathname starts in the directory whose parent is specified in the dirID parameter
Working directory reference number	Directory ID	Partial pathname	Directory specification in dirID overrides the directory implied by the working directory reference number Partial pathname starts in the directory whose parent is specified in dirID
Volume reference number or drive number	0	Partial pathname	Partial pathname starts in the root directory of the volume specified in vRefNum
Working directory	0	Partial pathname	Partial pathname

reference number			starts in the directory specified by the working directory reference number
Volume reference number or drive number	Directory ID	Empty string	The target object is the directory specified by the directory ID in dirID
Working directory reference number	0	Empty string	The target object is the directory specified by the working directory reference number in vRefNum
Volume reference number or drive number	0	Empty string	The target object is the root directory of the volume specified in vRefNum

The fourth parameter to **FSMakeFSSpec** is a pointer to the **FSSpec** record.

For a detailed description of **FSMakeFSSpec** , see **Making FSSpec Records** under **High-Level File Manager Routines** .

Updating Files

You can update an existing file simply and safely with the new **FSpExchangeFiles** function.

FSpExchangeFiles exchanges the contents of two files on a volume. You can use it to update a file without writing over the old version until the new version is stored safely. To update a file with **FSpExchangeFiles** , you first save a copy of the new data, and then call **FSpExchangeFiles** to put the new data in the original file. The file that you created to hold the new data now holds the original (obsolete) data. Updating files with the **FSpExchangeFiles** function preserves a file's Finder information and file ID.

FSpExchangeFiles does not move the data on the volume; it merely changes the information in the volume's catalog and, if the files are open, in the file control blocks (FCBs). The catalog entry for a file contains

- fields that describe the physical data, such as the first allocation block, physical end, and logical end of both the resource and data forks
- fields that describe the file within the file system, such as file number and parent directory ID

Fields that describe the data remain with the data; fields that describe the file remain with the file. The creation date remains with the file; the modification

date remains with the data.

The following Table illustrates the effects of **FSpExchangeFiles** on a set of sample catalog information.

Table The effect of **FSpExchangeFiles** on a catalog entry

Before FSpExchangeFiles			**After FSpExchangeFiles**	
First catalog entry	Second catalog entry	Catalog information	First catalog entry	Second catalog entry
		File description		
File A	File B	Filename	File A	File B
30	100	Parent directory	30	100
3000	100	File number	3000	100
Jan 1990	June 2000	Creation date	Jan 1990	June 2000
		Data description		
300	1000	First allocation block	1000	300
1024	7680	Physical end of data fork	7680	1024
998	7649	Logical end of data fork	7649	998
April 1990	July 2000	Last modification date	July 2000	April 1990

If one or both files are open, **FSpExchangeFiles** also updates the file control block, which describes the access path to data identified by a path reference number. Like the catalog entry, the file control block contains both physical information about the data and file system information about the file. The following table illustrates the effects of **FSpExchangeFiles** on the file control block.

Table The effect of **FSpExchangeFiles** on a file control block

Before FSpExchangeFiles			**After FSpExchangeFiles**	
Path reference number 4	Path reference number 10	FCB information	Path reference number 4	Path reference number 10
		File description		
File A	File B	Filename	File B	File A
30	100	Parent directory	100	30
		Data description		
300	1000	First allocation block	300	1000
1024	7680	Physical end-of-file	1024	7680
998	7649	Logical end-of-file	998	7649

The following code example illustrates the safe-save strategy using **FSpExchangeFiles** for an application that uses a memory-based editing system.

```
// Updating a file with FSpExchangeFiles

// Assuming inclusion of MacHeaders
#include <Folders.h>

// prototype routine like this prior to calling
```

```
void UpdateFile(void);

void UpdateFile()
{
    StandardFileReply reply;
    OSErr error;
    long seconds;
    Str255 tempFileName;
    short tempVRefNum;
    long tempDirID;
    FSSpec tempFSSpec;
    short fileRefNum;
    Ptr buffer;
    long count;

    // prototype for user defined routine to get data size and location
    void GetMyDataSizeLoc(long *, Ptr *);

    // NOTE: calling routine should make sure to initialize toolbox
    // or StandardPutFile may not work correctly

    StandardPutFile("\pSafe Save", "\pfilename", &reply);
    if ( reply.sfGood ) {
        // user saves file
        if ( reply.sfReplacing ) {
            GetDateTime(&seconds);

            // make up a temporary filename
            NumToString(seconds, tempFileName);

            // find the temporary folder;
            // create it if necessary
            error = FindFolder(reply.sfFile.vRefNum,
                    kTemporaryFolderType, kCreateFolder,
                    &tempVRefNum, &tempDirID);
            // make an FSSpec for the
            // temporary filename
            error = FSMakeFSSpec(tempVRefNum, tempDirID,
                        tempFileName, &tempFSSpec);
            // check for error

            // create a temporary file
            error = FSpCreate(&tempFSSpec, 'trsh', 'trsh',
                        reply.sfScript);
            // check for error

            // open the newly created file
            error = FSpOpenDF(&tempFSSpec, fsRdWrPerm, &fileRefNum);
            // check for error

            // get the data's size and location
            GetMyDataSizeLoc(&count, &buffer);

            // write to the file
            error = FSWrite(fileRefNum, &count, buffer);
            // check for error
```

```
            // close the temporary file
            error = FSClose(fileRefNum);
            // check for error

            // exchange the contents of the two files
            error = FSpExchangeFiles(&tempFSSpec, &reply.sfFile);
            // check for error

            // delete the temporary file
            error = FSpDelete(&tempFSSpec);
            // check for error
        } // if reply.sfReplacing
    } // reply.sfgood
}
```

Searching a Volume

You can search a volume's catalog efficiently using the new **PBCatSearch** function. **PBCatSearch** looks at all entries in all directories on the volume, and it returns a list of all files or directories that match the criteria you specify. You can ask **PBCatSearch** to match for names or partial names; file and directory attributes; Finder information; physical and logical file length; creation, modification, and backup dates; and parent directory ID.

Like all low-level **File Manager** functions, **PBCatSearch** exchanges information with your application through a parameter block. The **PBCatSearch** function uses a new parameter block variant, **CSParam**. For a description of all fields in the parameter block and how they are used, see the description of **PBCatSearch** .

PBCatSearch manipulates file and directory data in a catalog information **(CInfoPBRec**) record. (This record is also used by the **PBGetCatInfo** function.) You specify the limits of the search criteria to **PBCatSearch** in two catalog information records, called ioSearchInfo1 and ioSearchInfo2.

Some fields in the catalog information records apply only to files, some only to directories, and some to both. Some of the fields that apply to both have different names, depending on whether the target of the record is a file or a directory. **PBCatSearch** uses only some fields in the catalog information record. The first table below lists the fields used for files, and second lists the fields used for directories.

The fields in ioSearchInfo1 and ioSearchInfo2 have different uses:

- The name field in ioSearchInfo1 holds a pointer to the target string; the name field in ioSearchInfo2 must be NIL. (If you're not searching for the name, the name field in ioSearchInfo1 must also be NIL.)

- The date and length fields in ioSearchInfo1 hold the lowest values in the target range; the date and length fields in ioSearchInfo2 hold the highest values in the target range. **PBCatSearch** looks for values greater than or equal to the field values in ioSearchInfo1 and less than or equal to the values in ioSearchInfo2.

- The attributes and Finder information fields in ioSearchInfo1 hold the target values; the same fields in ioSearchInfo2 hold masks that specify which bits are relevant.

Fields in ioSearchInfo1 and ioSearchInfo2 used for a file

Field	Meaning
ioNamePtr	Filename
ioFlAttrib	File attributes
ioFlFndrInfo	Finder information (**FInfo** record, see **Finder Interface**)
ioFlLgLen	Data fork logical length
ioFlPyLen	Data fork physical length
ioFlRLgLen	Resource fork logical length
ioFlRPyLen	Resource fork physical length
ioFlCrDat	File creation date
ioFlMdDat	File modification date
ioFlBkDat	File backup date
ioFlXFndrInfo	Extended Finder information (**FXInfo** record, see **Finder Interface**)
ioFlParID	File's parent directory ID

Fields in ioSearchInfo1 and ioSearchInfo2 used for a directory

Field	Meaning
ioNamePtr	Directory name
ioFlAttrib	Directory attributes
ioDrUsrWds	Finder information (**DInfo** record, see **Finder Interface**)
ioDrNmFls	Number of files in the directory
ioDrCrDat	Directory creation date
ioDrMdDat	Directory modification date
ioDrBkDat	Directory backup date
ioDrFndrInfo	Extended Finder information (**DXInfo** record, **Finder Interface**)
ioDrParID	Directory's parent directory ID

In a pair of records that describe a file, for example, the variable fields have these meanings:

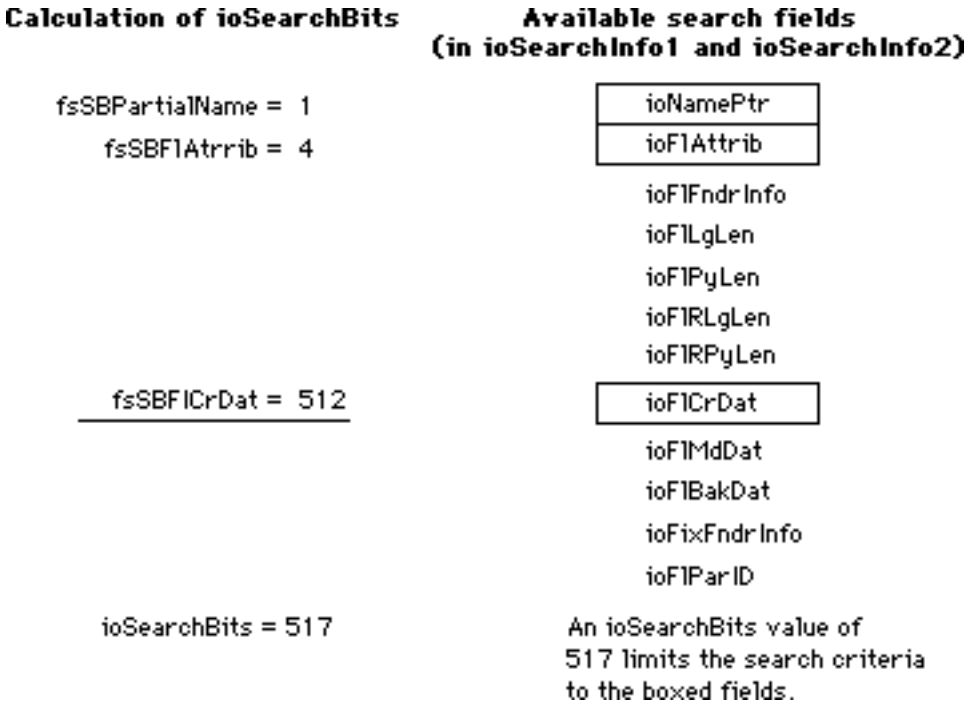
Field	ioSearchInfo1	ioSearchInfo2
ioNamePtr	Target string	Reserved (must be NIL)
ioFlAttrib	Desired attributes	Mask specifying which attributes are used in search
ioFlFndrInfo	Desired Finder profile	Mask specifying which Finder information is used in search
ioFlLgLen	Smallest desired size	Largest desired size
ioFlPyLen	Smallest desired size	Largest desired size
ioFlRLgLen	Smallest desired size	Largest desired size
ioFlCrDat	Earliest desired date	Latest desired date
ioFlMdDat	Earliest desired date	Latest desired date
ioFlBkDat	Earliest desired date	Latest desired date
ioFlXFndrInfo	Desired extended Finder profile	Mask specifying which extended Finder information is used in search

PBCatSearch searches only on bits 0 and 4 in the file attributes field (ioFlAttrib).

Attributes bit	Meaning
0	File or directory is locked
4	Entry is a directory

To fully describe the search criteria to **PBCatSearch**, you pass it a pair of catalog information records that determine the limits of the search and a mask that identifies the relevant fields within the catalog information records. You pass the mask in the ioSearchBits field in the **PBCatSearch** parameter block. To determine the value of ioSearchBits, add together the appropriate constants.

The following figure illustrates how the value in ioSearchBits determines which fields are used in ioSearchInfo1 and ioSearchInfo2.



The effect of ioSearchBits on interpretation of ioSearchInfo1 and ioSearchInfo2

A catalog entry must meet all of the specified criteria to be placed in the list of matches. After **PBCatSearch** has completed its scan of each entry, it checks the fsSBNegate bit. If that bit is set, **PBCatSearch** reverses the entry's match status (that is, if the entry is a match but the fsSBNegate bit is set, the entry is not put in the list of matches; if it is not a match, it is put in the list).

Although the use of **PBCatSearch** is significantly more efficient than searching the directories recursively, searching a large volume can take long enough to affect user response time. You can break a search into several shorter searches by specifying a maximum length of time in the ioSearchTime field and keeping an index in the ioCatPosition field. **PBCatSearch** stores its directory-location index in a catalog position record, **CatPositionRec** .

The following code example illustrates a code segment that uses **PBCatSearch** to find all files (not directories) whose names contain the string "Temp" and which were created within the past two days.

```
// Searching a volume with PBCatSearch

// Assuming inclusion of MacHeaders
```

```
// needed for prototype for strcpy, and printf
#include <stdio.h>
#include <string.h>

// function prototype
void SetupForFirstTime(void);

#define kMaxMatches 30              // find up to 30 matches in one pass
#define kOptBufferSize 16384     // use a 16k search cache for speed

// global variables
OSErr gErr;     // error variable
HParamBlockRec gPb; // parameter block for PBCatSearch
FSSpec gTheResults[kMaxMatches];     // put matches here
CInfoPBRec gSpec1;   // search criteria, part 1
CInfoPBRec gSpec2;   // search criteria, part 2
char gBuffer[kOptBufferSize];     // search cache
Str255 gFileName;    // name of string to look for

void  SetupForFirstTime()
{
    short vRefNum;    // volume on which to search
    long dirID;     // ignored dir ID for HGetVol
    long currentDateTime;     // current date in seconds
    long twoDaysAgo;    // date two days ago in seconds

    gErr = HGetVol(nil, &vRefNum, &dirID);   // search on default volume

    // search for temp
    // first use strcpy to copy string to gFileName, then convert to pascal style
    // string for toolbox use
    strcpy ((char *)gFileName, "Temp");
    CtoPstr(gFileName);

    GetDateTime(&currentDateTime); // get current time in seconds
    twoDaysAgo = currentDateTime - (2 * 24 * 60 * 60);

    gPb.csParam.ioCompletion = nil;  // no completion routine
    gPb.csParam.ioNamePtr = nil;    // no volume name;  use vRefNum
    gPb.csParam.ioVRefNum = vRefNum;  // volume to search
    gPb.csParam.ioMatchPtr = gTheResults;   // points to results buffer
    gPb.csParam.ioReqMatchCount = kMaxMatches;   // number of matches

    // search partial name, file attributs, and creation date
    gPb.csParam.ioSearchBits = fsSBPartialName + fsSBFlAttrib +
        fsSBFlCrDat;
    gPb.csParam.ioSearchInfo1 = &gSpec1;   // points to first criteria set
    gPb.csParam.ioSearchInfo2 = &gSpec2;   // points to second criteria set
    gPb.csParam.ioSearchTime = -1;            // don't time out on searches
    gPb.csParam.ioCatPosition.initialize = 0;   // set hint to 0
    gPb.csParam.ioOptBuffer = gBuffer;    // point to search cache
    gPb.csParam.ioOptBufSize  = kOptBufferSize;   // size of search cache

    gSpec1.hFileInfo.ioNamePtr = gFileName;   // point to string to find
    gSpec1.hFileInfo.ioFlAttrib = 0x00;    // clear bit 4 to ask for files
```

```
    gSpec1.hFileInfo.ioFlCrDat = twoDaysAgo;   // lower bound of creation date

    gSpec2.hFileInfo.ioNamePtr = nil;   // set to nil
    gSpec2.hFileInfo.ioFlAttrib  = 0x10;  // set mask for bit 4
    gSpec2.hFileInfo.ioFlCrDat = currentDateTime;// upper bound of creation
                                                 // date
}

main()
{
    short loopy;    // loop control variable
    short done;   // set to true when all matches done

    done = 0;  // be sure to initialize done
    SetupForFirstTime();          // initialize data records
    do {
        gErr = PBCatSearchSync(&gPb); // get some files
        done = (gErr == eofErr);      // eofErr returned when all done
        if (((gErr == noErr) || done) && (gPb.csParam.ioActMatchCount > 0) )
            for ( loopy = 0;  loopy < gPb.csParam.ioActMatchCount; loopy++) {
                // report all matches found
                printf ("name = %s\n",gTheResults[loopy].name);
            }
    } while (!done);
}
```

Tracking Files With File IDs

The **File Manager** provides a set of three low-level functions for creating, resolving, and deleting file ID references. These functions were developed for use by the **Alias Manager**, which uses them to track files that have been moved or renamed. In most cases, your application should track files with the **Alias Manager** , not with file IDs.

You establish a file ID reference when you need to identify a file by file number (see **File IDs**, under **New Special-Purpose Features in the File Manager**). You create a file ID reference with the **PBCreateFileIDRef** function. Because the **File Manager** assigns file numbers independently on each volume, a file ID is not guaranteed to be unique across volumes.

You can resolve a file ID reference by calling the **PBResolveFileIDRef** function, which determines the name and parent directory ID of the file with a given ID. With this information and a volume specification, you can uniquely identify any file in the file system.

If you no longer need a file ID, remove its record from the directory by calling the **PBDeleteFileIDRef** function. Removing a file ID is seldom appropriate, but the function is provided for completeness.

To preserve a file's ID when you are saving a new version of it, you should use the new safe-save strategy described earlier in **Updating Files**.

Mounting Volumes Programmatically

Your application can mount remote volumes, without requiring the user to go through the Chooser, using a set of three new functions: **PBGetVolMountInfoSize** , **PBGetVolMountInfo** , and **PBVolumeMount** .

Ordinarily, before you can mount a volume programmatically, you must record its mounting information while it is mounted. Because the size of the mounting information can vary, you first call the **PBGetVolMountInfoSize** function, which returns the size of the record you'll need to allocate to hold the mounting information. You then allocate the record and call **PBGetVolMountInfo** , passing a pointer to the record. When you want to mount the volume later, you can pass the record directly to the **PBVolumeMount** function.

> **Note:** The functions for mounting volumes programmatically are low-level functions designed for specialized applications. Even if your application needs to track and access volumes automatically, it can ordinarily use the **Alias Manager** . The **Alias Manager** can record mounting information and later remount most volumes, even those that do not support the programmatic mounting functions.

The programmatic mounting functions can now be used to mount AppleShare volumes. The functions have been designed so that they can eventually be used to mount local Macintosh volumes, such as partitions on devices that support partitioning, and local or remote volumes managed by non-Macintosh file systems.

The programmatic mounting functions use the ioParam variant of the **ParamBlockRec** record. They store the mounting information in a variable-sized structure called the **VolMountInfoHeader** record.

Manipulating Privilege Information in Foreign File Systems

Version 7.0 includes two low-level functions that support interaction with foreign file systems: **PBGetForeignPrivs** and **PBSetForeignPrivs** . These functions let you manipulate privilege information on a file system with a non-Macintosh privilege model.

The access-control functions were designed for use by shell programs, such as the Finder, that need to use the native privilege model of the foreign file system. Most applications can rely on using AFP functions, which are recognized by file systems that support the Macintosh privilege model. The new access-control functions do not relieve a foreign file system of the need to map its own privilege model onto the AFP functions.

Like all other low-level **File Manager** functions, the access-control functions exchange information with your application through parameter blocks. The meanings of some fields depend on what the foreign file system is. These fields are currently defined for A/UX, and you can define them for other file systems. If you are defining a new privilege model, register it with Macintosh Developer Technical Support.

You can identify the foreign file system through the **PBHGetVolParms** function. Version 7.0 defines a new attributes buffer for the **PBHGetVolParms** function, with a field for the foreign privilege model, vMForeignPrivID.

> **Note:** The value of vMForeignPrivID is unrelated to whether the remote volume supports the AFP access-control functions, described in the **File Manager** chapter of Volume V. You can determine whether the volume supports the AFP access-control functions by checking the bHasAccessCntl bit in the vMAttrib field.

A value of 0 for vMForeignPrivID signifies an HFS volume that supports no foreign privilege models. The field currently has one other defined value, fsUnixPriv.

For an updated list of supported models and their constants and fields, contact Macintosh Developer Technical Support.

A volume can support no more than one foreign privilege model.

The access-control functions store information in a new parameter block type, **ForeignPrivParam** .

The parameter block can store access-control information in one or both of

- a buffer of any length, whose location and size are stored in the parameter block
- four longwords of data stored in the parameter block itself

The meanings of the fields in the parameter block depend on the definitions established by the foreign file system.