Introducing the SMB Protocol

This document describes the SMB file sharing protocol. Systems can use these protocols to obtain or provide remote file services in a network environment. These protocols are designed to allow systems to transparently access files which reside on remote systems. Items which are mapped into the file space (such as UNIX style "device special files") are also transparently shared by these protocols.

When two machines first come into network contact they may negotiate the use of a higher level "Extension Protocol". For example, two MS-DOS machines would agree to use the MS-DOS-specific protocol extensions. These extensions can include both new messages as well as changes to the fields and semantics of existing messages. The "Core/Extension Protocol" definition allows a system to communicate at a strong, functional level with other "core" machines, and to communicate in full transparent detail to its "brother" systems. The ability to negotiate the protocol used across a given connection is also used, in those cases where multiple versions of a protocol exist, to ensure that only compatible versions of the protocol are used.

This document assumes the existence of, but does not describe, a lower level set of protocols that provide for virtual circuits and transport between clients and servers. Further, it does not discuss the mechanism used to "identify" and "locate" a correspondent in order to establish said virtual circuit.

The SMB Architectural Model

The Network File Access system described in this document deals with two types of systems on the network -- consumers and servers. A consumer is a system that requests network file services and a server is a system that delivers network file services. Consumers and servers are logical systems; a consumer and server may coexist in a single physical system.

Consumers are responsible for directing their requests to the appropriate server. The network addressing mechanism or naming convention through which the server is identified is outside the scope of this document.

Each server makes available to the network a self-contained file structure. There are no storage or service dependencies on any other servers. A file must be entirely contained by a single server.

The core file sharing protocol requires server authentication of users before file accesses are allowed. Each server processor authenticates its own users. A user must "login" to each server that it wishes to access.

This authentication model assumes that the LAN connects autonomous systems that are willing to make some subset of their local files available to remote users.

The following environments exist in the core file sharing protocol environment.

- Virtual Circuit Environment. This consists of one VC established between a consumer system and server system. Consumers may have only a single request active on any VC at any time -- i.e., a second request cannot be initiated until the response to the first has been received. A VC is formed using transport services.
- Logon Environment**.** This is represented by a Tree ID (TID). A TID uniquely identifies a file sharing connection between a consumer and server. It also identifies the scope and type of accesses allowed across the connection. With the exception of the Tree Connect and Negotiate commands, the TID field in a message must always contain a valid TID. There may be any number of file sharing connections per VC.
- Process Environment. This is represented by a process ID (PID). A PID uniquely identifies a consumer process within a given VC environment.
- File Environment. This is represented by a File Handle (FID). A FID identifies an open file and is unique within a given VC environment.

When one of these environments is terminated, all environments contained within it will be terminated. For example, if a VC is terminated all PIDs, TIDs and FIDs within it will be invalidated.

SMB Process Management

How and when servers create and destroy processes is, of course, an implementation issue and there is no requirement that this be tied in any way to the consumer's process management. However, it is necessary for the server to be aware of the consumer's process management activities as files are accessed on behalf of consumer processes. Therefore, the file sharing protocol includes appropriate notifications.

All messages, except Negotiate, include a process ID (PID) to indicate which user process initiated a request. Consumers inform servers of the creation of a new process by simply introducing a new PID into the dialogue. Process destruction must be explicitly indicated and the "Process Exit" command is provided for this purpose. The consumer must send a Process Exit command whenever a user process is destroyed. This enables the server to free any resources (for example, locks) reserved by that process as well as perform any local process management activities that its implementation might require.

The SMB Protocol Message Format

Every message has a common format. The following C-language style definition shows that format.

Data type	Field	Value
BYTE	smb_fid[4];	contains 0xFF, 'SMB'
BYTE	smb_com;	command code
BYTE	smb_rcls;	error code class
BYTE	smb_reh;	reserved (contains AH if DOS INT-24 ERR)
WORD	smb_err;	error code
BYTE	smb_res;	reserved
WORD	smb_res[7];	reserved
WORD	smb_tid;	tree id number
WORD	smb_pid;	caller's process id number
WORD	smb_uid;	user id number
WORD	smb_mid;	multiplex id number
BYTE	smb_wct;	count of parameter words
WORD	smb_vwv[];	variable number words of params
WORD	smb_bcc;	number of data bytes following
BYTE	smb_data[];	data bytes

The bytes within a word are ordered such that the low byte precedes the high byte.

File Sharing Connections: Overview

The networks using this file sharing protocol will contain not only multi-user systems with user based protection models, but single-user systems that have no concept of user-ids or permissions. Once these machines are connected to the network, however, they are in a multi-user environment and need a method of access control. First, unprotected machines need to be able to provide some sort of bona-fides to other net machines which do have permissions; secondly unprotected machines need to control access to their files by others.

This protocol defines a mechanism that enables the network software to provide the protection where it is missing from the operating system, and supports user-based protection where it is provided by the operating system. The mechanism also allows machines with no concept of user-id to demonstrate access authorization to machines which do have a permission mechanism. Finally, the permission protocol is designed so that it can be omitted if both machines share a common permission mechanism.

This protocol, called the "tree connection" protocol, does not specify a user interface. A possible user interface will be described by way of illustration.

Accessing Unprotected Servers: Examples

The following examples apply to access to serving systems which do not have a permission mechanism.

a) **NET SHARE**

By default (on unprotected machines) all network requests are refused as unauthorized. Should a user wish to allow access to some or all of his files he offers access to an arbitrary set of subtrees by specifying each subtree and a password.

Examples:

```
NET SHARE \dir1 "bonzo"
```

assigns password "bonzo" to all files within directory "dir1" and its subdirectories.

```
NET SHARE \  " "  RO
```

offers read-only access to everything (all files are within the root directory or its subdirectories).

```
NET SHARE \work "flipper" RW
```

Offers read-write access to all files within the \WORK directory and its subdirectories.

b) **NET USE**

Other users can gain access to one or more offered subtrees via the NET USE command. Once the NET USE command is issued the user can access the files freely without further special requirements.

Examples:

```
NET USE \\machine-name\dir1 "bonzo"
```

now any pathname starting with \\machine-name\dir1 is valid.

```
NET USE \\machine-name\
```

```
NET USE \\machine-name\work "flipper"
```

Now any read request to any file on that machine is valid. Read-write requests only succeed to files whose pathnames start with \\machine-name\work

The requester must remember the machine-name pathname prefix combination supplied with the NET USE request and associate it with the index value returned by the server. Subsequent requests using this index must include only the pathname relative to the connected subtree as the server treats the subtree as the root directory.

When the requester has a file access request for the server, it looks through its list of prefixes for that machine and selects the most specific (the longest) match. It then includes the index associated with this prefix in his request along with the remainder of the pathname.

Note that one always offers a directory and all files underneath that directory are then affected. If a particular file is within the range of multiple offer ranges, connecting to any of the offer ranges gains access to the file with the permissions specified for the offer named in the NET USE. The server will not check for nested directories with more restrictive permissions.

Accessing Protected Servers: Examples

Servers with user based file protection schemes will interpret the Tree Connect command slightly differently from systems with file oriented file protection schemes. They interpret the "name" parameter as a username rather than a pathname. When this request is received, the username is validated and a TID representing that authenticated instance of the user is returned. This TID must be included in all further requests made on behalf of the user.

The permission-based system need not execute a NET SHARE command; instead it sets up name/password (or whatever) information in its user definition files. The accessing user would type

```
NET USE \\machine-name\account-name <password>
```

and thereby "login" to the serving machine. He need not specify subtrees and so forth because the account-name/password pair establishes access permissions to everything on that machine.

This variation of Tree Connect is an aspect of the server's file system. Servers with user based protection schemes will always interpret the name supplied with Tree Connect as a user name. Users of Tree Connect simply provide a "name" and its associated "password"; they do not need to be aware of the server's interpretation of that name. If the name and password are successfully authenticated the caller receives access to the set of files protected by the name in the modes allowed by the server (also determined by the name/password pair).

The Tree Connect Command

From Consumer		To Consumer	
smb_com;	SMBtcon	smb com;	SMBtcon
smb_wct;	0	smb_wct;	2
smb_bcc;	min=4	smb_vwv[0];	max xmit size
smb_buf[];	ASCII -- 04	smb_vwv[1];	TID
	path/user-name	smb_bcc;	0
	ASCII--04		
	password		
	ASCII -- 04		
	dev name		

The device name is either <device>: for block device or LPT1: for a character device.

The path/username must be specified from the network root (including \\). The TID field in the request message is ignored by the server. The maximum transmit size field in the response message indicates the maximum size message that the server can handle. The consumer should not generate messages, nor expect to receive responses, larger than this. This should be constant for a given server.

Tree Connects must be issued for all subtrees accessed, even if they contain a null password.

Tree Connect may generate the following errors:

Error Class ERRDOS:

<implementation specific>

Error Class ERRSRV:

ERRerror

ERRbadpw

ERRinvnetname

<implementation specific>

Error Class ERRHRD:

<implementation specific>

The Tree Disconnect Command

From Consumer		To Consumer	
smb_com;	SMBtdis	smb_com;	SMBtdis
smb_wct;	0	smb_wct;	0
smb_bcc;	0	smb_bcc;	0

The file sharing connection identified by the TID is logically disconnected from the server. The TID will be invalidated; it will not be recognized if used by the consumer for subsequent requests.

Tree Disconnect may generate the following errors:

Error Class ERRDOS:

<implementation specific>

Error Class ERRSRV:

ERRinvnid

<implementation specific>

Error Class ERRHRD:

<implementation specific>

File Sharing Commands: Overview

The message definitions in this section indicate the command code and include the balance of the definition commencing at the field smb_wct. The omitted fields (smb_cls through smb_mid) are constant in the format and meaning defined in Section 1.0. When an error is encountered a server may return only the header portion of the response (i.e., smb_wct and smb_bcc both contain zero). The data objects used by these commands are described in section 6.0.

The use of commands other than those defined in this section will have undefined results.

The Open File Command

From Consumer		To Consumer	
smb_com;	SMBopen	smb_com;	SMBopen
smb_wct;	2	smb_wct;	7
smb_vwv[0];	r/w/share	smb_vwv[0];	file handle
smb_vwv[1];	attribute	smb_vwv[1];	attribute
smb_bcc;	min = 2	smb_vwv[2];	time1 low
smb_buf[];	ASCII -- 04	smb_vwv[3];	time1 high
	file pathname	smb_vwv[4];	file size low
		smb_vwv[5];	file size high
		smb_vwv[6];	access allowed
		smb_bcc;	0

This message is sent to obtain a file handle for a data file. The relevant tree id and any necessary additional pathname are passed. The handle returned can be used in subsequent read, write, lock, unlock and close messages. The file size and last modification time are also returned. The r/w/share word controls the mode. The file will be opened only if the requester has the appropriate permissions. The r/w/share word has the following format and values.

```
r/w/share format: - - - - - - - - rxxx yyyy
```

where: r = reserved

xxx = 0 -- MS-DOS Compatibility mode (exclusive to a VC, but that VC may have multiple opens). Support of this mode is optional. However, if it is not supported or is mapped to exclusive open modes, some existing MS-DOS applications may not work with network files. If reading map to deny write, otherwise map to deny read/write.

1 -- Deny read/write (exclusive to this open operation).

2 -- Deny write -- other users may access file in READ mode.

3 -- Deny read -- other users may access file in WRITE mode. Support of this mode is optional.

4 -- Deny none -- allow other users to access file in any mode for which they have permission.

yyyy = 0 -- Open file for reading.

1 -- Open file for writing.

2 -- Open file for reading and writing.

rxxx yyyy = 11111111 (hex FF)

FCB open: This type of open will cause an MS-DOS compatibility mode open with the read/write modes set to the maximum permissible, i.e., if the requester can have read and write access on the file, it will be opened in read/write mode.

The response message indicates the access permissions actually allowed in the "access allowed" field. This field may have the following values:

0 = read-only

1 = write-only

2 = read/write

File Sharing Notes:

1. File Handles (FIDs) are contained within the Virtual Circuit (VC) environment. A PID may reference

any FID established by itself or any other PID within its VC. The actual accesses allowed through the FID will depend on the open and deny modes specified when the file was opened.

2. The MS-DOS compatibility mode of file open provides exclusion at the VC level. A file open in compatibility mode may be opened (also in compatibility mode) any number of times for any combination of reading and writing (subject to the user's permissions) by any PID within the owning VC. If the first VC has the file open for writing, then the file may not be opened in any way by any PID within another VC. If the first VC has the file open only for reading, then other VCs may open the file, in compatibility mode, for reading. Once multiple VCs have the file open for reading, no VC is permitted to open the file for writing. No VC or PID may open the file in any mode other than compatibility mode.
3. The other file exclusion modes (Deny read/write, Deny write, Deny read, Deny none) provide exclusion at the file level. A file opened in any "Deny" mode may be opened again only for the accesses allowed by the Deny mode (subject to the user's permissions). This is true regardless of the identity of the second opener -- a PID within another VC, a PID within the same VC, or the PID that already has the file open. For example, if a file is open in "Deny write" mode a second open may only obtain read permission to the file.
4. Although FIDs are available to all PIDs on a VC, PIDs other than the owner may not have the full access rights specified in the open mode by the FID's creator. If the open creating the FID specified a deny mode, then any PID using the FID, other than the creating PID, will have only those access rights determined by "anding" the open mode rights and the deny mode rights, i.e., the deny mode is checked on all file accesses. For example, if a file is opened for Read/Write in Deny write mode, then other VC PIDs may only read from the FID and cannot write; if a file is opened for Read in Deny read mode, then the other VC PIDs can neither read nor write the FID.

If a file cannot be opened for any reason, including a conflict of share modes, a reply message indicating the cause of the failure will be returned.

Open may generate the following errors:

Error Class ERRDOS

ERRbadfile

ERRnofids

ERRnoaccess

ERRshare

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

ERRinvdevice

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Create File Command

From Consumer		To Consumer	
smb_com;	SMBcreate	smb_com;	SMBcreate
smb_wct;	3	smb_wct;	1
smb_vwv[0];	attribute	smb_vwv[0];	file handle
smb_vwv[1];	time low	smb_bcc;	0
smb_vwv[2];	time high		
smb_bcc;	min = 2		
smb_buf[];	ASCII -- 04		
	file pathname		

This message is sent to create a new data file or truncate an existing data file to length zero, and open the file. The handle returned can be used in subsequent read, write, lock, unlock and close messages.

Unprotected servers will require requesters to have create permission for the subtree containing the file in order to create a new file, or write permission for the subtree in order to truncate an existing one. The newly created file will be opened in compatibility mode with the access mode determined by the containing subtree permissions.

Protected servers will require requesters to have write permission on the file's parent directory in order to create a new file, or write permission on the file itself in order to truncate it. The access permissions granted on a created file will be read/write permission for the creator. Access permissions for truncated files are not modified. The newly created or truncated file is opened in read/write/compatibility mode.

Support of the create time supplied in the request is optional.

Create may generate the following errors:

Error Class ERRDOS

ERRbadpath

ERRnofids

ERRnoaccess

ERRbadaccess

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

ERRinvdevice

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Close File Command

From Consumer		To Consumer	
smb_com;	SMBclose	smb_com;	SMBclose
smb_wct;	3	smb_wct;	0
smb_vwv[0];	file handle	smb_bcc;	0
smb_vwv[1];	time low		
smb_vwv[2];	time high		
smb_bcc;	0		

The close message is sent to invalidate a file handle for the requesting process. All locks held by the requesting process on the file will be "unlocked". The requesting process can no longer use the file handle for further file access requests. The new modification time may be passed to the server. Server support of the modification time is optional; it may be ignored.

Close will cause all the file's buffers to be flushed to disk.

Close may generate the following errors:

Error Class ERRDOS

ERRbadfid

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Flush File Command

From Consumer		To Consumer	
smb_com;	SMBflush	smb_com;	SMBflush
smb_wct;	1	smb_wct;	0
smb_vwv[0];	file handle	smb_bcc;	0
smb_bcc;	0		

The flush message is sent to ensure all data and allocation information for the corresponding file has been written to non-volatile storage. When the file handle has a value -1 (hex FFFF), the server will perform a flush for all file handles associated with the consumer's process. The response is not sent until the writes are complete.

Note that this protocol does not require that only the specific file's data be written (flushed). It specifies that "at least" the file's data be written.

Flush may generate the following errors:

Error Class ERRDOS

ERRbadfid

ERRnoaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Read Command

From Consumer		To Consumer	
smb_com;	SMBread	smb_com;	SMBread
smb_wct;	5	smb_wct;	5
smb_vwv[0];	file handle	smb_vwv[0];	count
smb_vwv[1];	count of bytes	smb_vwv[1-4];	reserved (MBZ)
smb_vwv[2];	offset low	smb_bcc;	length of data + 3
smb_vwv[3];	offset high	smb_buf[];	Data Block -- 01
smb_vwv[4];	count left		length of data
smb_bcc;	0		data

The read message is sent to read bytes of a data file. The count of bytes field is used to specify the requested number of bytes. The offset field specifies the offset in the file of the first byte to be read. The count left field is advisory. If the value is not zero, then it is taken as an estimate of the total number of bytes that will be read -- including those read by this request. This additional information may be used by the server to optimize buffer allocation or read-ahead.

The count field in the response message indicates the number of bytes actually being returned. The count returned may be less than the count requested only if a read specifies bytes beyond the current file size. In this case only the bytes that exist are returned. A read completely beyond the end of file will result in a response of length zero. This is the only circumstance when a zero length response is generated. A count returned which is less than the count requested is the end of file indicator.

If a Read requests more data than can be placed in a message of the max-xmit-size for the TID specified, the server will abort the virtual circuit to the consumer.

Read may generate the following errors:

Error Class ERRDOS

ERRnoaccess

ERRbadfid

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Write Command

From Consumer		To Consumer	
smb_com;	SMBWrite	smb_com;	SMBWrite
smb_wct;	5	smb_wct;	1
smb_vwv[0];	file handle	smb_vwv[0];	count
smb_vwv[1];	count of bytes	smb_bcc;	0
smb_vwv[2];	offset low		
smb_vwv[3];	offset high		
smb_vwv[4];	count left		
smb_bcc;	length of data + 3		
smb_buf[];	Data Block -- 01		
	length of data		

The write message is sent to write bytes into a data file. The count of bytes field specifies the number of bytes to be written. The offset field specifies the offset in the file of the first byte to be written. The count left field is advisory. If the value is not zero, then it is taken as an estimate of the number of bytes that will be written -- including those written by this request. This additional information may be used by the server to optimize buffer allocation.

The count field in the response message indicates the actual number of bytes written, and for successful writes will always equal the count in the request message. If the number of bytes written differs from the number requested and no error is indicated, then the server has no disk space available with which to satisfy the complete write.

When a write specifies a byte range beyond the current end of file, the file will be extended. Any bytes between the previous end of file and the requested offset will be set to zero (ASCII null).

When a write specifies a length of zero, the file will be truncated to the length specified by the offset.

If a Write sends a message of length greater than the max-xmit-size for the TID specified, the server will abort the virtual circuit to the consumer.

Write may generate the following errors:

Error Class ERRDOS

ERRnoaccess

ERRbadfid

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Seek Command

From Consumer		To Consumer	
smb_com;	SMBlseek	smb_com;	SMBlseek
smb_wct	4	smb_wct;	2
smb_vwv[0];	file handle	smb_vwv[0];	offset-low
smb_vwv[1];	mode	smb_vwv[1];	offset-high
smb_vwv[2];	offset-low	smb_bcc;	0
smb_vwv[2];	offset-high		
smb_bcc;	min = 0		

The seek message is sent to set the current file pointer for the requesting process. The starting point of the seek is set by the "mode" field in the request. This may have the following values:

0 = seek from start of file

1 = seek from current file pointer

2 = seek from end of file

The response returns the new file pointer expressed as the offset from the start of the file, and may be beyond the current end of file. An attempt to seek to before the start of file set the file pointer to start of file.

Note: The "current file pointer" at the start of this command reflects the offset plus data length specified in the previous read, write or seek request, and the pointer set by this command will be replaced by the offset specified in the next read, write or seek command.

Seek may generate the following errors:

Error Class ERRDOS

ERRnoaccess

ERRbadfid

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Create Directory Command

From Consumer		To Consumer	
smb_com;	SMBmkdir	smb_com;	SMBmkdir
smb_wct;	0	smb_wct;	0
smb_bcc;	min = 2	smb_bcc;	0
smb_buf[];	ASCII -- 04		
	dir pathname		

The create directory message is sent to create a new directory. The appropriate TID and additional pathname are passed. The directory must not exist for it to be created.

Unprotected servers will require requesters to have create permission for the subtree containing the directory in order to create a new directory. The creator's access rights to the new directory will be determined by the containing subtree permissions.

Protected servers will require requesters to have write permission on the new directory's parent directory. The access permissions granted on a created directory will be read/write permission for the creator.

Create Directory may generate the following errors:

Error Class ERRDOS

ERRbadpath

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Delete Directory Command

From Consumer		To Consumer	
smb_com;	SMBrmdir	smb_com;	SMBrmdir
smb_wct;	0	smb_wct;	0
smb_bcc;	min = 2	smb_bcc;	0
smb_buf[];	ASCII -- 04		
	dir pathname		

The delete directory message is sent to delete an empty directory. The appropriate TID and additional pathname are passed. The directory must be empty for it to be deleted.

Unprotected servers will require the requester to have write permission to the subtree containing the directory to be deleted.

Protected servers will require the requester to have write permission to the target directory's parent directory.

The effect of a delete will be, to some extent, dependent on the nature of the server. Normally only the referenced directory name is deleted, the directory contents are only deleted when all the directory's names have been deleted.

In some cases a delete will cause immediate destruction of the directory contents.

Delete Directory may generate the following errors:

Error Class ERRDOS

ERRbadpath

ERRnoaccess

ERRremcd

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Query File System Information Command

When the NT SMB protocol has been negotiated, the Query File System Information Transact2 function supports the following additional information classes:

SMB_QUERY_FS_LABEL_INFO __

Equivalent to FileFsLabelInformation in the Windows NT I/O system. Uses the following response data format:

Data type	Field
ULONG	VolumeLabelLength;
UNICODE	VolumeLabel[];

SMB_QUERY_FS_VOLUME_INFO __

Equivalent to FileFsVolumeInformation in the Windows NT I/O system. Uses the following response data format:

Data type	Field
LARGE_INTEGER	VolumeCreationTime;
ULONG	VolumeSerialNumber;
ULONG	VolumenLabelLength;
BOOLEAN	SupportsObjects;
UNICODE	VolumeLabel[];

SMB_QUERY_FS_SIZE_INFO __

Equivalent to FileFsSizeInformation in the Windows NT I/O system. Uses the following response data format:

Data type	Field
LARGE_INTEGER	TotalAllocationUnits;
LARGE_INTEGER	AvailableAllocationUnits
ULONG	SectorsPerAllocationUnit;
ULONG	BytesPerSector;

SMB_QUERY_FS_DEVICE_INFO __

Equivalent to FileFsDeviceInformation in the Windows NT I/O system. Uses the following response data format.

Data type	Field
DEVICE_TYPE	DeviceType;
ULONG	Characteristics;

SMB_QUERY_FS_ATTRIBUTE_INFO __

Equivalent to FileFsAttributeInformation in the Windows NT I/O system. Uses the following response data format.

Data type	Field
ULONG	FileSystemAttributes;
LONG	MaximumComponentNameLength;
ULONG	FileSystemNameLength;
UNICODE	FileSystemName[];

Note that the VolumeLabel and FileSystemName fields in these additional information levels are always in Unicode format, even if Unicode is not a negotiated capability for the session.

The Rename File Command

From Consumer		To Consumer	
smb_com;	SMBmv	smb_com;	SMBmv
smb_wct;	1	smb_wct;	0
smb_vwv[0];	attribute	smb_bcc;	0
smb_bcc;	min = 4		
smb_buf[];	ASCII -- 04		
	old file pathname		
	ASCII -- 04		
	new file pathname		

The rename file message is sent to change the name of a file. The first file pathname must exist and the second must not. Both pathnames must be relative to the TID specified in the request. Open files may be renamed.

Multiple files may be renamed in response to a single request as Rename File supports "wild cards" in the file name (last component of the pathname). The wild card matching algorithm is described in the "Delete File" description.

The attribute field indicates the attributes that the target file(s) must have. If the attribute is zero, then only normal files are renamed. If the system file or hidden attributes are specified, then the rename is inclusive -- both the specified type(s) of files and normal files are renamed.

Unprotected servers require the requester to have both read and create permissions to the referenced subtree.

Protected servers require the requester to have write permission to the parent directories of both the source and destination files.

Rename is guaranteed to succeed if only the last component of the file pathnames differs. Other rename requests may succeed depending on the server implementation used.

Rename may generate the following errors:

Error Class ERRDOS

ERRbadfile

ERRnoaccess

ERRdiffdevice

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Get File Attributes Command

From Consumer		To Consumer	
smb_com;	SMBgetatr	smb_com;	SMBgetatr
smb_wct;	0	smb_wct;	10
smb_bcc;	min = 2	smb_vwv[0];	attribute
smb_buf[];	ASCII -- 04	smb_vwv[1];	time1 low
	file pathname	smb_vwv[2];	time1 high
	smb_vwv[3]		file size low
	smb_vwv[4]		file size high
	smb_vwv[5-9]		reserved (MBZ)
smb_bcc;	0		

The get file attributes message is sent to obtain information about a file. The attribute, time1, and file size fields must contain valid values for data files. The attribute and time1 fields must contain valid values for directories.

Get File Attributes may generate the following errors:

Error Class ERRDOS

ERRbadfile

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Delete File Command

From Consumer		To Consumer	
smb_com;	SMBunlink	smb_com;	SMBunlink
smb_wct;	1	smb_wct;	0
smb_vwv[0];	attribute	smb_bcc;	0
smb_bcc;	min = 2		
smb_buf[];	ASCII -- 04		
	file pathname		

The delete file message is sent to delete a data file. The appropriate TID and additional pathname are passed. A file must exist for it to be deleted. Read only files may not be deleted; the read-only attribute must be reset prior to file deletion.

Multiple files may be deleted in response to a single request as Delete File supports "wild cards" in the file name (last component of the pathname). "?" is the wild card for single characters, "*" or "null" will match any number of filename characters within a single part of the filename component. The filename is divided into two parts -- an eight character name and a three character extension. The name and extension are divided by a ".".

If a filename part commences with one or more "?"s then exactly that number of characters will be matched by the wildcards. For example, "??x" will equal "abx" but not "abcx" or "ax". When a filename part has trailing "?"s, then it will match the specified number of characters or less (for example, "x??" will match "xab", "xa" and "x", but not "xabc"). If only "?"s are present in the filename part, then it is handled as for trailing "?"s.

"*" or "null" match entire pathname parts, thus "*.abc" or ".abc" will match any file with an extension of "abc". "*.*", "*" or "null" will match all files in a directory.

The attribute field indicates the attributes that the target file(s) must have. If the attribute is zero then only normal files are deleted. If the system file or hidden attributes are specified then the delete is inclusive -- both the specified type(s) of files and normal files are deleted.

Unprotected servers will require the requester to have write permission to the subtree containing the file to be deleted.

Protected servers will require the requester to have write permission to the target file's parent directory.

The effect of a delete will be, to some extent, dependent on the nature of the server. Normally, only the referenced file name is deleted. The file contents are only deleted when all the file's names have been deleted and all file handles associated with it have been destroyed (closed).

In some cases (notably MS-DOS) a delete will cause immediate destruction of the file contents and invalidation of all FIDs associated with the file.

Delete File may generate the following errors:

Error Class ERRDOS

ERRbadfile

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Set File Attributes Command

From Consumer		To Consumer	
smb_com;	SMBsetatr	smb_com;	SMBsetatr
smb_wct;	8	smb_wct;	0
smb_vwv[0];	attribute	smb_bcc;	0
smb_vwv[1];	time 1 low		
smb_vwv[2];	time 1 high		
smb_vwv[3-7];	reserved (MBZ)		
smb_bcc;	min = 2		
smb_buf[];	ASCII -- 04		
	file pathname		
smb_nul[];	ASCII -- 04		
	null string		

The set file attributes message is sent to change the information about a file. Support of all parameters is optional. A server which does not implement one of the parameters will ignore that field. If the time1 field contains zero, then the file's time is not changed.

Unprotected servers require the requester to have write permission to the subtree containing the referenced file.

Protected servers will allow the owner of the file to use this command. Other legitimate users will be server dependent.

Set File Attributes may generate the following errors:

Error Class ERRDOS

ERRbadfunc

ERRbadpath

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRinvnid

ERRaccess

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Lock Record Command

From Consumer		To Consumer	
smb_com;	SMBlock	smb_com;	SMBlock
smb_wct;	5	smb_wct;	0
smb_vwv[0];	file handle	smb_bcc;	0
smb_vwv[1];	count low		
smb_vwv[2];	count high		
smb_vwv[3];	offset low		
smb_vwv[4];	offset high		
smb_bcc;	0		

The lock record message is sent to lock the given byte range. More than one non-overlapping byte range may be locked in a given file. Locks are coercive in nature. They prevent attempts to lock, read or write the locked portion of the file. Overlapping locks are not allowed. File addresses beyond the current end of file may be locked. Such locks will not cause allocation of file space.

Locks may only be unlocked by the process (PID) that performed the lock. The ability to perform locks is not tied to any file access permission.

Lock may generate the following errors:

Error Class ERRDOS

ERRbadfid

ERRlock

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Unlock Record Command

From Consumer		To Consumer	
smb_com;	SMBunlock	smb_com;	SMBunlock
smb_wct;	5	smb_wct;	0
smb_vwv[0];	file handle	smb_bcc;	0
smb_vwv[1];	count low		
smb_vwv[2];	count high		
smb_vwv[3];	offset low		
smb_vwv[4];	offset high		
smb_bcc;	0		

The unlock record message is sent to unlock the given byte range. The byte range must be identical to that specified in a prior successful lock request, and the unlock requester (PID) must be the same as the lock holder. If an unlock references an address range that is not locked it is treated as a no-op -- no action is taken and no error is generated.

Unlock may generate the following errors:

Error Class ERRDOS

ERRbadfid

ERRlock

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Create Temporary File Command

From Consumer		To Consumer	
smb_com;	SMBctemp	smb_com;	SMBctemp
smb_wct;	3	smb_wct;	1
smb_vwv[0];	attribute	SMB_VWV[0];	file handle
smb_vwv[1];	time low	smb_bcc;	min = 2
smb_vwv[2];	time high	smb_buf[];	ASCII = 04
smb_bcc;	min = 2		new file pathname
smb_buf[];	ASCII -- 04		
	directory pathname		

The server creates a data file in the directory specified in the request message and assigns a unique name to it. The file's name is returned to the requester. The file is opened in compatibility mode with read/write access for the requester.

Unprotected servers will require requesters to have create permission for the subtree containing the file. The newly created file will be opened in compatibility mode with the access mode determined by the containing subtree permissions.

Protected servers will require requesters to have write permission on the file's parent directory. The access permissions granted on a created file will be read/write permission for the creator. The newly created or truncated file is opened in read/write/compatibility mode.

Support of the create time supplied in the request is optional.

Create Temporary File may generate the following errors.

Error Class ERRDOS

ERRbadpath

ERRnofids

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

ERRinvdevice

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Process Exit Command

From Consumer		To Consumer	
smb_com;	SMBexit	smb_com;	SMBexit
smb_wct;	0	smb_wct	0
smb_bcc;	0	smb_bcc	0

This command informs the server that a consumer process has terminated. The server will close all files opened by the named process. This will automatically release all locks the process holds. Note that there is not a start process message; process-ids are assigned by the consumer.

Process Exit may generate the following errors:

Error Class ERRDOS

none

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Make New File Command

From Consumer		To Consumer	
smb_com;	SMBmknew	smb_com ;	SMBmknew
smb_wct;	3	smb_wct;	1
smb_vwv[0];	attribute	smb_vwv[0];	file handle
smb_vwv[1];	time low	smb_bcc;	0
smb_vwv[2];	time high		
smb_bcc	min = 2		
smb_buf[];	ASCII -- 04		
	file pathname		

The make new file message is sent to create a new data file. It is functionally equivalent to the create message, except it will always fail if the file already exists.

Make New File may generate the following errors:

Error Class ERRDOS:

ERRbadpath

ERRnofids

ERRnoaccess

<implementation specific>

Error Class ERRSRV:

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD:

<implementation specific>

The Check Path Command

From Consumer		To Consumer	
smb_con;	SMBchkpath	smb_com;	SMBchkpath
smb_wct;	0	smb_wct;	0
smb_bcc;	min = 2	smb_bcc;	0
smb_buf[];	ASCII -- 04		
	directory path		

The check path message is used to verify that a path exists and is a directory. No error is returned if the given path exists and the requester has read access to it. Consumer machines which maintain a concept of a "working directory" will find this useful to verify the validity of a "change working directory" command. Note that the servers do not have a concept of working directory. The consumer must always supply full pathnames (relative to the TID).

Check Path may generate the following errors:

Error Class ERRDOS

ERRbadpath

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Get Server Attributes Command

From Consumer		To Consumer	
smb_com;	SMBdskattr	smb_com;	SMBdskattr
smb_wct;	0	smb_wct;	5
smb_bcc;	0	smb_vwv[0];	# allocation units/server
		smb_vwv[1];	# blocks/allocation unit
		smb_vwv[2];	# block size (in bytes)
		smb_vwv[3];	# free allocation units
		smb_vwv[4];	reserved (media identifier code)
		smb_bcc;	0

This command is used to determine the total server capacity and remaining free space. The distinction between allocation units and disk blocks allows the use of the protocol with operating systems which allocate disk space in units larger than the physical disk block.

The blocking/allocation units used in this response may be independent of the actual physical or logical blocking/allocation algorithm(s) used internally by the server. However, they must accurately reflect the amount of space on the server.

The default value for smb_vwv[4] is zero.

Get Server Attributes may generate the following errors:

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Negotiate Protocol Command

From Consumer		To Consumer	
smb_com;	SMBnegprot	smb_com;	SMBnegprot
smb_wct;	0	smb_wct;	1
smb_bcc;	min = 2	smb_vwv[0];	index
smb_buf[];	Dialect -- 02	smb_bcc;	0
	dialect0		
	Dialect -- 02		
	dialect		

The consumer sends a list of dialects that he can communicate with. The response is a selection of one of those dialects (numbered 0 through n) or -1 (hex FFFF) indicating that none of the dialects were acceptable. The negotiate message is binding on the virtual circuit and must be sent. One and only one negotiate message may be sent; subsequent negotiate requests will be rejected with an error response, and no action will be taken.

The protocol does not impose any particular structure to the dialect strings. If you choose to implement you may choose to include, for example, version numbers in the string.

Negotiate may generate the following errors:

Error Class ERRDOS:

<implementation specific>

Error Class ERRSRV:

ERRerror

<implementation specific>

Error Class ERRHRD:

<implementation specific>

The Create Print File Command

From Consumer		To Consumer	
smb_con;	SMBsplopen	smb_com	SMBsplopen
smb_wct;	2	smb_wct	1
smb_vwv[0]	length of printer setup data	smb_vwv[0]	file handle
smb_vwv[1]	mode	smb_bcc	0
smb_bcc	min = 2		
smb_buf	ASCII -- 04		
	identifier string (max 15)		

This message is sent to create a new printer file. The file handle returned can be used for subsequent write and close commands. The file name will be formed by concatenating the identifier string and a server generated number. The file will be deleted once it has been printed.

The mode field can have the following values:

0 = Text mode. (DOS servers will expand TABs.)

1 = Graphics mode.

Protected servers grant write permission to the creator of the file. No other users will be given any permissions to the file. All users will have read permission to the print queue, but only the print server has write permission to it.

Create Print File may generate the following errors:

Error Class ERRDOS:

ERRbadpath

ERRnofids

ERRnoaccess

<implementation specific>

Error Class ERRSRV:

ERRerror

ERRqfull

ERRqtoobig

ERRinvnid

<implementation specific>

Error Class ERRHRD:

<implementation specific>

The Close Print File Command

From Consumer		To Consumer	
smb_com;	SMBsplclose	smb_com;	SMBsplclose
smb_wct;	1	smb_wct;	0
smb_vwv[0];	file handle	smb_bcc;	0
smb_bcc;	0		

This message invalidates the specified file handle and queues the file for printing. The file handle must reference a print file.

Close Print File may generate the following errors:

Error Class ERRDOS

ERRbadfid

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRqtoobig

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Write Print File Command

From Consumer		To Consumer	
smb_com;	SMBsplwr	smb_com;	SMBsplwr
smb_wct;	1	smb_wct;	0
smb_vwv[0];	file handle	smb_bcc;	0
smb_bcc;	min = 4		
smb_buf;	Data block -- 01		
	length of data		

This message appends the data block to the print file specified by the file handle. The file handle must reference a print file. The first block sent to a print file must contain the printer setup data. The length of this data is specified in the Create Print File request.

If a Write Print File sends a message of length greater than the max-xmit-size for the TID specified, the server will abort the virtual circuit to the consumer.

Write Print File may generate the following errors:

Error Class ERRDOS

ERRbadfid

ERRnoaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRqtoobig

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Get Print Queue Command

From Consumer		To Consumer	
smb_com;	SMBsplretq	smb_com;	SMBsplretq
smb_wct;	2	smb_wct;	2
smb_vwv[0];	max_count	smb_vwv[0];	count
smb_vwv[1];	start index	smb_vwv[1];	restart index
smb_bcc;	0	smb_bcc;	min = 3
		smb_buf;	Data block -- 01
			length of data
			queue elements

This message obtains a list of the elements currently in the print queue on the server. "start index" specifies the first entry in the queue to return, "max_count" specifies the maximum number of entries to return, this may be a positive or negative number. A positive number requests a forward search, a negative number indicates a backward search. In the response "count" indicates how many entries were actually returned. "Restart index" is the index of the entry following the last entry returned; it may be used as the start index in a subsequent request to resume the queue listing.

Get Print Queue will return less than the requested number of elements only when the top or end of the queue is encountered.

The format of the queue elements returned is:

smb_date	WORD	file date (yyyyyyy mmmm ddddd)
smb_time	WORD	file time (hhhhh mmmmmm xxxxx)
		where 'xxxxx' is in 2 second increments
smb_status	BYTE	entry status
		01 = held or stopped
		02 = printing
		03 = awaiting print
		04 = in intercept
		05 = file had error
		06 = printer error
		07-FF = reserved
smb_fileWORD		spool file number (from create print file request)
smb_sizelo	WORD	low word of file size
smb_sizehi	WORD	high word of file size
smb_res	BYTE	reserved
smb_name	BYTE[16]	originator name (from create print file request)

Get Print Queue may generate the following errors:

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRqeof

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Get Machine Name Command

From Consumer		To Consumer	
smb_com	SMBgetmac	smb_com	SMBgetmac
smb_wct	0	smb_wct	0
smb_bcc	0	smb_bcc	min = 2
	smb_buf		ASCII -- 04
			machine name (max 15 bytes)

The Get Machine Name command obtains the machine name of the target machine. It is used prior to the Cancel Forward command to determine which machine to send the Cancel Forward command to. Get Machine Name is sent to the forwarded name to be canceled, and the server then returns the machine name to which the Cancel Forward command must be sent.

Get Machine Name may return the following errors.

Error Class ERRDOS:

<implementation specific>

Error Class ERRSRV:

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD:

<implementation specific>

Message Commands: Overview

These commands provide a message delivery system between users of systems participating in the network. The message commands cannot use VCs established for the file sharing commands. A separate VC, dedicated to messaging, must be established.

Messaging services should support message forwarding. By convention user names used for message delivery have a suffix (in byte 16) of "03", forwarded names have a suffix of "05". The algorithm for sending messages is to first attempt to deliver the message to the forwarded name, and only if this fails to attempt to deliver to the normal name.

The Send Single Block Message Command

From Consumer		To Consumer	
smb_com	SMBsends	smb_com	SMBsends
smb_wct	0	smb_wct	0
smb_bcc	min = 7	smb_bcc	0
smb_buf[]	ASCII -- 04		
	originator name (max 15 bytes)		
	ASCII -- 04		
	destination (max 15 bytes)		
	Data Block -- 01		
	length of message (max 128 bytes)		
	message (max 128 bytes)		

Send Single Block Message sends a short message (up to 128 bytes in length) to a single destination (user).

The names specified in this message do not include the one byte suffix ("03" or "05").

Send Single Block Message may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRpaused

ERRmsgoff

ERRnoroom

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Send Broadcast Message Command

From Consumer		To Consumer
smb_com;	SMBsendb	No Response
smb_wct;	0	
smb_bcc;	min = 8	
smb_buf[];	ASCII -- 04	
	originator name (max 15 bytes)	
	ASCII -- 04	
	"*"	
	Data block -- 01	
	length of message (max 128 bytes)	
	message (max 128 bytes)	

Send Broadcast Message sends a short message (up to 128 bytes in length) to every user in the network.

The name specified in this message does not include the one byte suffix ("03").

There is no response message to this command, thus Send Broadcast Message cannot generate errors.

The Send Start of Multi-Block Message Command

From Consumer		To Consumer	
smb_com;	SMBsendstrt	smb_com;	SMBsendstrt
smb_wct;	0	smb_wct;	1
smb_bcc;	min = 0	smb_vwv;	message group ID
smb_buf[];	ASCII -- 04	smb_bcc;	0
	originator name (max 15 bytes)		
	ASCII -- 04		
	destination name (max 15 bytes)		

This command informs the server that a multi-block message will be sent. The server returns a message group ID to be used to identify the message blocks when they are sent.

The names specified in this message do not include the one byte suffix ("03" or "05").

Send Start of Multi-block Message may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRpaused

ERRmsgoff

ERRnoroom

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Send Text of Multi-Block Message Command

From Consumer		To Consumer	
smb_com	SMBsendtxt	smb_com	SMBsendtxt
smb_wct	1	smb_wct	0
smb_vwv	message group ID	smb_bcc	0
smb_bcc	min = 3		
smb_buf[]	Data Block -- 01		
	length of message (max 128)		
	message (max 128 bytes)		

This command delivers a segment of a multi-block message to the server. It must contain a valid message group ID returned by an earlier Start Multi-block Message command.

A maximum of 128 bytes of message may be sent with this command. A multi-block message cannot exceed 1600 bytes in total length (sum of all segments sent with a given message group ID).

Send Text of Multi-block Message may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRpaused

ERRmsgoff

ERRnoroom

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Send End of Multi-Block Message Command

From Consumer		To Consumer	
smb_com;	SMBsendend	smb_com;	SMBsendend
smb_wct;	0	smb_wct;	0
smb_vwv;	message group ID	smb_bcc;	0
smb_bcc;	0		

This command signals the completion of the multi-block message identified by the message group ID.

Send End of Multi-block Message may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRpaused

ERRmsgoff

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Forward User Name Command

From Consumer		To Consumer	
smb_com;	SMBfwdname	smb_com;	SMBfwdname
smb_wct;	0	smb_wct;	0
smb_bcc;	min = 2	smb_bcc;	0
smb_buf[];	ASCII -- 04		
	forwarded name (max 15 bytes)		

This command informs the server that it should accept messages sent to the forwarded name.

The name specified in this message does not include the one byte suffix ("03" or "05").

Forward User Name may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRrmuns

<implementation specific>

Error Class ERRHRD

<implementation specific>

Error Codes

The following error codes may be generated with the ERRDOS error class. The XENIX errors equivalent to each of these errors are noted at the end of the error description.

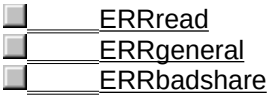
- ERRbadfunc
- ERRbadfile
- ERRbadpath
- ERRnofids
- ERRnoaccess
- ERRbadfid
- ERRbadmcb
- ERRnomem
- ERRbadmem
- ERRbadenv
- ERRbadformat
- ERRbadaccess
- ERRbaddata
- ERR 14
- ERRbaddrive
- ERRremcd
- ERRdiffdevice
- ERRnofiles
- ERRbadshare
- ERRlock
- ERRfilexists

The following error codes may be generated with the ERRSRV error class.

- ERRerror
- ERRbadpw
- ERRbadtype
- ERRaccess
- ERRinvnid
- ERRinvnetname
- ERRinvdevice
- ERRqfull
- ERRqtoobig
- ERRqeof
- ERRinvpfid
- ERRpaused
- ERRmsgoff
- ERRnoroom
- ERRnosupport

The following error codes may be generated with the ERRHRD error class. The XENIX errors equivalent to each of these errors are noted at the end of the error description.

- ERRnowrite
- ERRbadunit
- ERRnotready
- ERRbadcmd
- ERRdata
- ERRbadreq
- ERRseek
- ERRbadmedia
- ERRbadsector
- ERRnopaper
- ERRwrite



The Cancel Forward Command

From Consumer		To Consumer	
smb_com;	SMBcancelf	smb_com;	SMBcancelf
smb_wct;	0	smb_wct;	0
smb_bcc;	min = 2	smb_bcc;	0
smb_buf[];	ASCII -- 04		
	forwarded name (max 15 bytes)		

The Cancel Forward command cancels the effect of a prior Forward User Name command. The addressed server will no longer accept messages for the designated user name.

The name specified in this message does not include the one byte suffix ("05").

Cancel Forward may generate the following errors:

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

Exception Handling

Exception handling is built upon the various environments supported by the file sharing protocol (see ARCHITECTURAL MODEL section). When any environment is dissolved (in either an orderly or disorderly fashion) all contained environments are dissolved. The hierarchy of environments is summarized below:

Virtual Circuit

TID

PID

FID

As can be seen from this summary, the Virtual Circuit (VC) is the key environment. When a VC is dissolved, the server processes (or equivalent) are terminated; the TIDs, PIDs and FIDs are invalidated, and any outstanding request is dropped -- a response will not be generated.

The termination of a PID will close all FIDs it contains. The destruction of TIDs and FIDs has no affect on other environments.

If the server receives a message with a bad format, e.g., lacks the "FFSMB" header, it may abort the VC.

If a server is unable to deliver responses to a consumer within *n* seconds, it considers the consumer dead and drops the VC to it (we anticipate that n will be a function of the transport round trip delay time).

Message Objects

attribute: The attributes of the file. Portions of this field indicate the type of file. The rest of the contents are server specific. The MS-DOS server will return the following values in attribute (bit0 is the low order bit):

Generic Attributes:

bit4 - directory

MS-DOS Attributes:

bit0 - read only file

bit1 - "hidden" file

bit2 - system file

bit3 - volume id

bit5 - archive file

bits6-15 - reserved

Support of the Generic Attributes is mandatory; support of the MS-DOS Attributes is optional. If the MS-DOS Attributes are not supported, attempts to set them must be rejected and attempts to match on them (e.g., File Search) must result in a null response.

count of bytes

The count of bytes (1 to the maximum size) read/written. The maximum size is server specific.

count left

The count of bytes not yet read/written. This field is advisory only and is used for read-ahead in the server.

count-returned

The actual number of directory entries that are returned by a file-search response.

data read/written

The actual data.

dialect-0-dialect-n

A list of dialects, each of which identifies a requested protocol and version in a string.

Examples: "SNA-REV2" "TEST PROTOCOL" "RING.2"

dir_info

A data block containing an array of directory entries returned by file search.

dir pathname

An ASCII string, null terminated, that defines the location of a file within the tree. Use the '\' character to separate components. The last component names a directory. The maximum size of this field is server specific. The pathname is relative to a TID and may or may not commence with a '\'.

file handle

The file identifier obtained form an open, create, make new file, and make temp file. File handles are unique within a process id.

file pathname

An ASCII string, null terminated, that defines the location of a file within the tree. Use the '\' character to separate components. The last component names a file. The maximum size of this field is server specific. The pathname is relative to a TID and may or may not commence with a '\'.

file size low/hi

Low and hi words of a 32-bit long field that represents the Data file size.

identifier string

The "originator name" of the owner of a print file. The server will add a number to it to generate a unique file name. This is a null terminated ASCII string.

max-count

The maximum number of directory entries that can be returned by a file-search response.

max xmit size

The maximum size message that a server can handle.

message group ID

A message group ID uniquely identifies a multi-block message.

non-owner access

The access rights of other than the owner.

offset low/hi

The low and hi words of a 32-bit offset.

owner access

The access rights of the owner.

owner id

The user id of the owner of the file.

password

May be used with the pathname for authentication by the NET USE command. This is a null terminated ASCII string.

r/w/share

This field defines the file mode. It contains fields that represent the following:

Access modes:

Read

Write

Read/Write

Sharing modes:

Exclusive

No restriction

Multiple Readers

Multiple Writers

search-status

A variable block reserved for server specific information that is passed from each file search response message to the next file search request.

time1 low/hi

File modification time. these two words define a 32 bit field that contains the modification time expressed as seconds past Jan 1 1970 (local time zone). A value of zero indicates a null time field.

Data Buffer Formats (smb_buf)

The data portion of these messages typically contains the data to be read or written, file paths, or directory paths. The format of the data portion depends on the message. All fields in the data portion have the same format. In every case it consists of an identifier byte followed by the data.

Data Identifier Bytes

Name	Description	Value
Data Block	See Below	01
Dialect	Null terminated ASCII String	02
Pathname	Null terminated ASCII String	03
ASCII	Null terminated ASCII String	04
Variable block	See Below	05

When the identifier indicates a Data Block or Variable Block then the format is a word indicating the length followed by the data. ASCII strings are null terminated.

Despite the flexible encoding scheme, no field of a data portion may be omitted or included out of order. In addition, neither an smb_wct nor smb_bcc of value 0 at the end of a message may be omitted.

Command Codes

The following values have been assigned for the protocol commands.

Command	Value	Meaning
SMBmkdir	0x00	create directory
SMBrmdir	0x01	delete directory
SMBopen	0x02	open file
SMBcreate	0x03	create file
SMBclose	0x04	close file
SMBflush	0x05	flush file
SMBunlink	0x06	delete file
SMBmv	0x07	rename file
SMBgetatr	0x08	get file attributes
SMBsetatr	0x09	set file attributes
SMBread	0x0A	read from file
SMBwrite	0x0B	write to file
SMBlock	0x0C	lock byte range
SMBunlock	0x0D	unlock byte range
SMBctemp	0x0E	create temporary file
SMBmknew	0x0F	make new file
SMBchkpth	0x10	check directory path
SMBexit	0x11	process exit
SMBlseek	0x12	seek
SMBtcon	0x70	tree connect
SMBtdis	0x71	tree disconnect
SMBnegprot	0x72	negotiate protocol
SMBdskattr	0x80	get disk attributes
SMBsearch	0x81	search directory
SMBsplopen	0xC0	open print spool file
SMBsplwr	0xC1	write to print spool file
SMBsplclose	0xC2	close print spool file
SMBsplretq	0xC3	return print queue
SMBsends	0xD0	send single block message
SMBsendb	0xD1	send broadcast

		message
SMBfwdname	0xD2	forward user name
SMBcancelf	0xD3	cancel forward
SMBgetmac	0xD4	get machine name
SMBsendstrt	0xD5	send start of multi-block message
SMBsendend	0xD6	send end of multi-block message
SMBsendtxt	0xD7	send text of multi-block message

Error Classes

Class	Value	Definition
SUCCESS	0	The request was successful.
ERRDOS	0x01	Error is generated by server operating system
ERRSRV	0x02	Error is generated by server network file manager
ERRHRD	0x03	Error is an hardware error (MS-DOS int 24)
ERRCMD	0xFF	Command was not in the "SMB" format (optional)

The following error codes may be generated with the SUCCESS error class.

Code	Value	Definition
SUCCESS	0	The request was successful
BUFFERED	0x54	Message has been buffered
LOGGED	0x55	Message has been logged
DISPLAYED	0x56	User message displayed

Appendix A: An Example

In this example a MS-DOS machine will access a file on a remote machine that is running a server that supports MS-DOS file sharing.

- STEP 1: Using protocols described elsewhere, the MS-DOS machine has obtained a virtual circuit (VC) to the server on the remote machine. The MS-DOS machine will then generate "Negotiate Message" on the VC with a dialect field that contains "PC NETWORK PROGRAM 1.0". The remote server will respond with a "Negotiate Reply Message" which will contain the index of the dialect string that contained "PC NETWORK PROGRAM 1.0", in this case 1, which indicates that it will service that protocol.
- STEP 2: The MS-DOS machine now generates a "Tree Connect Message" with a pathname and a password. The remote server will respond with a "Tree Connect Response Message" indicating that the password has been validated permitting access to the associated sub-tree. A "Tree ID" is returned for future use.
- STEP 3: The MS-DOS machine wishes to open and read a file on the remote server. This would be in response to a program that referenced a file on that remote system. The MS-DOS machine will generate, in response to a user program open, an "Open Message" with the "file path" of the file to be opened along with the mode information and the tree id. The file-path must not contain the path specified in the tree connect message. The server will respond with a "Open Reply Message" which will contain a file handle for use with future messages. It will also return the file size and modification time.
- STEP 4: The MS-DOS machine now reads the file, in response to user program file reads. It will generate a "Read Message" with the "file handle" obtained from the "open message". The message will contain a count of bytes to be read and an offset within the file to start reading, and possibly count indicating future requests. The server will respond with a "Read Reply Message" with the count of data read and the data.
- STEP 5: Some number of "Read Messages" and possibly "Write Messages" are transmitted, and eventually the file is closed by the user process. The MS-DOS machine will generate a "Close Message" which contains the "file handle" obtained from the "Open Response Message" and a new modification time. The server responds with a "Close Response Message".
- STEP 6: At some time the MS-DOS machine generates a "Tree Disconnect Message" and receives a "Tree Disconnect Response Message." At this point the VC may be de-allocated.

Microsoft Networks/OpenNET

FILE SHARING PROTOCOL

INTEL Part Number 138446

Document Version 2.0

November 7, 1988

Microsoft Corporation
Intel Corporation

The File Search Command

From Consumer		To Consumer	
smb_com;	SMBsearch	smb_com;	SMBSearch
smb_wct;	2	smb_wct;	1
smb_vwv[0];	max_count	smb_vwv[0];	count-returned
smb_vwv[1];	attribute	smb_bcc;	min = 3
smb_bcc;	min = 5	smb_buf[];	Variable block -- 05
smb_buff;	ASCII -- 04		length of data
	file pathname		directory entries
	Variable block -- 05		
	length of data		
	search status		

This command is used to search directories. The file path name in the request specifies the file to be sought. The attribute field indicates the attributes that the file must have. If the attribute is zero then only normal files are returned. If the system file, hidden or directory attributes are specified then the search is inclusive -- both the specified type(s) of files and normal files are returned. If the volume label attribute is specified then the search is exclusive, and only the volume label entry is returned.

The max_count field specifies the number of directory entries to be returned. The response will contain one or more directory entries as determined by the count-returned field. No more than max_count entries will be returned. Only entries that match the sought filename/attribute will be returned.

The search-status field must be null (length = 0) on the initial search request. Subsequent search requests intended to continue a search must contain the search-status field extracted from the last directory entry of the previous response. The search-status field is self-contained, for on calls containing a search-status neither the attribute or pathname fields will be valid in the request. Search-status has the following format:

Data type	Field	Value
BYTE	sr_res;	reserved
		bit 7 -- reserved for consumer use
		bit 5,6 -- reserved for system use (must be preserved)
		bits 0-4 -- reserved for server (must be preserved)
BYTE	sr_name[11];	pathname sought. Format: 0-3 character extension, left justified (in last 3 chars)
BYTE	sr_server[5];	available for server use (1st byte must be non-zero)
BYTE	sr_res[4];	reserved for consumer use

A File Search request will terminate when either the requested maximum number of entries that match the named file are found, or the end of directory is reached without the maximum number of matches being found. A response containing no entries indicates that no matching entries were found between the starting point of the search and the end of directory.

There may be multiple matching entries in response to a single request as File Search supports "wild cards" in the file name (last component of the pathname). The wild card matching algorithm is described in the "Delete File" description.

Unprotected servers require the requester to have read permission on the subtree containing the directory searched.

Protected servers require the requester to have read permission on the directory searched.

If a File Search requests more data than can be placed in a message of the max_xmit_size for the TID specified, the server will abort the virtual circuit to the consumer.

dir_info entries have the following format:

Data type	Field	Value
BYTE	find_buf_reserved[21];	reserved (search_status)
BYTE	find_buf_attr;	attribute
WORD	find_buf_time;	modification time (hhhhh mmmmm xxxxx), where xxxxx is in two second increments
WORD	find_buf_date;	modification date (yyyyy mmmm ddddd)
WORD	find_buf_size_1;	file size -- low word
WORD	find_buf_size_h;	file size -- high word
BYTE	find_buf_pname[13];	file name -- ASCII (null terminated)

File Search may generate the following errors:

Error Class ERRDOS

ERRnofiles

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

Connection Protocols

The **NET SHARE** command generates no network messages. The server package remembers the pathname prefix and the password.

The **NET USE** command generates a message containing the path/username and the password. The serving machine verifies the combination and returns an error code or an identifier. The full name (path or user) is included in the Tree Connect request message and the identifier identifying the connection is returned in the smb_tid field. The meaning of this identifier (tid) is server specific; the requester must not associate any specific meaning to it.

The server makes whatever use of the tid field it desires. Normally it is an index into a server table which allows the server to optimize its response.

The NT Transact Command

The NT Transact SMB is used for commands that potentially need to transfer a large amount of data (greater than the negotiated buffer size). NT Transact operates in the same way as the original Transact. See Microsoft Networks SMB File Sharing Protocol Extensions Version 2.0 for a detailed description of Transact.

The command code for the NT Transact primary SMB is SMB_COM_NT_TRANSACT. The command code for the NT Transact secondary SMB is SMB_COM_NT_TRANSACT_SECONDARY.

The interim response format has command code SMB_COM_NT_TRANSACT and is only ever returned in response to a primary request. The response format can have either SMB_COM_NT_TRANSACT or SMB_COM_NT_TRANSACT_SECONDARY as its command code and is returned in response to either a primary or secondary request. Interim responses and primary responses are distinguished by the value of the WordCount field.

Primary Request Format:

Data type	Field
UCHAR	WordCount;
UCHAR	MaxSetupCount;
USHORT	Flags;
ULONG	TotalParameterCount;
ULONG	TotalDataCount;
ULONG	MaxParameterCount;
ULONG	MaxDataCount;
ULONG	ParameterCount;
ULONG	ParameterOffset;
ULONG	DataCount;
ULONG	DataOffset;
UCHAR	SetupCount;
USHORT	Function;
USHORT	Setup[];
USHORT	ByteCount;
UCHAR	Pad1[];
UCHAR	Parameters[];
UCHAR	Pad2[];
UCHAR	Data[];

The following table explains the components of this format:

This component	Specified
WordCount __	The number of words following. Must be equal to 19 + SetupCount.

MaxSetupCount __	The maximum number of setup words that the server may return in the response.
Flags __	Flags. None are currently defined.
TotalParameterCount __	The total number of parameter bytes that are being sent in the request.
TotalDataCount __	The total number of data bytes that are being sent in the request.
MaxParameterCount __	The maximum number of parameter bytes that the server may return in the response.
MaxDataCount __	The maximum number of data bytes that the server may return in the response.
ParameterCount __	The number of parameter bytes that are present in this SMB.
ParameterOffset __	The offset from the start of the SMB header to the parameter bytes.
DataCount __	The number of data bytes that are present in this SMB.
DataOffset __	The offset from the start of the SMB header to the data bytes.
SetupCount __	The number of setup words in the request.
Function __	The transaction function code.
Setup __	Supplies setup words, the contents of which are interpreted based on the function code.
ByteCount __	The number of bytes following.
Pad1 __	Zero to three pad bytes to align Parameters on a four-byte boundary. The pad bytes should be zero. If ParameterCount is zero, Pad1 is not present.
Parameters __	ParameterCount parameter bytes.
Pad2 __	Zero to three pad bytes to align Data on a four-byte boundary. The pad bytes should be zero. If DataCount is zero, Pad2 is not present.
Data __	DataCount data bytes.

Interim Response Format:

Data type	Field
UCHAR	WordCount;
USHORT	ByteCount;

The following table explains the components of this response:

This component's value	Must contain

WordCount __	The value 0
ByteCount __	The value 0

Secondary Request Format:

Data type	Field
UCHAR	WordCount;
UCHAR	Reserved1;
USHORT	Reserved2;
ULONG	TotalParameterCount;
ULONG	TotalDataCount;
ULONG	ParameterCount;
ULONG	ParameterOffset;
ULONG	ParameterDisplacement;
ULONG	DataCount;
ULONG	DataOffset;
ULONG	DataDisplacement;
UCHAR	Reserved3;
USHORT	ByteCount;
UCHAR	Pad1[];
UCHAR	Parameters[];
UCHAR	Pad2[];
UCHAR	Data[];

The following table explains the components of this response:

This component	Specified
WordCount __	Must contain the value 18.
Reserved1 __	Is a reserved field and should be zero.
Reserved2 __	Is a reserved field and should be zero.
TotalParameterCount __	The total number of parameter bytes that are being sent in the request.
TotalDataCount __	The total number of data bytes that are being sent in the request.
ParameterCount __	The number of parameter bytes that are present in this SMB.
ParameterOffset __	The offset from the start of the SMB header to the parameter bytes.

ParameterDisplacement __	The offset from the start of the overall parameter block to the parameter bytes that are contained in this message.
DataCount __	The number of data bytes that are present in this SMB.
DataOffset __	The offset from the start of the SMB header to the data bytes.
DataDisplacement __	The offset from the start of the overall data block to the data bytes that are contained in this message.
Reserved3 __	Is a reserved field and should be zero.
ByteCount __	The number of bytes following.
Pad1 __	Zero to three pad bytes to align Parameters on a four-byte boundary. The pad bytes should be zero. If ParameterCount is zero, Pad1 is not present.
Parameters __	ParameterCount parameter bytes.
Pad2 __	Zero to three pad bytes to align Data on a four-byte boundary. The pad bytes should be zero. If DataCount is zero, Pad2 is not present.
Data __	DataCount data bytes.

Response Format:

Data type	Field
UCHAR	WordCount;
UCHAR	Reserved1;
USHORT	Reserved2;
ULONG	TotalParameterCount;
ULONG	TotalDataCount;
ULONG	ParameterCount;
ULONG	ParameterOffset;
ULONG	ParameterDisplacement;
ULONG	DataCount;
ULONG	DataOffset;
ULONG	DataDisplacement;
UCHAR	SetupCount;
USHORT	Setup[];
USHORT	ByteCount;
UCHAR	Pad1[];
UCHAR	Parameters[];
UCHAR	Pad2[];

UCHAR Data[];

The following table explains the components of this response:

This component	Specifies
WordCount __	Must contain the value 18 + SetupCount.
Reserved1 __	Is a reserved field and should be zero.
Reserved2 __	Is a reserved field and should be zero.
TotalParameterCount __	The total number of parameter bytes that are being sent in the response.
TotalDataCount __	The total number of data bytes that are being sent in the response.
ParameterCount __	The number of parameter bytes that are present in this SMB.
ParameterOffset __	The offset from the start of the SMB header to the parameter bytes.
ParameterDisplacement __	Specifies the offset from the start of the overall parameter block to the parameter bytes that are contained in this message.
DataCount __	The number of data bytes that are present in this SMB.
DataOffset __	The offset from the start of the SMB header to the data bytes.
DataDisplacement __	The offset from the start of the overall data block to the data bytes that are contained in this message.
SetupCount __	The number of setup words in the response.
Setup __	Supplies setup words, the contents of which are interpreted based on the function code.
ByteCount __	The number of bytes following.
Pad1 __	Zero to three pad bytes to align Parameters on a four-byte boundary. The pad bytes should be zero. If ParameterCount is zero, Pad1 is not present.
Parameters __	ParameterCount parameter bytes.
Pad2 __	Zero to three pad bytes to align Data on a four-byte boundary. The pad bytes should be zero. If DataCount is zero, Pad2 is not present.
Data __	DataCount data bytes.

The User Logoff and X Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 2
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_reh2;	reserved (must be zero)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
WORD	smb_bcc;	value 0

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value 2
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_res2;	reserved (pad to word)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
WORD	smb_bcc;	value = 0

Service definition:

This protocol is used to "Log Off" the user (identified by the UID value in smb_uid) previously "Logged On" via the Session Set Up protocol.

The server will remove this UID from its list of valid UIDs for this session. Any subsequent protocol containing this UID (in smb_uid) received (on this session) will be returned with an access error.

Another Session Set Up ("User Logon") must be sent in order to reinstate the user on the session.

Session Termination also causes the UIDs registered on the session to be invalidated. When the session is reestablished, Session Setup request(s) must again be used to validate each user.

The following are the only valid protocol request commands for smb_com2(X) for User Logoff and X:

SESSION SETUP and X

User Logoff may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Find Close Command

Request Format

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_handle;	Find handle
WORD	smb_bcc;	value = 0

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 0
WORD	smb_bcc;	value = 0

Service

The Find Close protocol closes the association between a search handle returned following a successful FIND FIRST request sent to the server using the TRANSACT2 protocol and the resulting system file search. This request allows the server to free any resources held in support of the open handle.

The Find Close protocol is used to match the DosFindFirst2 OS/2 system call.

Find Close may generate the following errors.

Error Class ERRDOS

ERRbadfid

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Find Notify Close Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_handle;	Find notify handle
WORD	smb_bcq;	value = 0

Response Format:

Data type	Field	Value
BYTE	smb_yct;	value = 0
WORD	smb_bcc;	value = 0

Service

The Find Notify Close protocol closes the association between a directory handle returned following a resource monitor established using TRANSACT2_FINDNOTIFYFIRST request to the server and the resulting system directory monitor. This request allows the server to free any resources held in support of the open handle.

The Find Close protocol is used to match the DosFindNotifyClose OS/2 system call.

Find Notify Close may generate the following errors.

Error Class ERRDOS

ERRbadfid

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Session Setup and X Command

The SessSetupX request SMB for the LM 2.1 protocol has an additional three fields that are not contained in the LM 1.0 version of this request. These fields are used to indicate the name of the domain on which the client was authenticated (smb_domain), the type of operating system the client machine is using (smb_nativeos), and what kind of LAN Manager software the client is using (smb_nativelm). All these fields are null-terminated strings, and their orientation in the SMB is indicated in the data structure below (* indicated new field).

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_reh2;	reserved (must be zero)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
WORD	smb_bufsize;	the consumer's max buffer size
WORD	smb_mpxmax;	actual maximum multiplexed pending requests
WORD	smb_vc_num;	0 = first (only), non zero - additional VC number
DWORD	smb_sesskey;	Session Key (valid only if smb_vc_num 0)
WORD	smb_apasslen;	size of account password (smb_apasswd)
DWORD	smb_rsvd;	reserved
WORD	smb_bcc;	minimum value = 0
BYTE	smb_apasswd[*];	account password (* = smb_apasslen value)
BYTE	smb_aname[];	account name string
BYTE	smb_domain[];	name of domain on which client was authenticated
BYTE	smb_nativeos[];	native operating system of client
BYTE	smb_nativelm[];	native LAN Manager type

Some examples of the smb_nativeos field are:

"OS/2 1.0", "OS2 1.21", MS-DOS 5.0", "UNIX BSD 4.0", etc.

Examples of the smb_nativelm field are:

"LAN Manager 2.1", "LAN Server 2.0", etc.

The SessSetupX response SMB for the LM 2.1 protocol contains two fields not contained in the LM 1.0 implementation. These fields are the servers corresponding smb_nativeos and smb_nativelm fields. These fields are null-terminated strings. Their

orientation in the SMB are indicated in the data structure below (* indicates new field).

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 3
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_res2;	reserved (pad to word)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb-wct)
WORD	smb_action;	request mode:
		bit() = Logged in successfully - BUT as GUEST
WORD	smb_bcc;	min value = 0
BYTE	smb_nativeos[];	server's native operating system
BYTE	smb_nativelm[];	server's native LM type

Service definition:

This protocol function is unchanged from LANMAN 1.0 except that the station establishing the connection may now verify the validity of the server to which the request was made. The LANMAN 2.0 SESS_SETUPandX request uses a reserved DWORD field from the LANMAN 1.0 request to pass the length and offset of an encryption key contained in the data of the request to the server. The server will use the encryption key to format the smb_encresp field of the response protocol. The station may then use this response to validate the server session.

The LANMAN 2.0 SESS_SETUPandX also returns a UID in the smb_uid field. This is a validated UID which must be supplied by the workstation on all subsequent requests to the server.

The Copy Command

Request Format:

Data type	Component	Value
BYTE	smb_wct;	value = 3
WORD	smb_tid2;	second (destination) path tid
WORD	smb_ofun;	what to do if destination file exists
WORD	smb_flags;	flags to control copy operations:
		bit 0 - destination must be a file
		bit 1 - destination must be a directory
		bit 2 - copy destination mode; 0 = binary, 1 = ASCII
		bit 3 - copy source mode; 0 = binary, 1 = ASCII
		bit 4 - verify all writes
		bit 5 - tree copy. Source must be a directory. Copy mode must be binary.
		When tree copy is selected, smb_cct field in the response protocol is undefined.
		bit 6 - Action when source as EA and dest
		does not support EAs; 0 = Discard EAs, 1 = Fail copy.
WORD	smb_bcc;	minimum value = 2
BYTE	smb_path[];	pathname of source file
BYTE	smb_new_path[];	pathname of destination file

Response Format:

Data type	Component	Value
BYTE	smb_wct;	value = 1
WORD	smb_cct;	number of files copied
WORD	smb_bcc;	minimum value = 0
BYTE	smb_errfile[];	pathname of file where error occurred - ASCIIZ

Service:

The COPY protocol function for LANMAN 2.0 is unchanged from LANMAN 1.0 except that the request may now be used to

specify a tree copy on the remote server. The tree copy mode is selected by setting bit 5 of the smb_flags word in the COPY request. When the tree copy option is selected the destination must not be an existing file and the source mode must be binary. A request with bit 5 of the smb_flags word set and either bit 0 or bit 3 set is therefore an error. When the tree copy mode is selected, the smb_cct word of the response protocol is undefined.

EXTENDED PROTOCOL

The format of enhanced and new commands is defined commencing at the smb_wct field. All messages will include the standard SMB header defined in section 1.0. When an error is encountered a server may choose to return only the header portion of the response (i.e., smb_wct and smb_bcc both contain zero).

Copy may generate the following errors:

Error Class ERRDOS

ERRbadfile

ERRbadpath

ERRfileexists

ERRnoaccess

ERRnofiles

ERRbadshare

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRnosupport

ERRaccess

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Lock and Read Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = *5*
WORD	smb_fid;	file handle
WORD	smb_count;	number of bytes to lock and return
DWORD	smb_offset;	offset in file to lock and begin read
WORD	smb_remcnt;	number of bytes remaining to be read
WORD	smb_bcc;	value = 0

Response Format (same as core READ):

Data type	Field	Value
BYTE	smb_wct;	value = *5*
WORD	smb_count;	number of locked bytes read
WORD	smb_rsvd[4];	reserved (to match size of write request)
WORD	smb_bcc;	minimum value = 4
BYTE	smb_ident1;	value = DATA-BLOCK
WORD	smb_size;	length of data returned
BYTE	smb_data[];	data

Service:

The LockandRead request is used to lock and "read ahead" the specified bytes.

The locked portion of a file is "safe" to read ahead because no other process can access the locked bytes until this process unlocks the bytes. Thus the consumer can assume that the bytes being locked will be read and submit this protocol to both lock and read ahead the bytes.

This can provide significant performance improvements on data base update operations (lock data -> read data -> [update -> write data] -> unlock data).

Whether or not this protocol is supported (along with WriteandUnlock) is returned in the smb_flg field of the negotiate response.

The request and response format are identical to the core read. The server merely locks the bytes before reading them.

If an error occurs on the lock, the bytes should not be read.

LockandRead may generate the following errors:

Error Class ERRDOS

ERRnoaccess

ERRbadfid

ERRlock

ERRbadaccess

Error Class ERRSRV

ERRerror

ERRinvdevice

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Move Command

Request Format:

Data type	Component	Value
BYTE	smb_wct;	value = 3
WORD	smb_tid2;	second (destination) path tid
WORD	smb_ofun;	what to do if destination file exists
WORD	smb_flags;	flags to control move operations
		bit 0 - destination must be a file.
		bit 1 - destination must be a directory
		bit 2 - reserved (must be zero)
		bit 3 - reserved (must be zero)
		bit 4 - verify all writes
WORD	smb_bcc;	minimum value = 2
BYTE	smb_path[];	pathname of source file
BYTE	smb_new_path[];	pathname of destination file

Response Format:

Data type	Component	Value
BYTE	smb_wct;	value = 1
WORD	smb_cct;	number of files moved
WORD	smb_bcc;	minimum value = 0
BYTE	smb_errfile[];	pathname of file where error occurred - ASCIIZ

Service

The file referenced by smb_path (source) is copied to smb_new_path (destination), then the file referenced by smb_path (source) is deleted. Both smb_path and smb_new_path must refer to paths on the server. The server must do any necessary access permission checks on both the source and the destination paths.

The TID in smb_tid of the header is associated with the source while smb_tid2 is associated with the destination. These TID fields may contain the same or differing valid TIDS. Note that smb_tid2 can be set to -1 indicating that this is to be the same TID as the TID in smb_tid of the header. This allows use of the move protocol with TCONandX.

The source path must refer to an existing file or files. Wild Cards are permitted.. Source files specified by wild cards are processed until an error is encountered. If an error is encountered, the expanded name of the file is returned in smb_errfile. The error code is returned in smb_err.

The destination path can refer to either a file or a directory.

The destination can be required to be a file or a directory by smb_flags bits. If neither bit is set, the destination may be either a file or a directory.

Wild cards are not permitted in the destination path:

smb_ofun bit field mapping:

bits:

1111	11		
5432	1098	7654	3210
rrrr	rrrr	rrrr	rrOO

where:

O - Open (action to be taken if destination file exists).

0 - Fail.

1 - Reserved

2 - Truncate file

r - reserved (must be zero)

If target file does not exist, it will be created. All file components except the last must exist (directories will not be created).

Move may generate the following errors.

Error Class ERRDOS

ERRbadfile

ERRbadpath

ERRfileexists

ERRnoaccess

ERRnofiles

ERRbadshare

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRnosupport

ERRaccess

<implementation specific>

Error Class ERRHRD

<implementation specific>

Read and X

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_res2;	reserved (must be zero)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
WORD	smb_fid;	file handle
DWORD	smb_offset;	offset in file to begin read
WORD	smb_maxcnt;	max number of bytes to return
DWORD	smb_mincnt;	min number of bytes to return
DWORD	smb_timeout;	number of milliseconds to wait for completion
WORD	smb_countleft;	bytes remaining to satisfy user's request
WORD	smb_bcc;	value = 0

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 12
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_res2;	reserved (ad to word)
WORD	smb_off2;	offset (from start of SMB hearder) to next command (@smb_wct)
WORD	smb_remaining;	bytes remaining to be read (pipes/devices only)
DWORD	smb_rsvd;	reserved
DWORD	smb_dsize;	number of data bytes (minimum value = 0)
DWORD	smb_doff;	offset (from start of SMB header) to data bytes
WORD	smb_rsvd;	reserved (These last 5 words are reserved to mae the ReadingandX response reserved the same size as the WriteandX request).
DWORD	smb_rsvd;	

DWORD	smb_rsvd;	
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (value of smb_dsize)

Service

The expanded Read and X command allows reads to be timed out and offers a generalized alternative to the core Read command.

The entire message sent and received including the optional second protocol must fit in the negotiated max transfer size.

The following are the only valid protocol requests commands for smb_com2 (X) for READ and X:

CLOSE

CLOSE and DISCONNECT

When the smb_timeout field is non-zero, it specifies the maximum milliseconds the server is to wait for a response to its read command. This feature is useful when accessing remote devices, such as terminals, where indeterminate delays are possible.

The Read command's scope is extended to Named Pipes and I/0 Devices. Timeout and mincnt values are normally expected to be used only with these devices. In the case of a named pipe or I/O device, timeout is defined to be the time to delay for at least smb_mincnt bytes.

If smb_timeout is zero (or the server does not support timeout) and no data is currently available, the server will send a response with the smb_dsize field set to zero.

If smb_timeout is non-zero and the server supports timeout, the server will wait to send the response until the data becomes available or a timeout occurs. If smb_timeout is greater than zero (but less than forever (-1)) and a timeout occurs, the server will send a response with the smb_err field set to indicate that the timeout occurred along with any bytes already read.

The return field smb_remaining is to be returned for pipes or devices only. It is used to return the number of bytes currently available in the pipe or device (NOT including the bytes returned in this buffer). This information can then be used by the consumer to know when a subsequent (non-blocking) read of the pipe or device may return some data. Note - that when the read request is actually received by the server there may be more or less actual data in the pipe or device (more data has been written to the pipe / device or another reader drained it). If the information is currently not available or the request is NOT for a pipe or device (or the server does not support this feature), a -1 value should be returned.

A -2 smb_timeout value indicates that the server should use the default timeout value associated with the pipe or device being read. Thus no timeout is explicitly set to the resource; rather the current timeout set either as a default or as a result of an IOCTL remains in effect.

Read and X may generate the following errors:

Error Class ERRDOS:

ERRnocaccess

ERRbadfid

ERRlock

ERRbadaccess

Error Class ERRSRV

ERRerror

ERRinvnid

ERRtimeout

<implementation specific>

Error Class ERRHRD:

<implementation specific>

The Read Block Multiplexed Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 08
WORD	smb_fid;	file handle
DWORD	smb_offset;	offset in file to begin read
WORD	smb_maxcnt;	max number of bytes to return (max 65,535)
WORD	smb_mincnt;	min number of bytes to return (normally 0)
DWORD	smb_timeout;	number of milliseconds to wait for completion
WORD	smb_rsvd;	reserved
WORD	smb_bcc;	value = 0

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 8
DWORD	smb_offset;	offset in file where data read
WORD	smb_tcount;	total bytes being returned this protocol
WORD	smb_remaining;	bytes remaining to be read (pipes/devices only)
DWORD	smb_rsvd;	reserved
WORD	smb_dsize;	number of data bytes this buffer (num value = 0)
WORD	smb_doff;	offset (from start of SMB header) to data bytes
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes value of smb_dsize

Service

The Read Block Multiplexed protocol is used to maximize the performance of reading a large block of data from the server to the consumer on a multiplexed VC.

The Read Block Multiplexed command's scope includes (but is not limited to) files, Named Pipes and communication devices.

When this protocol is used, other requests may be active on the multiplexed VC. The server will respond with the one or more

response protocol message as defined above until the requested data amount has been returned. Each response contains the smb_pid and smb_mid of the Read Block Multiplexed request, the file offset and data length defined in the Read response protocol (including the SMB header). This allows the consumer's message delivery (multiplexing) system to deliver the response to the appropriate thread.

The Consumer knows the maximum amount of data bytes which the server may return (from smb_maxcnt of the request). Thus it initializes its bytes expected variable to this value. The Server then informs the consumer of the actual amount being returned via each "packet" (buffer) of the response (smb_tcount).

The server may reduce the expected bytes by lowering the total number of bytes expected (smb_tcount) in each (any) response.

Thus, when the amount of data bytes received (total of each smb_dsize) equals the total amount of data bytes expected (smallest smb_tcount received), then the consumer has received all the data bytes. This allows the protocol to work even if the "packets" (buffers) are received out of sequence.

Note that the buffer size being returned here can not be larger than the smaller of the consumer's buffer size (as specified in smb_bufsize on the SESSION SETUP and X request protocol) or the server's buffer size (as specified in smb_maxxmt of the NEGOTIATE response protocol).

As is true in the core read protocol, (while reading a "standard blocked disk file"), the total number of bytes returned may be less than the number requested only if a read specifies bytes beyond the current file size. In this case only the bytes that exist are returned. A read completely beyond the end of file will result in a single response with a zero value in smb_rcount. If the total number of bytes returned is less than the number of bytes requested, this indicates end of file (if reading other than a standard blocked disk file, only ZERO bytes returned indicates end of file).

This protocol eliminates nearly half the protocols involved with reading a block of data since the Read Block Multiplexed request is sent only once as opposed to one for each negotiated buffer size as defined with the Read protocol.

The transport layer guarantees delivery of all responses to the consumer. Thus no "got the data you sent" protocol is needed. If an error should occur at the consumer end, all bytes must be received and thrown away. There is no need to inform the server of the error.

Once started, the Read Block Multiplexed operation is expected to go to completion. The consumer is expected to receive all the responses generated by the server. Conflicting commands (such as file close) must not be sent to the server while a multiplexed operation is in progress.

The flow for the Read Block Multiplexed (R.B.M.) protocol is:

consumer---->	R.B.M. request>---	>----------server
consumer<----<	R.B.M. response 1 with data	<----------server
consumer<----<	R.B.M. response 2 with data	<----------server
consumer<----<	R.B.M. response n with data	<----------server

Note that the request through the final response make up the complete protocol, thus the TID, PID, UID and MID are expected to remain constant and can be used by the consumer to route the individual messages of the protocol to the correct process.

The return field smb_remaining is to be returned for pipes or devices only. It is used to return the number of bytes currently available in the pipe or device (NOT including the bytes returned with this protocol). This information can then be used by the consumer to know when a subsequent (non blocking) read of the pipe or device may return some data. Note - that when the read request is actually received by the server there may be more or less actual data in the pipe or device (more data has been written to the pipe / device or another reader drained it). If the information is currently not available or the request is NOT for a pipe or device (or the server does not support this feature), a -1 value should be returned.

Read Block Multiplexed may generate the following errors. Note that the error ERRnoresource (or ERRusestd) may be returned by the server if it is temporarily out of large buffers. The consumer could then retry using the standard "core" read request, or delay and retry the read block multiplexed request.

Error Class ERRDOS

ERRnoaccess

ERRbadfid

ERRIock

ERRbadaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRnoresource

ERRusestd

ERRtimeout

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Read Block Raw Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 08
WORD	smb_fid;	file handle
DWORD	smb_offset;	offset in file to begin read
WORD	smb_maxcnt;	max number of bytes to return (max 65,535)
WORD	smb_mincnt;	number of bytes to return (usually 0)
DWORD	smb_timeout;	number of milliseconds to wait for completion
WORD	smb_rsvd;	reserved
WORD	smb_bcc;	value = 0

Response is the raw data (one send).

Service

The Read Block Raw protocol is used to maximize the performance of reading a large block of data from the server to the consumer.

The Read Block Raw command's scope includes (but is not limited to) files, Named Pipes and communication devices.

When this protocol is used, the consumer has guaranteed that there is (and will be) no other request on the VC for the duration of the Read Block Raw request. The server will respond with the raw data being read (one send). Thus the consumer is able to request up to 65,535 bytes of data and receive it directly into the user buffer. Note that the amount of data requested is expected to be larger than the negotiated buffer size for this protocol.

The reason that no other requests can be active on the VC for the duration of the request is that if other receives are present on the VC, there is normally no way to guarantee that the data will be received into the user space, rather the data may fill one (or more) of the other buffers.

The number of bytes actually returned is determined by the length of the message the consumer receives as reported by the transport layer (there are no overhead "header bytes").

If the request is to read more bytes than are present in the file, the read response will be of the length actually read from the file.

If none of the requested bytes exist (EOF) or an error occurs on the read, the server will respond with a zero byte send. Upon receipt of a zero length response, the consumer will send a "standard read" request to the server. The response to that read will then tell the consumer that EOF was hit or identify the error condition.

As is true in the core read protocol, (while reading a "standard blocked disk file"), the number of bytes returned may be less than the number requested only if a read specifies bytes beyond the current file size. In this case only the bytes that exist are returned. A read completely beyond the end of file will result in a response of zero length. If the number of bytes returned is less than the number of bytes requested, this indicates end of file (if reading other than a standard blocked disk file, only ZERO bytes returned indicates end of file).

The transport layer guarantees delivery of all response bytes to the consumer. Thus no "got the data you sent" protocol is

needed.

If an error should occur at the consumer end, all bytes must be received and thrown away. There is no need to inform the server of the error.

Support of this protocol is optional.

Whether or not Read Block Raw is supported is returned in the response to negotiate and in the LAN MAN 1.0 extended "Query Server Information" protocol.

The flow for reading a sequential file (or down-loading a program) using the Block Read Raw protocol is:

consumer------->	OPEN for read request	---------->server
consumer<------	<Open succeeded response	<----------server
consumer------->	BLOCK READ RAW request 1	>--------->server
consumer<-------	<raw data returned	<----------server
consumer------->	BLOCK READ RAW request 2	>--------->server
consumer<-------	<raw data returned	<----------server
consumer------->	BLOCK READ RAW request n	>--------->server
consumer<------	<ZERO LEN SEND (EOF or ERROR)	<----------server
consumer------->	"standard READ request>	----------server
consumer<-----<	READ response EOF/ERROR	<---------server
consumer------->	CLOSE request>----------	>-------->server

This approach minimizes the number of overhead protocols (and bytes) required.

Read Block Raw may generate NO errors. Because the response to this protocol is raw data only, a zero length response indicates EOF, a read error or that the server is temporarily out of large buffers. The consumer should then retry using a Multiplexed Read Request or a standard "core" read request. This request will then either return the EOF condition, an error if the read is still failing, or will work if the problem was due to being temporarily out of large buffers.

The Locking and X Command

The LockingX request SMB in the LM 2.1 dialect modified the meaning modified the meaning of previous smb_locktype field to take advantage of new locking features provided in various operating systems. This LockingX request SMB is detailed below (* indicates changes from earlier protocol).

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 8
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_reh2;	reserved (must be zero)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb-wct)
WORD	smb_fid;	file handle
*WORD	smb_lockflags;	locking mode: bit 0 - 0 = lock out all access, 1 = read OK while locked bit 1 - 1 = single user total file unlock (OpLock Break) bit 2 = 1, Requesting change lock type on supplied smb_lockrng[] bit 3 = 1, Requesting cancel of lock specified in smb_unlkrng[]
DWORD	smb_timeout;	number of milliseconds to attempt each lock
WORD	smb_unlocknum;	number of unlock range structures following
WORD	smb_locknum;	number of lock range structures following

WORD	smb_bcc;	total bytes following */
struct	smb_unlkrng[*];	unlock range structures
struct	smb_lockrng[*];	lock range structures (* = smb_locknum)

Unlock Range Structure (smb_unffimg) Format:

Data type	Field	Value
WORD	smb_lpid;	pid of process "owning" the lock
DWORD	smb_unlockoff;	file offset to bytes to be unlocked
DWORD	smb_unlocklen;	number of bytes to be unlocked

Lock Range Structure (smb_lockrng) Format:

Data type	Field	Value
WORD	smb_lpid;	pid of process owning the lock
DWORD	smb_lockoff;	file offest to bytes to be locked
DWORD	smb_locklen;	number of bytes to be locked

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 2
BYTE	smb_com2;	secondary (X) command, OxFF = none
BYTE	smb_res2;	reserved (pad to word)
WORD	smb_off2;	offset (from SMB header start) to next cmd (@smb_wct)
DWORD	smb_bcc;	value = 0

Service Enhancement:

This protocol allows both locking and/or unlocking of file range(s).

If unlocking is specified (smb_unlocknum non-zero), the number of bytes specified by smb_unlocklen at the file offset specified by smb_unlockoff will be unlocked for each unlock range. Then if locking is specified (smb_locknum non-zero), the number of bytes specified by smb-locklen at the file offset specified by smb_lockoff will be locked for each lock range.

The time specified by smb_timeout is the maximum amount of time to wait for the byte range(s) specified to become unlocked (so that they can be locked by this protocol). A timeout value of 0 indicates that this protocol should fail immediately if any lock range specified is locked. A timeout value of -1 indicates that the server should wait as long as it takes for each byte range specified to become unlocked so that it may be again locked by this protocol. Any other value of smb_timeout specifies the maximum number of milliseconds to wait for all lock range(s) specified to become available.

If any of the lock ranges timeout because the area to be locked is already locked (or the lock fails), the other ranges in the protocol request which were successfully locked as a result of this protocol will be unlocked (either all requested ranges will be locked when this protocol returns to the consumer or none).

If smb_locktype is 1, the lock is specified as a "read only" lock. Locks for both read and write (where smb_locktype is 0) should be prohibited, but other "read only" locks should be permitted. If this mode can not be supported on a given server, the smb_locktype field should always be treated as 0 in that any lock attempt will fail if the byte range specified is locked.

Closing a file with locks still in force causes the locks to be released in no defined order.

Locking is a simple mechanism for excluding other processes read/write access to regions of a file. The locked regions can be anywhere in the logical file. Locking beyond end-of-file is NOT an error. Any process using the FID specified in smb_fid has access to the locked bytes; other processes will be denied the locking of the same bytes.

The proper method for using locks is not to rely on being denied read or write access on any of the read/write protocols but rather to attempt the locking protocol and proceed with the read/write only if the lock succeeded.

Locking a range of bytes will fail if any subranges or overlapping ranges are locked. In other words, if any of the specified bytes are already locked, the lock will fail.

The time which a byte range is locked should be kept as short as possible.

The entire message sent and received including the optional second protocol must fit in the negotiated max transfer size.

NOTE - LANMAN 1.0 does not support smb_locktype where bit 1 is set (read ok while locked); also smb_timeout is ignored and treated as if it were set to zero.

The following are the only valid protocol request commands for smb_com2 (X) for LOCKING and X:

READ

READ and X

A "single user total file lock" is also known as an "opportunistic lock". A consumer requests an "opportunistic lock" by setting the appropriate bit in the OpenX, Open, Create and MakeNew protocols whenever the file is being opened in a mode which is not exclusive. The server responds by setting the appropriate bit in the response protocol indicating whether or not the "opportunistic lock" was granted. By granting the "oplock", the server tells the consumer that the file is currently ONLY being used by this one consumer process at the current time. The consumer can therefore safely do read ahead and write behind as well as local caching of file locks knowing that the file will not be accessed/changed in any way by another process while the "oplock" is in effect.

The consumer will be notified when any other process attempts to open the "oplocked" file and if "opbatch" (bit 2 of smb_flags) was set on the OpenX request the consumer will also be notified on any operation which may change the file.

When another user attempts to Open (or otherwise modify if "opbatch" was requested) the file which a consumer has oplocked, the server will "hold off" the 2nd attempt and notify the consumer via a LockingX protocol (with bit one of smb_locktype set) that the "oplock" is being broken. The consumer is expected to then flush any dirty buffers, submit any file locks (LockingX protocol can be used for this) and respond to the server with either a LockingX protocol (with bit one of smb_locktype set) or with a close protocol if the file is no longer in use. Note that because of a close being sent to the server and break oplock notification from the server could cross on the wire, if the consumer gets an oplock notification on a file which it does not have open, that notification should be ignored. Once the "oplock" has been broken, the consumer must no longer do any form of data or lock caching. The " oplock" is never reinstated while the file is open. If the file is still open once the consumer has been notified, the second opener does not get the file "oplocked" along with the open. If the file is closed by the consumer which had it open, the server is again free to grant the new opener the oplock.

Note that the "oplock" broken notification will only go to one consumer because after the oplock is broken, any further open attempts will just get the oplock request denied.

Also note that due to timing, the consumer could get an "oplock" broken notification in a user's data buffer as a result of this notification crossing on the wire with a Read Raw request. The consumer must detect this (use length of msg, "FFSMB", MID of

-1 and smb_cmd of SMBLockingX) and honor the "oplock" broken notification as usual. The server must also note on receipt of an Read Raw request that there is an outstanding (unanswered) "oplock" broken notification to the consumer and return a zero length response denoting failure of the read raw request. The consumer should (after responding to the "oplock" broken notification), use a standard read protocol to redo the read request. This allows a file to actually contain data matching an "oplock" broken notification and still be read correctly.

"Oplock" is a major performance win in the real world because many files must be opened in a non exclusive mode because the file could be used by others. However often, the files are not actually in use by multiple users at the same instant.

Locking and X may generate the following errors.

Error Class ERRDOS

ERRbadfile

ERRbadfid

ERRlock

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRnoresource

<implementation specific>

Error Class ERRHRD

<implementation specific>

The IOCTL Command

Primary Request Format

Data type	Field	Value
BYTE	smb_wct;	value = 14
WORD	smb_fid;	file handle
WORD	smb_cat;	device category
WORD	smb_fun;	device function
WORD	smb_tpscnt;	total number of paramter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	max number of parameter bytes to return
WORD	smb_mdrcnt;	max number of data bytes to return
DWORD	smb_timeout;	number of milliseconds to wait fore completion
WORD	smb_rsvd;	reserved
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB header) to parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_bcc;	total bytes (including pad bytes) following
WORD	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_pscnt)
BYTE	smb_pad1[];	(optional to pad to word or dword boundary
BYTE	smb_data[*];	data bytes

Interim Response Format (if no error - ok send remaining data)

Data type	Field	Value
BYTE	smb_wct;	value = 0
WORD	smb_bcc;	value = 0

Secondary Request Format (more data - may be zero or more of these)

Data type	Field	Value
BYTE	smb_wct;	value = 8
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB hdr) to paramter bytes
WORD	smb_psdisp;	byte displacement for these paramter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_pscnt)
BYTE	smb_pad1p[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Response Format (may respond with one or more of these)

Data type	Component	Value
BYTE	smb_wct;	value = 8
WORD	smb_tprcnt;	total number of parameter bytes being returned
WORD	smb_tdrcnt;	total number of data bytes being returned
WORD	smb_prcnt;	number of param bytes being returned this buffer
WORD	smb_proff;	offset (from start of SMB hdr) to paramter bytes
WORD	smb_prdisp;	byte displacement for these parameter bytes
WORD	smb_drcnt;	number of data bytes being returned this buffer
WORD	smb_droff;	offset (from start of SMB

		hdr) to data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_prcnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary

This function delivers a device/file specific request to a server, and the device/file specific response to the requester. The target file is identified by the file handle in smb_fid.

The request defines a function specific to a particular device type on a particular server type. Therefore the functions supported are not defined by the protocol, but by consumer/server implementations. The protocol simply provides a means of delivering them and retrieving the results.

Note that the primary request through the final response make up the complete protocol, thus the TID, PID, UID, and MID are expected to remain constant and can be used by both the server and consumer to route the individual messages of the protocol to the correct process.

The number of bytes needed in order to perform the IOCTL request may be more than will fit in a single buffer.

At the time of the request, the consumer knows the number of parameter and data bytes expected to be sent and passes this information to the server via the primary request (smb_tpscnt and smb_tdscnt). This may be reduced by lowering the total number of bytes expected (smb_tpscnt and/or smb_tdscnt) in each (any) secondary request.

Thus when the amount of parameter bytes received (total of each smb_pscnt) equals the total amount of parameter bytes expected (smallest smb_tpscnt) received, then the server has received all the parameter byes.

Likewise, when the amount of data bytes received (total of each smb_dscnt) equals the total amount of data bytes expected (smallest smb_tdscnt) received, then the server has received all the data bytes.

The parameter bytes should normally be sent first followed by the data bytes. However, the server knows where each begins and ends in each buffer by the offset fields (smb_psoff and smb_dsoff) and the length fields (smb_pscnt and smb_dscnt). The displacement of the bytes (relative to start of each) is also known (smb_psdisp and smb_dsdisp). Thus the server is able to reassemble the parameter and data bytes should the "packets" (buffers) be received out of sequence.

If all parameter bytes and data bytes fit into a single buffer, then no interim response is expected (and no secondary request is sent).

The Consumer knows the maximum amount of data bytes and parameter bytes which the server may return (from smb_mprcnt and smb_mdrcnt of the request). Thus it initializes its bytes expected variables to these values. The server then informs the consumer of the actual amounts being returned via each "Packet" (buffer) of the response (smb_tprcnt and smb_tdrcnt).

The server may reduce the expected bytes by lowering the total number of bytes expected (smb_tprcnt and/or smb_tdrcnt) in each (any) response.

Thus when the amount of parameter bytes received (total of each smb_prcnt) equals the total amount of parameter bytes expected (smallest smb_tprcnt) received, then the consumer has received all the parameter bytes.

Likewise, when the amount of data bytes received (total of each smb_drcnt) equals the total amount of data bytes expected (smallest smb_tdrcnt) received, then the consumer has received all the data bytes.

The parameter bytes should normally be returned first followed by the data bytes. However, the consumer knows where each begins and ends in each buffer by the offset fields (smb_psoff and smb_dsoff) and the length fields (smb_prcnt and smb_drcnt). The displacement of the bytes (relative to the start of each) is also known (smb_prdisp and smb_dsdisp). Thus the consumer is able to reassemble the parameter and data bytes should the "packets" (buffers) be received out of sequence.

In the simplest form, a single request is sent and a single response is returned.

Thus the flow is:

1. The consumer sends the first (primary) request which identifies the total bytes (both parameters and data) which are expected to be sent and contains as many of those bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (both parameters and data) the server will need to return an IOCTL completion. If all the bytes fit in the single buffer, skip to step 4.
2. The server responds with a single interim response meaning "OK", send the remainder of the bytes".
3. The consumer then sends another buffer full of bytes to the server. On each iteration of this secondary request, smb_tpscnt and/or smb_tdscnt could be reduced. This step is repeated until all bytes have been delivered to the server (total of all smb_pscnt equals smallest smb_tpscnt and total of all smb_dscnt equals smallest smb_tdscnt).
4. The Server sets up and performs the IOCTL with the information provided.
5. Upon completion of the IOCTL, the server sends back (up to) the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all result bytes have been returned. On each iteration of this response, smb_tprcnt and/or smb_tdrcnt could be reduced. This step is repeated until all bytes have been delivered to the consumer (total of all smb_prcnt equals smallest smb_tprcnt and total of all smb_drcnt equals smallest smb_tdrcnt).

The flow for the IOCTL protocol when the request parameters and data does NOT all fit in a single buffer is:

consumer---->	IOCTL request (data)	>---------server
consumer<-----	<OK send remaining data<	<---------server
consumer------>	IOCTL secondary request 1 (data)	>-------->server
consumer------->	IOCTL secondary request 2 (data)	>-------->server

The flow for the IOCTL protocol when the request parameters and data does all fit in a single buffer is:

consumer----->	IOCTL request (data)	>------->server
	(server sets up and performs the IOCTL)	
consumer<------	<IOCTL response 1 (data)	<--------server
	(server sets up and performs the IOCTL)	
consumer<-----	<IOCTL response 2 (data)	<---------server
consumer<------	<IOCTL response n (data)	<---------server

The first release of LANMAN 1.0 will support only the most simple form of the IOCTL protocol. Only a single request and a single response is expected. Further the maximum number of parameter bytes is limited to 128 bytes and the maximum number of data bytes is limited to 128 bytes on both the request and response IOCTL protocols. This ensures that the request and response will fit within the minimum 1024 byte SMB buffers.

The flow for the IOCTL protocol when the request parameters and data does all fit in a single buffer is and the reply parameters and data all fit in a single buffer is:

consumer------->	IOCTL request (data)	>---------->server
	(server sets up and performs the IOCTL)	

consumer<------- <IOCTL response (data) <----------server

IOCTL may generate the following errors.

Error Class ERRDOS

ERRbadfile

ERRbadfid

ERRbaddata

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Get Expanded File Attributes Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_fid;	file handle
WORD	smb_bcc;	value = 0

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 11
WORD	smb_cdate;	date of creation
WORD	smb_ctime;	time of creation
WORD	smb_adate;	date of last access
WORD	smb_atime;	time of last access
WORD	smb_mdate;	date of last modificafion
WORD	smb_mtime;	time of last modification
DWORD	smb_datasize;	file end of data
DWORD	smb_allocsize;	file allocation
WORD	smb_attr;	file attribute
WORD	smb_bcc;	minimum value 0
BYTE	smb_rsvd[];	reserved

Service Enhancement

The Expanded Get File Attributes is enhanced to return more information about the queried file. The current values of the file attributes listed as output parameters are returned to the requester. If a server does not support one of the optional attributes, a null value (hex FFFF) is returned.

The file being interrogated is specified by the file handle (FID).

The values of the response fields which are for information not requested (via smb-info of the request) are undefined.

The attribute field (smb_attr) has the following format (bit 0 is the least significant bit). This field matches that used by OS/2.

bit0 - read only file

bit1 - "hidden" file

bit2 - system file

bit3 - reserved

bit4 - directory

bit5 - archive file

bits 6 - 15 - reserved (must be zero)

Note that the volume label bit (as defined in the core protocol) is no longer a valid attribute. The volume label is now retumed in the Query Server Information response.

The contents of response parameters is not guaranteed in the case of an error return (any protocol response with an error set in the header may have smb_wct of zero and smb_bcc count of zero).

The dates are in the following format:

bits:

1111	11		
5432	1098	7654	3210
yyyy	yyym	mmmd	dddd

where:

y- bit of year 0-119 (1980-2099)

m-bit of month 1-12

d- bit of day 1-31

The time is in the following format:

bits:

1111	11		
5432	1098	7654	3210
hhhh	hmmm	mmmx	xxxx

where:

h- bit of hour (0-23)

m- bit of minute (0-59)

x- bit of 2 second increment

Get Extended File Attributes may generate the following errors.

Error Class ERRDOS

ERRbadfile

ERRbadfid

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Find Unique Command

Request Format: (same as core Search Protocol - "Find First" form)

BYTE	smb_wct;	value = 2
WORD	smb_count;	max number of entries to find
WORD	smb_attr;	search attribute
WORD	smb_bcc;	minimum value = 5
BYTE	smb_ident1;	ASCII (04)
BYTE	smb_pathname[];	filename (may contain global variables)
BYTE	smb_ident2;	Variable block (05)
WORD	smb_keylen;	must be zero ("Find First only)

Response Format: (same as core Search Protocol)

BYTE	smb_wct;	value = 1
WORD	smb_count;	number of entries found
WORD	smb_bcc;	minimum value = 3
BYTE	smb_ident;	Variable Block (05)
WORD	smb_datalen;	data length
BYTE	smb_data[*];	directory entries

Directory Information Entry (dir_info) Format- (same as core Search Protocol)

BYTE	find_buf_reserved [21];	reserved (resume_key)
BYTE	find_buf_attr;	attribute
WORD	find_buf_time;	modification time (hhhhh mmmmmm xxxxx) where x is in two second increments.
WORD	find_buf_date;	modification date (yyyyyy mmmm ddddd)
DWORD	find_buf_size;	file size
BYTE	find_buf_pname[13];	file name -- ASCII (null-terminated)

The resume key (or close key) has the following format:

BYTE	sr_res;	reserved
		bit 7 - reserved for consumer use
		bit 5, 6 - reserved for system use (must be preserved)
		bits 0-4 - reserved for server (must be preserved by consumer)
BYTE	sr_name [1];	pathname sought. Format:
		1-8 character file name, left justified
		0-3 character extension, left justified (in last 3 chars)
BYTE	sr_findid[1];	uniquely identifies find through find close
BYTE	sr_server[4];	available for server use (must be non-zero)
BYTE	sr_res[4];	reserved for consumer use

Service

The Find Unique protocol finds the directory entry or group of entries matching the specified file path name. The filename portion of the pathname may contain global (wild card) characters, but the search may not be resumed and no Find Close protocol is expected.

The protocols "Find", "Find Unique" and "Find Close" are methods of reading (or searching) a directory. These protocols may be used in place of the core "Search" protocol when LANMAN 1.0 dialect has been negotiated. There may be cases where the Search protocol will still be used.

The format of the Find Unique protocol is the same as the core "Search" protocol. The difference is that the directory is logically opened, any matching entries returned, and then the directory is logically closed.

This allows the Server to make better use of its resources. No Search buffers are held (search resumption via presenting a "resume_key" will not be allowed).

(Only one buffer of entries is expected and find close need not be sent).

The file path name in the request specifies the file to be sought. The attribute field indicates the attributes that the file must have. If the attribute is zero, then only normal files are returned. If the system file, hidden or directory attributes are specified then the search is inclusive -- both the specified type(s) of files and normal files are returned. If the volume label attribute is specified then the search is exclusive, and only the volume label entry is returned

The max_count field specifies the number of directory entries to be returned. The response will contain zero or more directory entries as determined by the count_returned field. No more than max_count entries will be returned. Only entries that match the sought filename/attribute will be returned.

The resume-key field must be null (length = 0).

A Find Unique request will terminate when either the requested maximum number of entries that match the named file are found, or the end of directory is reached without the maximum number of matches being found. A response containing no entries indicates that no matching entries were found between the starting point of the search and the end of directory.

There may be multiple matching entries in response to a single request as Find Unique supports "wild cards" in the file name (last component of the pathname). "?" is the wild card for single characters, "*" or "null" will match any number of filename characters within a single part of the filename component. The filename is divided into two parts -- an eight character name and a three character extension. The name and extension are divided by a ".".

If a filename part commences with one or more "?"s, then exactly that number of characters will be matched by the Wild Cards, e.g., "??x" will equal "abx" but not "abcx" or "ax". When a filename part has trailing "?"s then it will match the specified number of

characters or less, e.g., "x??" will match "xab", "xa" and "x", but not "xabc". If only "?"s are present in the filename part, then it is handled as for trailing "?"s

"*" or "null" match entire pathname parts, thus "*.abc" or ".abc" will match any file with an extension of "abc". "*" or "null" will match all files in a directory.

Unprotected servers require the requester to have read permission on the subtree containing the directory searched (the share specifies read permission).

Protected servers require the requester to have permission to search the specified directory.

If a Find Unique requests more data than can be placed in a message of the max_xmit_size for the TD specified, the server will abort the virtual circuit to the consumer.

The number of entries returned will be the minimum of:

- The number of entries requested.
- The number of (complete) entries that will fit in the negotiated SMB buffer.
- The number of entries that match the requested name pattern and attributes.

The error ERRnofiles set in smb_err field of the response header or a zero value in smb_count of the response indicates no matching entry was found.

The resume search key returned along with each directory entry is a server defined key. This key will be returned as in the Find protocol and Search protocol; however it may NOT be returned to continue the search.

The date is in the following format:

bits:

1111	11		
5432	1098	7654	3210
yyyy	yyym	mmmd	dddd

where:

y- bit of year 0-119 (1980-2099)

m-bit of month 1-12

d- bit of day 1-31

The time is in the following format:

bits:

1111	11		
5432	1098	7654	3210
hhhh	hmmm	mmmx	xxxx

where:

h - bit of hour (0-23)

m - bit of minute (0-59)

x - bit of 2 second increment

Find Unique may generate the following errors:

Error Class ERRDOS

ERRnofiles

ERRbadpath

ERRnoaccess

ERRbadaccess

ERRbadshare

<implementation specific>

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Echo Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 01
WORD	smb_reverb;	number of times to echo data back
WORD	smb_bcc;	minimum value = 4
BYTE	smb_data[];	data to echo back

Response Format (one for each echo requested):

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_seq;	sequence number of this echo
WORD	smb_bcc;	minimum value = 4
BYTE	smb_data[];	echo data

Service:

The ECHO protocol is used to test the VC and to see if the server is still responding. It is also used to see if the TID is still valid.

When this protocol is used, other requests may be active on the multiplexed VC. The server will respond with the zero or more response protocol messages as requested in smb_reverb.

Each response echoes data sent (note - data size may be zero). If smb_reverb is zero, no response will be sent.

There is no need for a valid TID field in smb_tid for the success of this protocol (a null TID is defined as OXFFFF). However, if a TID is present, then the server must check the TID for validity because the consumer may be sending this protocol to see if the TID is still valid. ERRSRV error class and ERRinvnid error code should be set in the protocol response if the TID is invalid.

The flow for the ECHO protocol is:

consumer ---->ECHO request (data) >------>server

consumer ---->ECHO response 1 (data) >---->server

consumer ---->ECHO response 2 (data) >--->server

consumer ---->ECHO response n (data) >-->server

ECHO may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRnosupport

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Tree Connect and X Command

Request Format

Data type	Field	Value
BYTE	smb_wct;	value = 4
+BYTE	smb_com2;	secondary (X) command, OXFF = none
+BYTE	smb_reh2;	reserved (must be zero)
+WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
+WORD	smb_flags;	additional information bit 0 - if set, disconnect TID in current smb_tid
+WORD	smb_spasslen;	length of smb-spasswd
WORD	smb_bcc;	minimum value = 3
BYTE	smb-spasswd[*];	net-name password (* = smb_spasslen value)
BYTE	smb_path[];	server name and net-name
BYTE	smb-dev[];	service name string

+ Additional parameters (compared with core TREE CONNECT protocol)

The LM 2.1 response to the TreeConnX SMB returns 2 fields that the LM 1.1 implementation does not contain. With the addition of some features in various operating systems, it has become necessary to differentiate which of these features can be used by consumers on a connection basis. The exclusive search feature of OS/2 1.3 is an example of such features. The smb_optsupp field now returned by the server can be used to determine what, if any, of these features are available. In addition to this field, the server can be used to determine was added to help the consumer determine the type of file system the consumer is connecting. The smb_optsupp field is a word of bit masks and the smb_nativefs field is a null-terminated string. Their orientation is the smb indicated below (* indicates new field).

Response Format

Data type	Field	Value
BYTE	smb_wct;	value = 2
+BYTE	smb_com2;	secondary (X) command, OxFF = None

+BYTE	smb_res2;	reserved (pad to word)
+WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb-wct)
WORD	smb_optsupp;	bitmask indicating advanced OS features available; bit 0 = 1, exclusive search bits supported
WORD	smb_bcc;	minimum value = 3;
BYTE	smb_nativefs[];	native file system for this connection.

Servers that support the search bits defined below will identify themselves by setting bit0 of smb_oppsupp to 1. Note that this allows the server to optionally support these features on a per-connection basis.

For servers that support the search bits, the search bits will be passed along in the smb_attr field of the Find, Find Unique, Search, Trans2, and FindFirst SMBs. All these SMBs will interpret the new bits of the smb_attr field as follows. New bits are 8-13:

Bit	Meaning
13	If set, only files marked as archived are included
12	If set, only directories are included
11	No meaning
10	If set, only files marked as system are included
9	If set, only files marked as hidden are included
8	If set, only files marked as read-only are included

Tree Connect and X may generate the following errors:

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror

ERRbadpw

ERRinvnetname

<implementation specific>

Error Class ERRHRD

<irnplementation specific>

Transact Command

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = (14 + value of smb_suwcnt)
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	max number of parameter bytes to return
WORD	smb_mdrcnt;	max number of data bytes to retum
BYTE	smb_msrcnt;	max number of setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information
		bit 0 - if set, also disconnect TID in smb_tid
		bit 1 - if set, transaction is one way (no final response)
DWORD	smb_timeout;	number of milliseconds to wait for completion
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	set up word count

BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup[*];	variable number of set up words
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_name[1];	Must be a null byte
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_pscnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Interim Response Format (if no error - ok send remaining data):

Data type	Field	Value
BYTE	smb_wct;	value = 0
WORD	smb_bcc;	value = 0

Secondary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 9
BYTE	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_psdisp;	byte displacement for these parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to parameter bytes

WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	file id for handle based requests, else 0xfff
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word to dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_pscnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (*= value of smb_dscnt)

Response Format (may respond with zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 10 + value of smb_suwcnt
WORD	smb_tprcnt;	total number of parameter bytes being returned
WORD	smb_tdrcnt;	total number of data bytes being returned
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	number of parameter bytes being returned this buf
WORD	dmb_droff;	offset (from start of SMB hdr) to data types
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	setup return word count
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_setup[*];	variable # of set up return words (* = smb_suwcnt)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword

		boundary
BYTE	smb_param[*];	param bytes (* = value of smb_prcnt)
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_drcnt)

Service:

The Transaction2 protocol allows transfer of parameter and data blocks greater than a negotiated buffer size between the requester and the server.

The Transaction2 command scope includes (but is not limited to) IOCTL device requests and file system requests which require the transfer of an extended attribute list.

The Transaction2 protocol is used to ransfer a request for any of a set of supported functions on the server which may require the transfer of large data blocks. The function requested is identified by the first workd in the transaction2 smb_setup field. Other function specific information may follow the function identifier in the smb_setup file id or in the smb_param field. The functions supported are not defined by the protocol, but by consumer/server implementations. The protocol simply provides a means of delivering them and retrieving the results.

The number of bytes needed in order to perform the TRANSACT2 request may be more than will fit in a single buffer.

At the time of the request, the consumer knows the number of parameter and data bytes expected to be sent and passes this information to the server via the primary request (smb_tpscnt and smb_tdscnt). This may be reduced by lowering the total number of bytes expected (smb_tpscnt and/or smb_tdscnt) in each (any) secondary request.

Thus when the amount of parameter bytes received (total of each smb_dscnt) equals the total amount of parameter bytes expected (smallest smb_tpscnt) received, then the server has received all the parameter bytes.

Likewise, when the amound of data bytes received (total of each smb_dscnt) equals the total amount of data bytes expected (smallest smb_tdscnt) received, then the server has received all the data bytes.

The parameter bytes should normally be sent first followed by the data bytes. However, the server knows where each begins and ends in each buffer by the offset fields (smb_psoff and smb_dsoff) and the length fields (smb_pscnt and smb_dsnct). The displacement of bytes (relative to start of each) is also known (smb_psdisp and smb_dsdisp). Thus the server is able to reassemble the parameter and data bytes should the "packets" (buffers) be received out of sequence.

Thus the flow is:

1. The consumer send the first (primary) request which identifies the total bytes (both parameters and data) which are expected to be sent and contains the set up words and as many of the parameter and data bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters and data) the server is to return on TRANSACT2 completion. If all the bytes fit in the single buffer, skip to step 4.
2. The server responds with a single interim response meaning "ok, send the remainder of the bytes" or (if error response) terminate the transaction.
3. The consumer then sends another buffer full of bytes to the server. On each iteration of this secondary request, smb_tpscnt and/or smb_tdscnt could be reduced. This step is repeated until all bytes have been delivered to the server (total of all smb_pscnt equals smallest smb_tpscnt and total of all smb_dscnt equals smallest smb_tdscnt).
4. The Server sets up and performs the TRANSACT2 with the information provided.
5. Upon completion of the TRANSACT2, the server sends back (up to) the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all result bytes have been returned. On each iteration of this response, smb_tprcnt and/or smb_tdrcnt could be reduced. This step is repeated until all bytes have been delivered to the consumer (total of all smb_prcnt equals smallest smb_tprcnt and total of all smb_drcnt equals smallest smb_tdrcnt).

Thus the flow is:

1. The consumer sends the first (primary) request which identifies the total bytes (parameters and data) which are to be sent, contains the set up words and as many of the parameter and data bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters, and data) the server is to return on TRANSACT2 completion. The parameter bytes are immediately followed by the data bytes (the length fields identify the break point). If all the bytes fit in the single buffer, skip to step 4.
2. The server responds with a single interim response meaning "ok, send the remainder of the bytes" or (if error response) terminate the transaction.
3. The consumer then sends another buffer full of bytes to the server. This step is repeated until all bytes have been delivered to the server.
4. The Server sets up and performs the TRANSACT2 with the information provided.
5. Upon completion of the TRANSACT2, the server sends back up to the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all bytes requested have been returned. On each iteration of this response, smb_rprcnt and smb_rdrcnt are reduced by the number of matching bytes returned in the previous response. The parameter count (smb_rdrcnt) may then continue to be counted down. Fewer than the requested number of bytes may be returned.

The flow for the TRANSACT2 protocol when the request parameters and data does NOT all fit in a single buffer is:

consumer---------->	TRANSACT2 request(data)	>------>server
consumer<--------	OK send remaining data	<-------server
consumer--------->	TRANSACT2 secondary request 1 (data)	>----->server
consumer--------->	TRANSACT2 secondary request 2 (data)	------->server
	server sets up and performs the TRANSACT2	
consumer<----------	TRANSACT2 response 1 (data)	<-------server
consumer<----------	TRANSACT2 response 2 (data)	<-------server
consumer<----------	TRANSACT2 response n (data)	<--------server

The flow for the Transaction protocol when the request parameters and data does not all fit in a single buffer is:

consumer---------->	TRANSACT2 request(data)	>------>server
	server sets up and performs the TRANSACT2	
consumer<--------	OK send remaining data	<-------server
consumer<----------	TRANSACT2 response 1 (data) (only if all data fits in buffer)	<-------server
consumer<----------	TRANSACT2 response 2 (data)	<-------server

consumer<----------	TRANSACT2 response n (data)	<--------server

Note that the primary request through the final response make up the complete protocol, thus the TID, PID, UID, and MID are expected to remain constant and can be used by both the server and consumer to route the individual messages of the protocol to the correct process.

Transaction may generate the following errors:

Error Class ERRDOS

ERRnoaccess

Error Class ERRSRV

ERRerror

ERRinvnid

ERRaccess

ERRmoredata

<irnplementafion specific>

Error Class ERRHRD

<implementation specific>

The Write and Close Command

Request Format (same length as core WRITE or extended WRITEandX):

Data type	Field	Value
BYTE	smb_wct;	value = 6 OR 12
WORD	smb_fid;	file handle (close after write)
WORD	smb_count;	number of bytes to write
DWORD	smb_offset;	offset in file to begin write
DWORD	smb_mtime;	modification time
DWORD	smb_rsvd1;	Optional
DWORD	smb_rsvd1;	Optional
DWORD	smb_rsvd1;	Optional
WORD	smb_bcc;	1 (for pad) + value of smb_count
BYTE	smb_pad;	force data to dword boundary
BYTE	smb_data[];	data

Response Format (same as core WRITE):

Data type	Field	Value
BYTE	smb_yct;	value = 1
WORD	smb_count;	number of bytes written
WORD	smb-bcc;	value = 0

Service:

The Write and Close request is used to first write the specified bytes and then close the file.

Buffered write behind data (and read ahead data) is commonly kept in a buffer also containing space for the Wiite SMB protocol. This protocol allows the final write behind data to be flushed when the file is closed with a single protocol.

NOTE - the smb_wct field MUST be used in order to correctly locate the data to be written.

This protocol may be the same length (smb_wct = 6) as the "core" Write request protocol such that the buffered data is in the correct position and only the smb header need be changed to cause the final bytes to be written along with the file close. This is efficient if the data were read using the "core" read protocol. Note that the "core" Read response protocol is this same size as the "core" write request protocol and the "extended" "WriteandUnlock" and "LockandRead" protocols defined in this document.

Alternately, this protocol may be the same length (smb_yct = 12) as the "extended" WriteandX protocol such that the buffered data is in the correct position and only the smb header need be chainged to cause the final bytes to be written along with the file close. This is efficient if the data were read using the "extended" ReadandX protocol. Note that the "extended" ReadandX response protocol is this same size as the "extended" WriteandX request defined in this document.

If an error occurs on the write, the file should still be closed.

The server should "spin" writing all data to the file/pipe/device before doing the close.

Write and Close may generate the following errors:

Error Class ERRDOS:

ERRnoaccess

ERRbadfid

ERRlock

ERRbadfiletype

ERRbadaccess

Error Class ERRSRV:

ERRerror

ERRinvnid

<implementation specific>

Error Class ERRHRD:

<implementation specific>

The Write and Unlock Command

Request Format (same as core WRITE):

Data type	Field	Value
BYTE	smb_wct;	value = 5
WORD	smb_fid;	file handle
WORD	smb_count;	number of bytes to write and then unlock
DWORD	smb_offset;	offset in file to unlock and begin write
WORD	smb_remcnt;	number of bytes remaining to be written
WORD	smb_bcc;	minimum value = 3
BYTE	smb_identl;	value = DATA-BLOCK
WORD	smb_size;	length of data being written
BYTE	smb_data[];	data

Response Format (same as core WRITE):

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_count;	number of bytes written
WORD	smb_bcc;	value = 0

Service:

The Write and Unlock request is used to first write the specified bytes and then unlock them.

The locked portion of a file is "safe" to write behind because no other process can access the locked bytes until this process unlocks the bytes. Thus the consumer can buffer the locked bytes locally while they are being updated, then when the unlock request is received submit this protocol to both write and then unlock bytes.

This can provide significant performance improvements on data base update operations (lock data-> read data -> [update -> write data] -> unlock data).

Whether or not this protocol is supported (along with LockandRead) is returned in the smb_flg field of the negotiate response.

The request and response format are identical to the core write. The server merely unlocks the bytes after writing them.

If an error occurs on the write, the bytes should remain locked.

Write and Unlock may generate the following errors:

Error Class ERRDOS:

ERRnoaccess

ERRbadfid

ERRIock

ERRbadaccess

Error Class ERRSRV:

ERRerror

ERRindevice

ERRinvnid

<implementation specific>

Error Class ERRHRD:

<implementadon specific>

The Write and X Command

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 12
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_reh2;	reserved (must be zero)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb-wct)
WORD	smb_fid;	file handle
DWORD	smb_offset;	offset in file to begin write
DWORD	smb_timeout;	number of milliseconds to wait for completion
WORD	smb-wmode;	write mode: bitO - complete write before retum (write through) bit1 - return smb - remaining (pipes/devices only) bit2 - use WriteRawNamedPipe (pipes only) bit3 - this is the start of a message (pipes only)
WORD	smb_countleft;	bytes remaining to write to satisfy user's request
WORD	smb_rsvd;	reserved
WORD	smb_dsize;	number of data bytes in buffer (min value = 0)
WORD	smb_doff;	offset (from start of SMB bdr) to data bytes
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes value of smb_dsize)

Response Format:

Data type	Field	Value

BYTE	smb_wct;	value = 6
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_res2;	reserved (pad to word)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
WORD	smb_count;	number of bytes written
WORD	smb_remaining;	bytes remaining to be read (pipes/devices only)
DWORD	smb_rsvd;	reserved
WORD	smb-bcc;	value = 0

Service:

The expanded write and X command allows writes to be timed out, and offers a generalized alternative to the core write command.

Note that a zero length write (smb_count = 0) does NOT truncate the file as is true of the core write protocol. Rather a zero length write merely transfers zero bytes of information to the file (times associated with the file may be updated however). The the core "Write" protocol must be used to truncate the file.

The entire message sent and received including the optional second protocol must fit in the negotiated max transfer size.

The following are the only valid protocol requests commands for smb_com2(X) for WRITE and X:

READ

READ and X

LOCKING AND X

LOCKREAD

CLOSE

CLOSE and DISCONNECT

When the smb_timeout field is non-zero, it specifies the maximum milliseconds the server is to wait for a response to its write command. This feature is useful when accessing remote devices, such as terminals, where indeterminate delays are possible (e.g. contirol-S active).

Zero in the smb_timeout field indicates that no blocking is desired. The server should write only as many bytes to the pipe or device as will be accepted without causing any delay.

A negative 2 smb_timeout value indicates that the server should use the default timeout value associated with the pipe or device being written. Thus no timeout is explicitly set to the resource, rather the current timeout set either as a default or as a result of an IOCTL remains in effect.

A negative 1 value in the smb_timeout field indicates that the server should block (or loop) writing all the data (or error) before retunaing. Thus the server should try "forever" to get the data to the resource.

The Write command's scope is extended to Named Pipes, communication devices, printer devices and spooled output (can be used in place of "Write Print File").

T'he server should "spin" here writing all date to the file/pipe/device if the write is followed by a close protocol (the "X" of WriteAndX present in the same request is a close).

The retum field smb_remaining is to be returned for pipes or devices only. It is used to return the number of bytes currently available in the pipe or device. This information can then be used by the consumer to know when a subsequent (non-blocking) read of the pipe or device may return some data. Note - that when the read request is actually received by the server there may

be more or less actual data in the pipe or device (more data has been written to the pipe / device or another reader drained it). If the information is currently not available or the request is NOT for a pipe or device (or the server does not support this feature), a -1 value should be returned.

Write and X may generate the following errors:

Error Class ERRDOS:

ERRnoaccess

ERRbadfid

ERRlock

ERRbadfiletype

ERRbadaccess

Error Class ERRSRV:

ERRerror

ERRinvnid

ERRtimeout

<implementation specific>

Error Class ERRHRD:

<implementation specitic>

The Write Block Multiplexed Command

Primaly Request Format (smb-com = SMBwriteBmpx):

Data type	Field	Value
BYTE	smb_wct;	value = 12
WORD	smb_fid;	file handle
WORD	smb_tcount;	total bytes (including this buf, 65,535 max)
WORD	smb_ysvd;	reserved
DWORD	smb_offset;	offset in file to begin write
DWORD	smb_timeout;	number of milliseconds to wait for completion
WORD	smb_wmode;	write mode:
		bitO - complete write to disk and send final result response
DWORD	smb_rsvd2;	reserved
WORD	smb_dsize;	number of data bytes this buffer (min value = 0)
WORD	smb_doff,	offset (from start of SMB hdr) to data bytes
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb-data[*];	data bytes (* = value of smb-dsize)

First Response Format (ok send remaining data), (smb-com = SMBwriteBmpx):

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_remaining;	bytes remaining to be read (pipes/devices only)
WORD	smb_bcc;	value = 0

Secondary Request Format (more data) (zero to n of these):

Data type	Field	Value
BYTE	smb_wct;	value = 8
WORD	smb_fid;	file handle
WORD	smb_tcount;	total bytes to be sent this

		protocol
DWORD	smb_offset;	offset in file to begin write
DWORD	smb_mvd;	reserved
WORD	smb_dsize;	number of data bytes this buffer (min value = 0)
WORD	smb doff;	ouset (horn start of SMB hdr) to data bytes
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb-data[*];	data bytes (* = value of smb-dsize)

First Response Format:

Data type	Field	Value
BYTE	smb_yct;	value = 1
WORD	smb_count;	total number of bytes written
WORD	smb_bcc;	value = 0

Service:

The Write Block Multiplexed protocol is used to maximize the performance of writing a large block of data from the consumer to the server on a mulfiplexed VC.

The Write Block Multiplexed command's scope includes (but is not limited to) files, Named Pipes, communication devices, printer devices and spooled output (can be used in place of "Write Print File").

Note that the first response format will be that of the final response (SMBwriteC) in the case where the server gets an error while writing the data sent along with the request. Thus the word parameter is smb_count (the number of bytes which did get written) any time an error is retumed. If an error occurs AFTER the first response has been sent allowing the consumer to send the remaining data, the final response should NOT be sent unless write through is set. Rather the server should retum this "write behind" error on the next access to the file/pipe/device.

When this protocol is used, other requests may be active on the multiplexed VC. The server will respond with the response protocol message as defined above. The consumer will then send a sequence of "Secondary Write" protocol requests until the remaining data amount has been sent (unless afl data fit within primary request). Each request contains the smb_pid of the original Write Block Mulfiplexed request, the file offset and data length defined in the Write response protocol (including the SMB header). This allows the server's message delivery (multiplexing) system to deliver the response to the appropriate server process.

At the time of the request, the consumer knows the number of data bytes expected to be sent and passes this information to the server via the primary request (smb_tcount). This may be reduced by lowering the total number of bytes expected (smb_tcount) in each (any) secondary request. Thus, when the amount of data bytes received by the server (total of each smb_dsize) equals the total amount of data bytes expected (smallest smb_tcount received), then the server has received afl the data bytes. This allows the protocol to work even if the "packets" (buffers) are received out of sequence.

This protocol eliminates nearly half the protocols involved with writing a block of data since the Write Block Mulfiplexed response is sent only once as opposed to each negotiated buffer size as defined with the Write protocol.

When write through is not specified (smb_wmode zero), this protocol is assumed to be a form of write behind. The transport layer guarantees delivery of all secondary requests from the consumer. Thus no "got the data you sent" protocol is needed. If an error should occur at the server end, all bytes must be received and thrown away. If an error occurs while writing data to disk such as disk full, the next access of the file handle (another write, close, read, etc.) will return the fact that the error occurred.

If write through is specified (smb_wmode set), the server will collect all the data, write it to disk and then send a final response

indicating the result of the write, (no error in smb_err indicates data is on disk ok). The total number of bytes written is also returned in this response.

The flow for the Write Block Multiplexed (W.B.M.) protocol is:

consumer-------->	WRITE BLOCK MULTIPLEXED request (data)	>--------> server
consumer<--------	<OK send remaining data<	--------- server
consumer-------->	W.B.M. secondary request 1 (data)>-----	--------->server
consumer-------->	W.B.M. secondary request 2 (data)>-----	--------->server
consumer-------->	W.B.M. secondary request n (data)------>	>------->server
consumer<-----<	data on disk or error (write through only)<	-------- server

Note - if all the data being sent fits in the first request buffer, the primary response will still be sent followed by the final response after the data is actually on disk (if write through is set). This is done in order to simplify the implementation of this protocol. When writing data which all fits within a negotiated buffer size, the "Write and X" protocol may be a better choice.

Note that the primary request through the final response make up the complete protocol; thus the TID, PBD, UiD and MID are expected to remain constant and can be used by both the server and consumer to route the individual messages of the protocol to the correct process.

The return field smb_remaining is to be refumed for pipes or devices only. It is used to return the number of bytes currently available in the pipe or device. This information can then be used by the consumer to know when a subsequent (non blocking) read of the pipe or device may retum some data. Note - that when the read request is actually received by the server there may be more or less actual data in the pipe or device (more data has been written to the pipe / device or another reader drained it). If the information is currently not available or the request is NOT for a pipe or device (or the server does not support this feature), a -1 value should be returned.

Write Block Multiplexed may generate the following errors. Note that the error ERRnoresource (or ERRusestd) may be naumed by the soner if it is temporarily out of large buffers. The consumer could then retry using the standard "core" write request or delay and retry the read block multiplexed request.

Error Class ERRDOS

ERRbadfid

ERRnoaccess

ERRIock

ERRbadfiletype

ERRbadaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRnoresource

ERRusestd

ERRtimeout

<implementation specific>

Error Class ERRHRD

<irnplementation specific>

The Write Block Raw Command

Primary Request Format (smb-com = SMBwriteBraw):

Data type	Field	Value
BYTE	smb_wct;	value = 12
WORD	smb_fid;	file handle
WORD	smb - tcount;	total bytes (including this buf, 65,535 max
WORD	smb_rsvd;	reserved *
DWORD	smb_offset;	offset in file to begin write
DWORD	smb timeout;	number of milliseconds to wait for completion
WORD	smb-wmode;	write mode:
		bit O - complete write to disk and send final result response
		bit 1 - retum smb_remaining (pipes/devices only)
DWORD	smb rsvd2;	reserved
WORD	smb dsize;	number of data bytes this buffer (min value = 0)
WORD	smb_doff;	offset (from start of SMB hdr) to data bytes
WORD	smb - bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb-data[*];	data bytes (* = value of smb_dsize)

First Response Format (ok smd the remaining data), (smb_com = SMBwriteBraw):

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_remaining;	bytes remaining to be read (pipes/devices only)
WORD	smb_ftc;	value = 0

Secondary Request Format (write through or error), (smb_com = SMBwriteC):

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_cound;	total number of bytes written
WORD	smb_bcc;	value = 0

Service:

The Write Block Raw protocol is used to maximize the performance of writing a large block of data from the consumer to the server.

The Write Block Raw command's scope includes (but is not limited to) files, Named Pipes, communication devices, printer devices and spooled output (can be used in place of "Write Print File").

Note that the first response format will be that of the final response (SMBwriteC) in the case where the server gets an error while writing the data sent along with the request. Thus the word parameter is smb_count (the number of bytes which did get written) any time an error is returned. If an error returns AFTER the first response has been sent allowing the consumer to send the remaining data, the final response should NOT be sent unless write through is set. Rather the server should retum this "write behind" error on the next access to the file/pipe/device.

When this protocol is used, the consumer has guaranteed that there is (and will be) no other request on the VC for the duration of the Write Block Raw request. The server will allocate (or reserve) enough memory to receive the data and respond with a response protocol message as defined above. The consumer will then send the raw data (one send). Thus the server is able to receive up to 65,535 bytes of data directly into the server buffer. Note that the amount of data transferred is expected to be larger than the negotiated buffer size for this protocol.

The reason that no other requests can be active on the VC for the duration of the request is that if other receives are present on the VC, there is normally no way to guarantee that the data will be received into the large server buffer; rather the data may fill one (or more) of the other buffers. Also if the consumer is sending other requests on the VC, a request may land in the buffer that the server has allocated for the Write Raw Data.

Support of this protocol is optional.

Whether or not Write Block Raw is supported is retumed in the response to negotiate and in the LAN MAN 1.0 extended "Query Server Information" protocol.

When write through is not specified (smb_wmode zero), this protocol is assumed to be a form of write behind. The transport layer guarantees delivery of all secondary requests from the consumer. Thus no "got the data you sent" protocol is needed. If an error should occur at the server end, all bytes must be received and thrown away. If an error occurs while writing data to disk such as disk full, the next access of the file handle (another write, close, read, etc.) will return the fact that the error occurred.

If write through is specified (smb_wmode set), the server will receive the data, write it to disk and then send a final response indicating the result of the write(no error in smb_err indicates data is on disk ok). The total number of bytes written is also retumed in this response.

The flow for the Write Block Raw protocol is:

consumer------>	WRITE BLOCK RAW request (optional data)	>-------server---->
consumer<------	<-OK send (more) data	<-------server
consumer------>	raw data>----------	---------server
consumer<------	<- data on disk or error (write through only)	<-------server

This protocol is set up such that the Write Block Raw request may also carry data. This is an optimization in that up to the server's buffer size (smb_maxxmt from negotiate response), minus the size of the Write Block Raw protocol request, may be sent along with the request. Thus if the server is busy and unable to support the Raw Write of the remaining data, the data sent along with the request has been delivered and need not be sent again. The Server will write any data sent in the Write Block Raw request (and wait for it to be on the disk or device if write through is set), prior to sending the "send raw data" or "no resource" response.

The specific responses error class ERRSRV, error codes ERRusempx and ERRusestd, indicate that the server is temporarily out of huge buffers needed to support the Raw Write of the remaining data, but that any data sent adong with the request has been successfully written. The consumer should then write the remaining data using Write Block Multiplexed (if ERRusempx was returned) or the standard "core" write request (if ERRusestd was retumed), or delay and retry using the Write Block Raw request. If a write error occurs writing the initial data, it will be retumed and the Write Raw request is implicitly denied.

Note that the primary request through the final response make up the complete protocol, thus the TID, PID, UID and MID are expected to remain constant and can be used by the consumer to route the individual messages of the protocol to the correct process.

The retum field smb_remaining is to be retumed for pipes or devices only. It is used to return the number of bytes currently available in the pipe or device. This information can then be used by the consumer to know when a subsequent (non blocking) read of the pipe or device may retum some data. Note that when the read request is actually received by the server there may be more or less actual data in the pipe or device (more data has been written to the pipe/device or another reader drained it). If the information is currently not available or the request is NOT for a pipe or device (or the server does **not** support thd feature), a -1 value should be retumed.

Write Block Raw may generate the following errors.

Error ERRDOS

ERRbadhd

Ekkwamen

ERRIock

ERRbadfiletype

ERRbadaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRnoresource

ERRtimeout

ERRusempx

ERRusestd

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Find Command

Request Format (same as core Search Protocol):

Data type	Field	Value
BYTE	smb_wct;	value = 2
WORD	smb_count;	max number of entries to find
WORD	smb_attr;	search attribute
WORD	smb_bcc;	minimum value = 5
BYTE	smb_ident1;	ASCII (04)
BYTE	smb_pathname[];	filename (may contain global characters)
BYTE	smb_ident2;	Variable Block (05)
WORD	smb_keylen;	resume key length (zero if "Find First")
BYTE	smb_resumekey[*];	"Find Next" key value of smb_keylen)

Response Format (same as core Search Protocol):

Data type	Field	Value
BYTE	smb_wct;	value = 1
WORD	smb_count;	number of entries found
WORD	smb_bcc;	minimum value = 3
BYTE	smb_ident;	Variable Block (05)
WORD	smb_datalen;	data length
BYTE	smb_data[*];	directory entries

Directory Information Entry (dir_info) Format (same as core Search Protocol):

Data type	Field	Value
BYTE	find_buf _reserved[21];	reserved (resume-key)
BYTE	find_buf_attr;	attribute
WORD	find_buf_time;	modification time (hhhhb mmmmmm xxxxx) where XXXX is in two second increments
WORD	find_buf_date;	modification date (yyyyyyy ddddd)
DWORD	find_buf_size;	file size
BYTE	find_buf_pname[13];	file naine -- ASCII (null

terminated)

The resume key has the following format:

Data type	Field	Value
BYTE	sr_res;	reserved
		bit 7 - reserved for consumer use
		bit 5, 6 - reserved for system use (must be preserved)
		bits 0-4 - reserved for server (must be preserved)
BYTE	sr_name [11];	pathname sought. Format: 0 - 3 character extension, left justified (in last 3 characters)
BYTE	sr_findid[1];	uniquely identifies find through find close
BYTE	sr_server[4]'	available for server use (must be non-zero)
BYTE	sr_res[4];	reserved for consumer use

Service:

The Find protocol finds the directory entry or group of entries matching the specified file pathname. The filename portion of the pathname may contain global (wild card) characters.

The Find protocol is used to match the find OS/2 system call. The protocols "Find", "Find Unique" and "Find Close" are methods of reading (or searching) a directory. These protocols may be used in place of the core "Search" protocol when LANMAN 1.0 dialect has been negotiated. There may be cases where the Search protocol will sill be used.

The format of the Find protocol is the same as the core "Search" protocol. The difference is that the directory is logically Opened with a Find protocol and logically closed with the Find Close protocol. This allows the Server to make better use of its resources. Search buffers are thus held (allowing search resumption via presenting a "resume_key") until a Find Close protocol is received. The sr_findid field of each resume key is a unique identifier (within the session) of the search from "Find" through "Find close". Thus if the consumer does "Find ahead", any find buffers containing resume keys with the matching find id may be released when the Find Close is requested.

As is true of a failing open, if a Find request (Find "first" request where resume-key is null) fails (no entries are found), no find close protocol is expected.

If no global characters are present, a "Find Unique" protocol should be used (only one entry is expected and find close need not be sent).

The file path name in the request specifies the file to be sought. The attribute field indicates the attributes that the file must have. If the attribute is zero then only normal files are returned. If the system tile, hidden or directory attributes are specified then the search is inclusive -- both the specified type(s) of files and normal files are returned. If the volume label attribute is specified then the search is exclusive, and only the volume label entry is returned

The max_count field specifies the number of directory entries to be returned. The response will contain zero or more directory entries as determined by the count-returned field. No more than max_count entries will be returned. Only entries that match the sought filename/attribute will be returned.

The resume_key field must be null (length = 0) on the initial ("Find First") find request. Subsequent find requests intended to continue a search must contain the resume-key field extracted from the last directory entry of the previous response. The resume_key field is self-contained, for on calls containing a resume - key neither the attribute or pathname fields will be valid in the request. A find request will terminate when either the requested maximum number of entries that match the named file are

found, or the end of directory is reached without the maximum number of matches being found. A response containing no entries indicates that no matching entries were found between the starting point of the search and the end of directory.

There may be multiple matching entries in response to a single request as Find supports "wild cards" in the file name (last component of the pathname). "?" is the wild card for single characters, "*" or "null" will match any number of filename characters within a single part of the filename component. The filename is divided into two parts -- an eight character name and a three character extension. The name and extension are divided by a ".".

If a filename part commences with one or more "?"s then exactly that number of characters will be matched by the Wild Cards, e.g., "??X" will equal "abx" but not "abcx" or "ax". When a filename part has trailing "?"s then it will match the specified number of characters or less, e.g., "x??" will match "xab", "xa" and "x", but not "xabc". If only "?"s are present in the filename part, then it is handled as for trailing "?"s.

"*" or "null" match entire pathname parts, thus "*.abc" or ".abc" will match any file with an extension of "abc". or "null" will match all files in a directory.

Unprotected servers require the requester to have read permission on the subtree containing the directory searched (the share specified read permission).

Protected servers require the requester to have permission to search the specified directory.

If a Find requests more data than can be placed in a message of the max_xmit_size for the TID specified, the server will return only the number of entries which will fit.

The number of entries returned will be the minimum of-

1. The number of entries requested.
2. The number of (complete) entries that will fit in the negotiated SMB buffer.
3. The number of entries that match the requested name pattern and attributes.

The error ERRnofiles set in smb_err field of the response header or a zero value in smb_count of the response indicates no matching entry was found.

The resume search key returned along with each directory entry is a server defined key which when returned in the Find Next protocol, allows the directory search to be resumed at the directory entry following the one denoted by the resume search key.

The date is in the following format:

bits:

1111	11		
5432	1098	7654	3210
yyyy	yyym	mmmd	dddd

where:

y - bit of year 0 - 119 (1980 - 2099)

m - bit of month 1 - 12

d - bit of day 1- 31

The time is in the following format:

bits:

1111	11		
5432	1098	7654	3210
hhhh	hmmm	mmmx	xxxx

where:

h - bit of hour (0 - 23)

m - bit of minute (0-59)

x - bit of two-second increment

Find may generate the following errors.

Error Class ERRDOS

ERRnofiles

ERRbadpath

ERRnoaccess

ERRbadshare

<implementation specific>

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implmentation specific>

Error Class ERRHRD

<implementation specific>

The Open and X Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_reh2;	reserved (must be zero)
WORD	smb_off2;	offset (from SMB header start) to next cmd (@smb_wct)
WORD	smb_flags;	additional information: bit 0 - if set, return additional information bit 1 - if set, set single user total file lock (if only access) bit 2 - if set the server should notify the consumer on any action which can modify the file (delete, setattrib, rename, etc.). if not set, the server need only notify the consumer on another open request. This bit only has meaning if bit 1 is set.
WORD	smb_mode	file open mode
WORD	smb_sattr;	search attributes
WORD	smb_attr;	file attributes (for create)
DWORD	smb_time;	create time
WORD	smb_ofun;	open function
DWORD	smb_size;	bytes to reserve on "create" or "truncate"
DWORD	smb_timeout;	max milliseconds to wait for resource to open
DWORD	smb_rsvd;	reserved (must be zero)
WORD	smb_bcc;	minimum value = 1
BYTE	smb_pathname[];	the pathname

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15

BYTE	smb_com2;	secondary (X) command, OXFF = none
BYTE	smb_res2;	reserved (pad to word)
WORD	smb_off2;	offset (from SMB hdr start) to next cmd (@smb_wct)
WORD	smb_fid;	file handle
+ WORD	smb_attribute;	attributes of file or device
+DWORD	smb_time;	last modification time
+DWORD	smb_size;	current file size
+WORD	smb_access;	access permissions actually allowed
+WORD	smb_type;	file type
+WORD	smb_state;	state of IPC device (e.g., pipe)
WORD	smb_action;	action taken
DWORD	smb_fileid;	server unique file id
WORD	smb_rsvd;	reserved
WORD	smb_bcc;	value = 0

+ returned only if bit 0 of smb_flags is set in request

Service Enhancement:

The open protocol request is enhanced in order to accommodate the new open system call used in OS/2 and provide additional functionality.

The entire message sent and received including the optional second protocol must fit in the negotiated max transfer size.

The following are the only valid protocol requests commands for smb_com2 (X) for OPEN and X:

READ

READ and X

IOCTL

The "mode" field for open, referenced as r/w/share in the core protocol document, is enhanced to allow direct access mode for the file, and to allow an open for execute. Systems that do not support execute mode should treat the execute mode as equivalent to read mode. This word has the following format:

smb_mode bit field mapping:

bits:

1111	11		
5432	1098	7654	3210
rWrr	rrrr	rSSS	rAAA

where:

W - Write through mode. No read ahead or write behind allowed on this file (or device). When protocol is returned, data is expected to be on the disk (or device).

r - reserved (must be zero).

SSS - Sharing mode

Compatibility mode (as in core open protocol)

Deny read/write/execute (exclusive).

Deny write.

Deny read/execute.

Deny none.

AAA - Access mode

Open for reading.

Open for writing.

Open for reading and writing.

Open for execute

rSSSrAAA = 11111111 (hex FF)

indicates FCB open (as in core open protocol)

The "open function" field specifies the action to be taken depending on whether or not the file exists.

This word has the following format:

smb_ofun bit field mapping:

bits:

1111	11		
5432	1098	7654	3210
rrrr	rrrr	rrrC	rrOO

where:

C - Create (action to be taken if file does not exist).

0 Fail.

0 Create file.

r - reserved (must be zero).

0 - Open (action to be taken **if** file exists).

0 - fail.

1 - Open file

2 - Truncate file

I/O devices can be opened in queuing mode, in that if the device (or all devices of type requested) is currently in use the user may optionally queue waiting for the device to become free. Thus a non zero smb_timeout field is used to indicate that queuing is desired.

If queuing is requested, the value in the smb_timeout field is used as the maximum number of milliseconds to wait for the device to become free. A value of zero means no delay (do not queue), a value of (long) -1 indicates to wait forever (no timeout). The server will not send the response back to the consumer until the resource being queued for is actually opened (or the specified timeout time has passed). Note that although the timeout is specified in milliseconds (in order to match the OS/2 system calls), by the time that the timeout occurs and the consumer receives the timeout protocol much more time than specified may have occurred.

The "Action Taken" field specifies the action as a result of the Open request. This word bas the following format:

smb_action bit field mapping:

bits:

1111	11		
5432	1098	7654	3210

rrrr rrrr rrrC rrOO

where:

L - Lock (single user total file lock status).

0 file opened by another user (or mode not supported by server).

file is opened only by this user at the present time.

0 - reserved (must be zero).

O - Open (action taken on Open).

1 - The file existed and was opened.

2 - The file did not exist but was created.

3 - The file existed and was truncated.

The attribute fields (smb_attr, smb_sattr and smb_attribute) have the following format (bit0 is the least significant bit. This field matches that used by OS/2.

bit0 - read only file

bit1 - "hidden" file

bit2 - system file

bit3 - reserved

bit4 - directory

bit5 - archive file

bits6-15 - reserved (must be zero)

The search attribute field (smb_sattr) indicates the attributes that the file must have to be found while searching to see if it exists. If the search attribute is zero then only normal files are returned. If the system file, hidden or directory attributes are specified then the search is inclusive both the specified type(s) of files and nominal files are returned.

The resource type field (smb_type) defines the additional resource types:

0 - Disk file or directory as defined in the attribute field.

1 - FIFO (named pipe)

2 - Named pipe (message mode)

3 - LPT (printer) Device

4 - COM (communication) Device

IPC State Bits (smb_state)

5432109876543210

BE**TTRR|---Icount--|

where:

B - Blocking - 0 => reads/writes block if no data available

1 => reads/writes return immediately if no data

E - Endpoint - 0 => consumer end of pipe

1 => server end of pipe

TT - Type of pipe - 00 => pipe is a bytes stream pipe

00 => pipe is a message pipe

RR - Read mode - 00 => Read pipe as a byte stream

01 => Read messages from pipe

Icount - 8 bit count to control pipe instancing (N/A)

A "single user total file lock" is also known as an "opportunistic lock". A consumer requests an "opportunistic lock" by setting the

appropriate bit in the OpenX, Open, Create and MakeNew protocols whenever the file is being opened in a mode which is not exclusive. The server responds by setting the appropriate bit in the response protocol indicating whether or not the "opportunistic lock" was granted. By granting the "oplock", the server tells the consumer that the file is currently ONLY being used by this one consumer process at the current time. The consumer can therefore safely do read ahead and write behind as well as local caching of file locks knowing that the file will not be accessed/changed in any way by another process while the "oplock" is in effect.

The consumer will be notified when any other process attempts to open the "oplocked" file and if "opbatch" (bit 2 of smb_flags) was set on the OpenX request the consumer will also be notified on any operation which may change the file.

When another user attempts to Open (or otherwise modify if "opbatch" was requested) the file which a consumer has oplocked, the server will "hold off" the 2nd attempt and notify the consumer via a LockingX protocol (with bit one of smb_locktype set) that the "oplock" is being broken. The consumer is expected to then flush any dirty buffers, submit any file locks (LockingX protocol can be used for this) and respond to the server with either a LockingX protocol (with bit one of smb_locktype set) or with a close protocol if the file is no longer in use. Note that because a close being sent to the server and break oplock notification from the server could cross on the wire, if the consumer gets an oplock notification on a file which it does not have open, that notification should be ignored. Once the ."oplock" has been broken, the consumer must no longer do any form of data or lock caching. The "oplock" is never reinstated while the file is open. If the file is still open once the consumer has been notified, the 2nd opener does not get the file "oplocked" along with the open. If the file is closed by the consumer which had it open, the server is again free to grant the new opener the oplock.

Note that the "oplock" broken notification will only go to one consumer because after the oplock is broken, any further open attempts will just get the oplock request denied.

Also note that due to tuning, the consumer could get an "oplock" broken notification in a user's data buffer as a result of this notification crossing on the wire with a Read Raw request. The consumer must detect this (use length of msg, "FFSMB", MID of -1 and smb_cmd of SMBLockingX) and honor the "oplock" broken notification as usual. The server must also note on receipt of an Read Raw request that there is an outstanding (unanswered) "oplock" broken notification to the consumer and return a zero length response denoting failure of the read raw request. The consumer should (after responding to the "oplock" broken notification), use a standard read protocol to redo the read request. This allows a file to actually contain data matching an "oplock" broken notification and still be read correctly.

"Oplock" is a major performance win in the real world because many files must be opened in a non exclusive mode because the file could he used by others. However often, the files are not actually in use by multiple users at the same instant.

The following errors may be generated by Open and X:

Error Class ERRDOS

ERRbadfile

ERRnofids

ERRnoaccess

ERRshare

ERRbadaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRaccess

ERRinvnid

<implementation specific>

Error Class ERRHRD

<implementation specific>

The Set Extended File Attributes Command

Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 7
WORD	smb fid;	file handle
WORD	smb cdate;	date of creation
WORD	smb_ctime;	time of creation
WORD	smb_adate;	date of last access
WORD	smb_atime;	time of last access
WORD	smb_mdate;	date of last modification
WORD	smb_mtime;	time of last modification
WORD	smb_bcc;	minimum value = 0
BYTE	smb_rsvd[];	reserved

Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 0
WORD	smb_bcc;	value = 0

Service Enhancement:

The Set Extended File Attributes is enhanced to set information about the queried file. The target file is updated from the values specified. A null date/time (0) indicates to leave that specific date/time unchanged.

The file is specified by the file handle (FID).

The dates are in the following format:

bits:

1111	11		
5432	1098	7654	3210
yyyy	yyym	mmmd	dddd

where:

y - bit of year 0 - 119 (1980 - 2099)

m - bit of month 1 -12

d - bit of day 1 - 31

The times are in the following format:

bits:

1111	11		
5432	1098	7654	3210
hhhh	hmmm	mmmx	xxxx

where:

h - bit of hour (0 - 23)

m - bit of minute (0 - 59)

x - bit of 2 second increment

Set Expanded File Attributes may generate the following errors:

Error Class ERRDOS

ERRbadfile

ERRbadfid

ERRnoaccess

<implementation specific>

Error Class ERRSRV

ERRerror

ERRinvnid

ERRaccess

<implementation specific>

Error Class ERRHRD

<implementafion specific>

The Transaction Command

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 14 + value of smb_suwcnt
WORD	smb_tpscnt;	total number of param bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	max number of parameter bytes to return
WORD	smb_mdrcnt;	max number of data bytes to return
BYTE	smb_msrcnt;	max number of setup words to return
BYTE	smb_rsvd1;	reserved
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB header) to parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	setup word count
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup[*];	variable number of setup words (* = smb_suwcnt)
WORD	smb_bcc;	total bytes (including pad bytes) following
WORD	smb_name[];	name of transaction
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_pscnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Interim Response Format (if no error - ok send remaining data):

Data type	Field	Value
BYTE	smb_wct;	value = 0
WORD	smb_bcc;	value = 0

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 8
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_psdisp	byte displacement for these parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these parameter bytes
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_pscnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Interim Response Format (if no error - ok send remaining data):

Data type	Field	Value

BYTE	smb_wct;	value = 0
WORD	smb_bcc;	value = 0

Data type	Field	Value
BYTE	smb_wct;	value = 10 + value of smb_suwcnt
WORD	smb_tprscnt;	total number of parameter bytes being returned
WORD	smb_tdrscnt;	total number of data bytes being returned
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	number of parameter bytes being returned this buffer
WORD	smb_proff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_drcnt;	number of data bytes being returned this buffer
WORD	smb_droff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	setup word count
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_setup[*];	variable number of setup words (* = smb_suwcnt)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_prcnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_drcnt)

The Transaction protocol performs a symbolically named transaction. This transaction is known only by a name (no file handle used).

The Transaction command's scope includes (but is not limited to) Named Pipes and Mail Slots. Where the resource is unidirectional (such as class 2 writes to Mail Slots), bit 1 of smb_flags on the request can be set indicating that no response is needed.

The Transaction "set up information" and/or parameters define functions specific to a particular resource on a particular server.

Therefore the functions supported are not defined by the protocol, but by consumer/server implementations. The protocol simply provides a means of delivering them and retrieving the results.

The number of bytes needed in order to perform the TRANSACTION request may be more than will fit in a single buffer.

At the time of the request, the consumer knows the number of parameter and data bytes expected to be sent and passes this information to the server via the primary request (smb_tpscnt and smb_tdscnt). This may be reduced by lowering the total number of bytes expected (smb_tpscnt and/or smb_tdscnt in each (any) secondary request.

Thus when the amount of parameter bytes received (total of each smb_pscnt) equals the total amount of parameter bytes expected (smallest smb_tpscnt) received, then the server has received all the parameter bytes.

Likewise, when the amount of data bytes received (total of each smb_dscnt) equals the total amount of data bytes expected (smallest smb_tdscnt) received, then the server has received all the data bytes.

The parameter bytes should normally be sent first followed by the data bytes. However, the server knows where each begins and ends in each buffer by the offset fields (smb_psoff and smb_dsoff) and the length fields (smb_pscnt and smb_dscnt). The displacement of the bytes (relative to sun of each) is also known (smb_psdisp and smb_dsdisp). Thus the server is able to reassemble the parameter and data bytes should the "packets" (buffers) be received out of sequence.

If all parameter and data bytes fit into a single buffer, then no interim response is expected (and no secondary request is sent).

The Consumer knows the maximum amount of data bytes and parameter bytes which the server may return (from smb_mprcnt and smb_mdrcnt of the request). Thus it initializes its bytes expected variables to these values. The Server then informs the consumer of the actual amounts being returned via each "packet" (buffer) of the response (smb_tprcnt and smb_tdrcnt).

The server may reduce the expected bytes by lowering the total number of bytes expected (smb_tprcnt and/or smb_tdrcnt) in each (any) response.

Thus when the amount of parameter bytes received (total of each smb_psent) equals the total amount of parameter bytes expected (smallest smb_tprcnt) received, then the consumer has received all the parameter bytes.

Likewise, when the amount of data bytes received (total of each smb_dscnt) equals the total amount of data bytes expected (smallest smb_tdrcnt) received, then the consumer has received all the data bytes.

The parameter bytes should normally be sent first followed by the data bytes. However, the consumer knows where each begins and ends in each buffer by the offset fields (smb_proff and smb_droff) and the length fields (smb_prcnt and smb_drcnt). The displacement of the bytes (relative to sum of each) is also known (smb_prdisp and smb_dsdisp). Thus the consumer is able to reassemble the parameter and data bytes should the "packets" (buffers) be received out of sequence.

Thus the flow is:

1. The consumer sends the first (primary) request which identifies the total bytes (both parameters and data) which are expected to be sent and contains the set up words and as many of the parameter and data bytes bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters and data) the server is to return on TRANSACTION completion. If all the bytes fit in the single buffer, skip to step 4.
2. The server responds with a single interim response meaning "ok, send the remainder of the bytes" or (if error response) terminate the transaction.
3. The consumer then sends another buffer full of bytes to the server. On each iteration of this secondary request, smb_tpscnt and/or smb_tdscnt could be reduced. This step is repeated until all bytes have been delivered to the server (total of all smb_pscnt equals smallest smb_tpscnt and total of all smb_dscnt equals smallest smb_tdscnt).
4. The Server sets up and performs the TRANSACTION with the information provided.
5. Upon completion of the IOCTL, the server sends back (up to) the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all result bytes have been returned. On each iteration of this response, smb_tprcnt and/or smb_tdrcnt could be reduced. This step is repeated until all bytes have been delivered to the consumer (total of all smb_prcnt equals smallest smb_tprcnt and total of all smb_drcnt equals smallest smb_tdrcnt).

Thus the flow is:

1. The consumer sends the first (primary) request which identifies the total bytes (parameters and data) which are to be sent, contains the set up words and as many of the parameter and data byes as will

fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters, and data) the server is to return on TRANSACTION completion. The parameter bytes are immediately followed by the data bytes (the length fields identify the break point). If all the bytes fit in the single buffer, skip to step 4.

2. The server responds with a single interim response meaning "ok, send the remainder of the bytes" or (if error response) terminate the transaction.
3. The consumer then sends another buffer full of bytes to the server. This step is repeated until all bytes have been delivered to the server.
4. The server sets up and performs the TRANSACTION with the information provided.
5. Upon completion of the TRANSACTION, the server sends back up to the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all bytes requested have been returned. On each iteration of this response, smb_rprcnt and smb_rdrcnt are reduced by the number of matching bytes returned in the previous response. The parameter count (smb_rprcnt) is expected to go to zero first because the parameters are sent before the data. The data count (smb_rdrcnt) may then continue to be counted down. Fewer than the requested number of bytes may be returned.

The flow for the TRANSACTION protocol when the request parameters and data does NOT all fit in a single buffer is:

consumer>------	TRANSACTION request (data)	>----------server
consumer-------<	OK send remaining data	<---------server
consumer>-------	TRANSACTION secondary request 1 (data)	>---------server
consumer------->	TRANSACTION secondary request 2 (data)	>---------server
	(server sets up and performs the TRANSACTION)	
consumer<-------	TRANSACTION response 1 (data)	<----------server
consumer<-------	TRANSACTION response 2 (data)	<----------server
consumer<-------	TRANSACTION response n (data)	<----------server

The flow for the Transaction protocol when the request parameters and data does not all fit in a single buffer is:

consumer>------	TRANSACTION request (data)	>----------server
	(server sets up and performs the TRANSACTION)	
consumer<-------	TRANSACTION response 1 (data) -- only one if all data fit in buffer	<---------server
consumer<-------	TRANSACTION response 2 (data)	<----------server
consumer<-------	TRANSACTION response 2 (data)	<----------server
consumer<-------	TRANSACTION response n (data)	<----------server

Note that the primary request through the final response make up the complete protocol; thus the TID, PID, UID, and MID are expected to remain constant and can be used by both the server and consumer to route the individual messages of the protocol to the correct process.

Transaction may generate the following errors:

Error Class ERRDOS:

ERRnoaccess

ERRbadaccess

Error Class ERRSRV:

ERRerror

ERRinvnid

ERRaccess

ERRmoredata

<implementafion specific>

Error Class ERRHRD:

<implementation specific>

Mail Slot Transaction Protocol

The identifier "\MAILSLOT\<name>" denotes a mail slot transaction, where the <name> is the mail slot name to apply the transaction against.

Mail slots using unreliable "class 2" mode may be transmitted via datagrams. However, Mail slots using reliable "class 1" mode must be transmitted on an established VC (reliable delivery is needed).

When "class 1" mail slot transactions are transmitted via a VC, a response may still be desired to ensure that the mail slot transaction was delivered to the mail slot without error. Thus the response bit may be zero in smb_flags to indicate that the error code associated with delivery should be returned.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 17
WORD	smb_tpscnt;	value = 0 total number of param bytes being sent
WORD	smb_tdscnt;	total size of data to write to mail slot (if any)
WORD	smb_mprcnt;	value = 2 one word return code expected
WORD	smb_mdrcnt;	value = 0 size of data read from mail slot (N/A)
BYTE	smb_msrcnt;	value = 0 max number of setup words to return (N/A)
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - if set, also disconnect TID in smb_tid
		bit 1 - if set, no response is required
DWORD	smb_timeout;	(user defined) number of milliseconds to wait
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = 0 no param bytes being sent this buffer
WORD	smb_psoff;	value = 0 no parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 3
BYTE	smb_rsvd2;	reserved (pad above to word)

WORD	smb_setup1;	(op code) value = 1 - Write Mail slot
WORD	smb_setup2;	(priority) priority of transaction
WORD	smb_setup3;	(class) 1 = reliable, 2 = unreliable
WORD	smb_bcc;	total bytes (including pad bytes) following
WORD	smb_name[];	"\MAILSLOT\<name>0"
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data;	data to be written to Mail Slot (if any)
BYTE	smb_data[*];	data to be written to Mail Slot (if any)
		(* = value of smb_dscnt)

Response Format (may respond with zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 2 one word return code
WORD	smb_tdrcnt;	value = 0 no data bytes
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	value = 2 parameter bytes being returned this buf
WORD	smb_proff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_prdisp;	value = 0 byte displacement for these param bytes
WORD	smb_drcnt;	value = 0 no data bytes
WORD	smb_droff	value = 0 no data bytes
WORD	smb_drdisp;	value = 0 no data bytes
BYTE	smb_suwcnt;	value = 0 no setup return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
WORD	smb_retcode;	mail slot delivery return code (ZERO = OK)

Announce (and Request Announce) Mail Slot Transaction Protocol

The LANMAN 1.0 server nodes send the following Mail Slot transaction protocol (announcement form) as a datagram (SEND DATAGRAM to an installation determined group name) periodically to inform consumer nodes that the server exists and is ready to accept VC connection requests.

The LANMAN 1.0 consumer nodes send the following Mail Slot Transaction protocol (announce request form) as a datagram (SEND DATAGRAM to an installation determined group name) to request that server nodes available identify themselves via the announcement Transaction datagram.

Note that the Mail Slot transaction name "\MAILSLOT\LANMAN" is reserved for use by the LAN Manager.

The default group name used by LANMAN 1.0 is "LANGROUP".

Also note that there is no "security" involved with these protocols. The smb_tid and smb_uid fields will be set to -1 and will be ignored by the node receiving this transaction. Each node may apply its own security mechanisms to determine whether to reply to (or send) these protocols.

Announce Mail Slot Transaction Format:

Data type	Field	Value
BYTE	smb_wct;	value = 17
WORD	smb_tpscnt;	value = 0 no param bytes being sent
WORD	smb_tdscnt;	size of announce or req_announce
WORD	smb_mprcnt;	value = 0 no param bytes to return (N/A)*/
WORD	smb_mdrcnt;	value = 0 no data to read from mail slot (N/A)
BYTE	smb_msrcnt;	value = 0 no setup words to return (N/A)
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0 N/A
		bit 1 - set, no response is required (value = 1)
DWORD	smb_timeout;	(user defined) number of milliseconds to wait
WORD	smb_rsvd;	reserved
WORD	smb_pscnt;	value = 0 no parameter bytes being sent this buf
WORD	smb_psoff;	value = 0 no parameter bytes
WORD	smb_dscnt;	size of announce or req_announce
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 3

BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	(op code) value = I - Write Mail slot
WORD	smb_setup2;	(priority) priority of transaction
WORD	smb_setup3;	(class) 2 = unreliable
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_name[];	"\MAILSLOT\LANMAN" (null terminated string)
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	(announce or req_announce structure)
		(* = value of smb_dscnt)

Announcement Structure Format:

Data Type	Field	Value
WORD	op_code;	value = 1 (announce)
DWORD	services;	may both be set
		bit 0 - work station
		bit 1 - server
BYTE	vers_major;	major version number of node software
BYTE	vers_minor;	minor version number of node software
WORD	periodicity;	announcement cycle in seconds
BYTE	node_name[];	computer name of this node
BYTE	comment[];	descriptive remark

Request Announce Structure Format:

Data type	Field	Value
WORD	op_code;	value = 2 (request announce)
BYTE	node_name[];	computer name of this node

TRANSACT2_SETFILEINFO

The function code TRANSACT2_SETFILEINFO in smb_setup[0] in the primary TRANSACT2 requests identifies a request to set information for a specific file.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = 6, total number of param bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = 0. No data
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for setfileinfo
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = 6, params must be in primary request
WORD	smb_psoff;	offset (from start of SMB Hdr to parameter bytes)
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 8:- TRANSACT2_-SETFILEINFO
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_SET-FILEINFO

		function is the setfileinfo specific information in the following format.
	WORD	setfileinfo_FileHandle; (File handle)
	WORD	setfileinfo_FileInfoLevel; (Info level required)
	WORD	setfileinfo_IOFlag; (Flag: 0x0010 - Write Through; 0x0020 - No cache)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Additional FileInfoLevel dependent file system information. For level = 2, smb_data[] contains the FEAList structure to set for this file.

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	value = 4
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No params in secondary request
WORD	smb_psdisp;	value = 0. No params in secondary request
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	file handle
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Response Format (one only):

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 2
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	value = 2 parameter bytes being returned
WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
DWORD	smb_drcnt;	value = 0 no data bytes
WORD	smb_droff;	value = 0 no data bytes
WORD	smb_drdisp;	value = 0 no data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_SET-FILEINFO function response is the setfileinfo specific return information in the following format
	WORD	setfileinfo_error; (Offset into FEAList data of first error which occurred while setting the extended attributes.

TRANSACT2_QFILEINFO

The function code TRANSACT2_QFILEINFO in smb_setup[0] in the primary TRANSACT2 requests identifies a request to query information about a specific file.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = 4, total number of param

		bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = 0. No setup words to return
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for qfileinfo
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = tpscnt, all params are in primary request
WORD	smb_psoff;	offset (from start of SMB Hdr to parameter bytes)
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 7: TRANSACT2_-QFILEINFO
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_QFILE-INFO function is the qfileinfo specific information in the following format.
	WORD	qfileinfo_FileHandle; (File handle)
	DWORD	qfileinfo_FileInfoLevel; (Info level required)
BYTE	smb_pad1[];	(optional) to pad to word or

		dword boundary
BYTE	smb_data[*];	Additional FileInfoLevel dependent file system information

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No params in secondary request
WORD	smb_psdisp;	value = 0. No params in secondary request
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	file handle
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

First Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 2
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd	reserved
WORD	smb_prcnt;	value = 2 parameter bytes returned for qfileinfo

WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
DWORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_droff;	offset (from start of SMB hdr) to data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_QFILE-INFO function response is the qfileinfo specific return information in the following format
	WORD	qfileinfo_offerror; (Error offset if EA error)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt) The data block contains the requested level dependent information about the path

Subsequent Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 2
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd	reserved
WORD	smb_prcnt;	value = 0
WORD	smb_proff;	value = 0
WORD	smb_prdisp;	value = 0
DWORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_droff;	offset (from start of SMB hdr) to

		data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Return data bytes (* = value of smb_dscnt) The data block contains the requested level dependent information about the file.

TRANSACT2_SETPATHINFO

The function code TRANSACT2_SETPATHINFO in smb_setup[0] in the primary TRANSACT2 requests identifies a request to set information for a file or directory.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = 0. No setup words to return
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for setfsinfo
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = tpscnt, all params are in primary request
WORD	smb_psoff;	offset (from start of SMB Hdr to parameter bytes)
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 5 :- TRANSACT2_SETPATH-INFO
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the

		TRANSACT2_SET-PATHINFO function is the setpathinfo specific information in the following format.
	WORD	setpathinfo_PathInfoLevel;
	DWORD	setpathinfo_rsvd; (Reserved. Must be zero.)
	BYTE	setpathinfo_pathname[]; (Path name to set information on).
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Additional FileInfoLevel dependent file system information

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No params in secondary request
WORD	smb_psdisp;	value = 0. No params in secondary request
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	value = 0xffff, no handle on request
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Response Format (one only):

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 2
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd	reserved
WORD	smb_prcnt;	value = 2 parameter bytes being returned
WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
DWORD	smb_drcnt;	value = 0 no data bytes
WORD	smb_droff;	value = 0 no data bytes
WORD	smb_drdisp;	value = 0 no data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_SET-PATHINFO function response is the setpathinfo specific return information in the following format
	WORD	setpathinfo_offerror; (offset into FEAList data of first error which occurred while setting the extended attributes)

TRANSACT2_QPATHINFO

The function code TRANSACT2_QPATHINFO in smb_setup[0] in the primary TRANSACT2 requests about specific file or subdirectory.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = total number of param bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	maximum data length to return
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for qpathinfo
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = tpscnt, all params are in primary request
WORD	smb_psoff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 5 :- TRANSACT2_QPATHINFO
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*]	The parameter block for the TRANSACT2_QPATHINFO function is the qpathinfo specific information in the following format
	WORD	qpathinfo_PathInfoLevel;
	DWORD	qpathinfo_rsvd;

	BYTE	qpathinfo_PathName[];
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Additional FileInfoLevel dependent information

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No params in secondary request
WORD	smb_psdisp;	value = 0. No params in secondary request
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	value = 0xffff, no handle on request
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

First Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 2
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd	reserved
WORD	smb_prcnt;	value = 2 parameter bytes

		returned for QFSINFO
WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
DWORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_droff;	offset (from start of SMB hdr) to data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_QPATHINFO function response is the qpathinfo specific return information in the following format
	WORD	qpathinfo_offerror; (Error offset if EA error)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt) The data block contains the requested level dependent information about the path

TRANSACT2_SETFSINFO

The function code TRANSACT2_SETFSINFO in smb_setup[0] in the primary TRANSACT2 requests identifies a request to set information for a file system device.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = 2, total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = 0. No data returned
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for setfsinfo
WORD	smb_rsvd1;	reserve
WORD	smb_pscnt;	value = 4, all params are in primary request
WORD	smb_psoff;	offset (from start of SMB Hdr to parameter bytes)
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 4 :- TRANSACT2_SETFSINFO
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_SETFSINFO

		function is the setfsinfo specific information in the following format.
	WORD	setfsinfo_FSInfoLevel (Level of information provided.)
BYTE	smb_pad1;	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Level dependent file system information

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total number of parameter byes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No parameters in secondary request
WORD	smb_psdisp;	value = 0. No parameters in secondary request.
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	value = Oxffff, no handle on request
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad1;	optional to pad to word or dword boundary
BYTE	smbdata[*];	data bytes (* = value of smb_dscnt)

Response Format (one only):

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 0

WORD	smb_bdrcnt;	value = 0 no data bytes
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	value = 0 no return parameters for setfsinfo
WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
WORD	smb_drcnt;	value = 0 no data bytes
WORD	smb_droff;	value = 0 no data bytes
WORD	smb_drdisp;	value = 0 no data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	value = 0

TRANSACT2_QFSINFO

The function code TRANSACT2_QFSINFO in smb_setup[O] in the primary TRANSACT2 requests identifies a request to query information about a file system.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = 2, total parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	maximum data length to return
BYTE	smb_msrcdt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for qfsinfo
WORD	smb_rsvd;	reserved
WORD	smb_pscnt;	value = 2, params are in primary request

WORD	smb_psoff;	offset (from start of SMB header to parameter byte)
WORD	smb_dscnt;	value = 0, no data sent with qfsinfo
WORD	smb_dsoff;	value = 0, no data sent with qfsinfo
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 3: - TRANSACT2_QFSINFO
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_QFSINFO function is the qfsinfo specific information in the following format.

Response Format (One or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 0
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	value = 0, no return param bytes for QFSINFO
WORD	smb_proff;	offset (from start of SMB header) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
WORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_droff;	offset (from start of SMB header) to data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes)

		following
BYTE	smb_pad1[];	(optional) to pad to word to dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_QFSINFO function is the qfsinfo specific information in the following format
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt). The data block contains the level dependent information about the file system

TRANSACT2_FINDNEXT

The function code TRANSACT2_FINDNIEXT in smb_setup[0] in the primary TRANSACT2 request identifies a request to continue a file search started by a TRANSACT_FINDFIRST search.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	total param bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = maximum return data length
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for find next
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = tpscnt, params must be in primary request
WORD	smb_psoff;	offset (from start of SMB hdr to parameter bytes)
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB header) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 2 : - TRANSACT2_FINDNEXT
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_FINDNEXT

		function is the find next specific information in the following format:
	WORD	findnext_DirHandle; (Directory search key)
	WORD	findnext_SearchCount; (Number of entries to find)
	WORD	findnext_FileInfoLevel (Search Level)
	DWORD	findnext_ResumeKey (Server reserved resume key)
	WORD	findnext_flags; Bit 0: set - close search after this request Bit 1: set - close search if end of search reached Bit 2: set - Requester requires resume key for each entry found Bit 3: set - Continue search from last entry returned clr - Rewind search
BYTE	findnext FileName[];	Name of file to resume search from
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Additional FileInfoLevel dependent match information. For a search requiring extended attribute matching the data buffer contains the FEAList data for the search.

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total parameter bytes sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No parameters in secondary request
WORD	smb_psdisp;	value = 0. No parameters in

		secondary request
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	search handle returned from find first
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

First Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 6
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	parameter bytes returned in this buffer
WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
DWORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_droff;	offset (from start of SMB hdr) to data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the

		TRANSACT2_FINDNEXT function response is the find next specific return information in the following format
	WORD	findnext_searchcount; (Number of matching entries found)
	WORD	findnext_eos; (End of search indicator)
	WORD	findnext_offerror; (Error offset if EA error)
	WORD	findfirst_lastname; (0 - server does not require findnext_FileName[] in order to continue search. Else offset from start of returned data to filename of last found entry returned)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt) The data block contains the level dependent information about the matches found in the search. If bit 2 in the findfirst_flags is set, each returned file descriptor block will be preceded by the four byte resume key.

Subsequent Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 6
WORD	smb_tdrcnt;	value = total length of returned data buffer
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	value = 0
WORD	smb_proff;	value = 0
WORD	smb_prdisp;	value = 0
WORD	smb_drcnt;	data bytes returned this buffer
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following

BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt) The data block contains the level dependent information about the matches found in the search. If bit 2 in the findfirst_flags is set, each returned file descriptor block will be preceded by a four byte resume key

TRANSACT2_FINDFIRST

The function code TRANSACT2_FINDFIRST in smb_setup[0] in the primary TRANSACT2 request identifies a request to find the first file that matches the specified file specification.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = total number of param bytes being sent
WORD	smb_tdscnt;	total size of extended attribute list
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = maximum return data length
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 1 - 0
DWORD	smb_timeout;	value = 0. Not used for find first
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = tpscnt, params must be in primary request
WORD	smb_psoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup1;	value = 1: - TRANSACT2_FINDFIRST
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_FINDFIRST

		function is the find first specific information in the following format
	WORD	findifrst_Attribute (search attribute)
	WORD	findfirst_SearchCount;
	WORD	findfirst_flags; (find flags)
		Bit 0: set - close search after this request
		Bit 1: set - close search if end of search reached
		Bit 2: set - Requester requires resume key for each entry found
	WORD	findfirst_FileInfoLevel; (search level)
	DWORD	findfirst_rsvd;
	BYTE	findfirst_FileName[];
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	Additional FileInfoLevel dependent match information. For a search requiring extended attribute matching the data buffer contains the FEAList data for the search.

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No parameters in secondary request.
WORD	smb_psdisp;	value = 0. No parameters in secondary request.
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to

		data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	value = 0xffff, no handle on request
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

First Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	value = 10
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	parameter bytes returned in this buffer
WORD	smb_proff;	offset (from start of SMB hdr) to param bytes
WORD	smb_prdisp;	value = 0 byte displacement for param bytes
WORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no set up return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_FINDFIRST function response is the find first specific return information in the following format
	WORD	findfirst_dir_handle; (Directory search handle)
	WORD	findfirst_searchcount; (Number of matching entries found);

	WORD	findfirst_eos; (End of search indicator)
	WORD	findfirst_offerror; (Error offset if EA error)
	WORD	findfirst_lastname; (0 - server does not require findnext_Filename[] in order to continue search -- else, offset from start of returned data to filename of last found entry returned).
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt). The data block contains the level-dependent information about the matches found in the search. If bit 2 in the findfirst_flags is set, each returned file descriptor block will be preceded by a four byte resume key.

Subsequent Response Format:

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt	value = 8
WORD	smb_tdrcnt;	value = total length of return data buffer
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	value = 0
WORD	smb_proff;	value = 0
WORD	smb_prdisp;	value = 0
WORD	smb_drcnt;	data bytes returned in this buffer
WORD	smb_droff	offset (from start of SMB hdr) to data bytes
WORD	smb_drdisp;	byte displacement for these data bytes
BYTE	smb_suwcnt;	value = 0 no setup return words
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad1[];	(optional) to pad to word or

		dword boundary
BYTE	smb_data[*];	return data bytes (* = value of smb_dscnt). The data block contains the level dependent information about the matches found in the search. If bit 2 in the findfirst_flags is set, each returned file descriptor block will be preceded by a four byte resume key.

TRANSACT2_OPEN

The function code TRANSACT2 OPEN in smb_setup[O] in the primary TRANSACT2 requests identifies a request to create a file with extended attributes.

Primary Request Format:

Data type	Field	Value
BYTE	smb_wct;	value = 15
WORD	smb_tpscnt;	value = total number of param bytes being sent
WORD	smb_tdscnt;	total size of extended attribute list
WORD	smb_mprcnt;	value = maximum return parameter length
WORD	smb_mdrcnt;	value = 0. No data returned
BYTE	smb_msrcnt;	value = 0. No setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information:
		bit 0 - 0
		bit 0 - 0
DWORD	smb_timeout;	max milliseconds to wait for resource to open
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	value = tpscnt; parms must be in primary request
WORD	smb_psoff;	offset (from start of SMB hdr to parameter bytes)
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 1
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup;	value = 0:- TRANSACT2-OPEN
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parmater block for the the TRANSACT2_OPEN function is

		the open specific information in the following format.
	WORD	open_flags2;
		bit 0 - if set, return additional information
		bit 1 - if set, set single user total file lock
		bit 2 - if set, the server should notify the consumer on any action which can modify the file (delete, setattrib, rename, etc.). If not set, the server need only notify the consumer on another open request.
	WORD	open_mode; (file open mode)
	WORD	open_sattr; (search attributes)
	WORD	open_attr; file attributes (for create)
	DWORD	open_time; (create time)
	WORD	open_ofun; (open function)
	DWORD	open_size; (bytes to reserve on "create" or "truncate")
	WORD	open_rsvd; reserved (must be zero)
	BYTE	open_pathname[]; (file pathname)
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	FEAList structure for the file to be opened.

Secondary Request Format (more data - may be zero or more of these):

Data type	Field	Value
BYTE	smb_wct;	value = 9
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	total number of data bytes being sent
WORD	smb_pscnt;	value = 0. All params in primary request
WORD	smb_psoff;	value = 0. No parameters in

		secondary request
WORD	smb_psdisp;	value = 0. No parameters in secondary request
WORD	smb_dscnt;	number of data bytes being sent this buffer
WORD	smb_dsoff;	offset (from start of SMB hdr) to data bytes
WORD	smb_dsdisp;	byte displacement for these data bytes
WORD	smb_fid;	value = Oxffff, no handle on request
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_dscnt)

Response Format (one only):

Data type	Field	Value
BYTE	smb_wct;	value = 10
WORD	smb_tprcnt;	total parameter length returned
WORD	smb_tdrcnt;	value = 0 no data bytes
WORD	smb_rsvd;	reserved
WORD	smb_prcnt;	parameter bytes being returned
WORD	smb_proff;	offset (from start of SMB hdr) to parameter bytes
WORD	smb_prdisp;	value = 0 byte displacement for these param bytes
WORD	smb_drcnt;	value = 0 no data bytes
WORD	smb_droff;	value = 0 no data bytes
WORD	smb_drdisp;	value = 0 no data bytes
BYTE	smb_rsvd1;	reserved (pad above to word)
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_pad[]	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	The parameter block for the TRANSACT2_OPEN function response is the open specific return information in the following

	format
WORD	open_fid (file handle)
+WORD	open_attribute (attributes of file or device)
+DWORD	open_time (last modification time)
+DWORD	open_size (current file size)
+WORD	open_access (access permissions actually allowed)
+WORD	open_type (file type)
+WORD	open_state (state of IPC device; for example, a pipe)
WORD	open_action (action taken)
DWORD	open_fileid (server unique file id)
WORD	open_offerror (offset into FEAList data of first error which occurred while setting the extended attributes)
++DWORD	open_EAlength (Total EA length for opened file)

SetNmPHandState

This form of the pipe Transaction protocol is used to implement DosSetNmPHandState remotely.

Purpose: Set pipe-specific handle states.

The request form of this protocol should set smb_pscnt to 2 (one word parameter, the pipe state to be set). The pipe handle should be in smb_setup2. smb_dscat should be set to 0 (not writing data to pipe). smb_prcnt should be set to 0 and smb_drcnt set to 0 (not reading the pipe).

The response contains no data or parameters. If smb_err is 0, the requested state has been set on the pipe.

Pipe Handle State Bits:

5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

B * * * * * R R 0 0 0 0 0 0 0 0

Where

B	Blocking -	0 =>	reads/writes block if no data available
		1 =>	read/writes return immediately if no data
RR	Read Mode -	00 =>	Read pipe as a byte stream
		01 =>	Read messages from pipe

Note that only the read mode (byte versus message) and blocking/nonblocking mode of a named pipe can be changed. Some combinations of parameters may be illegal and will be rejected as an error.

QNmPHandState

This form of the pipe Transaction protocol is used to implement DosQNmPHandState remotely.

Purpose: Return pipe-specific state information.

The request form of this protocol should set smb_pscnt to 0 (no parameters). The pipe handle should be in smb_setup2. smb_dscnt should be set to 0 (not writing data to pipe). smb_prcnt should be set to 2 (requesting the 1 word of information about the pipe) and smb_drcnt set to 0 (not reading the pipe).

The response will return the 1 parameter (smb_rprcnt = 1) of pipe state information.

Pipe Handle State Bits:

5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

B E * * T T R R |---------Icount--|

where:

B	Blocking - 0 =>	reads/writes block if no data available
E	Endpoint - 0 =>	reads/writes return immediately if no data
TT	Type of pipe - 00 =>	pipe is a bytes stream pipe
		01 => pipe is a message pipe
RR	Read Mode - 00 =>	Read pipe as a byte stream
	01 =>	Read messages from pipe
ICount	8 bit count to control pipe instancing (N/A)	

The E (endpoint) bit is 0 because this handle is the client end of the pipe.

The values returned are those originally established by Open or a subsequent SetNmPHandState.

PeekNmPipe

This form of the pipe Transaction protocol is used to implement DosPeekNmPipe remotely.

Purpose: Read pipe without removing the read data from the pipe.

PeekNmPipe acts like Read except as follows:

- The bytes read are not removed from the pipe.
- The peek may return only part of a message (that part currently in the pipe), even if the size of the peek would accommodate the whole message.
- PeekNmPipe never blocks, regardless of the blocking mode.
- Additional information about the status of the pipe and remaining data are returned. The caller can use this, for example, to determine whether the peek returned all of the current message or whether the pipe is at EOF (pipe is at EOF when there are no bytes left in the pipe and Status is Closing or Disconnected).

The request format of this protocol should set smb_pscnt to 0. The pipe handle being "peeked" should be set in smb_setup2. smb_dscnt should be set to 0 (not writing data to pipe). smb_prcnt should be set to 6 (requesting the 3 words of information about the pipe) and smb_drcnt set to the number of bytes to "peek".

The response will return the 3 parameters (smb_rprcnt = 6), smb_rdrcnt will be set to the number of bytes "peeked" and smb_bcc will be set to 6 (the 3 param words) + the amount of data bytes being returned in the first buffer. Subsequent responses would have smb_rprcnt set to 0, smb_rdrcut set to the data bytes remaining and smb_bcc indicating the number of data bytes being returned in each buffer.

The following defines the format of the parameter words:

WORD	bytes remaining in the pipe
WORD	bytes remaining in the current message
WORD	pipe status

1. Disconnected (disconnected by server)
2. Listening (N/A not returned on consumer end of pipe).
3. Connected (connection to server OK)
4. Closing (server end of pipe closed)

WaitNmPipe

This form of the pipe Transaction protocol waits for the availability of a named pipe instance. It is used to implement the DosWaitNmPipe call on a remote pipe.

DosWaitNmPipe allows an application to wait for a pipe when all available instances are currently busy. This protocol may be used when the error ERRpipebusy is returned from an Open (pipe) protocol attempt.

The server will wait up to smb_timeout milliseconds for a pipe of the name given to become available. Note that although the timeout is specified in milliseconds (in order to match the OS/2 system calls), by the time that the timeout occurs and the consumer receives the timed out protocol much more time than specified may have occurred.

This form of the transaction protocol sends no data or parameter bytes. The response also contains no data or parameters. If smb_err is 0, the requested named pipe may now be available.

Note that this protocol does NOT reserve the pipe, thus all waiting programs may race to get the pipe now available. The losers will again get ERRpipebusy on the Open attempt.

WaitNmPipe uses priority in smb_setup2. The priority values range from 0 (use server default) to 0x3FF (highest priority). The priority passed in smb_setup2 from a LANMAN consumer will be the value as returned from a DosGetPrty OS/2 system call. The server may use the priority in determining which process to notify when a pipe becomes available.

CallNmPipe

This protocol is used to implement DosCallNmPipe remotely.

This transaction has the combined effect on a named pipe of Open, TransactNmPipe, Close. It provides a very efficient means of implementing Remote Procedure Call (RPC) interfaces between processes.

This form of the transaction protocol sends no parameter bytes, thus the bytes to be written to the pipe are sent as data bytes (smb_databytes) and the bytes read from the pipe are returned as data bytes (smb_databytes).

The number of bytes being written is defined by smb_dscnt and the max number of bytes to returned is defined by smb_drcnt.

On the response smb_rprcnt is 0 (no param bytes to return), smb_rdcnt indicates the amount of data-bytes being returned in total and smb_bcc identifies the amount of data being returned in each buffer.

Note that the full form of the Transaction protocol can be used to write and read up to 65,535 bytes each utilizing the secondary requests and responses.

CallNmPipe uses priority in smb_setup2. The priority values range from 0 (use server default) to 0x3FF (highest priority). The priority passed in smb_setup2 from a LANMAN consumer will be the value as returned from DosGetPrty OS/2 system call. The server may use the priority in determining which process to run next when a pipe becomes available.

Named Pipe Transaction Protocol

Named pipes require reliable delivery; thus this Transaction protocol is sent/received only on an established VC.

A named pipe transaction is used to wait for the specified named pipe to become available (WaitNmPipe) or perform a logical "open -> write -> read -> close" of the pipe (CallNmPipe).

Other Standard protocols (Open, Read, Write, Close, etc.) may also be used to access Named pipes when pipe is being accessed like a "standard" file (a file handle is being used).

The identifier "\PIPE\<name>" denotes a named pipe transaction, where the <name> is the pipe name to apply the transaction against.

Note that the named pipe transaction name "\PIPE\LANMAN" is reserved for use by the LAN Manager.

Data type	Field	Value
BYTE	smb_wct;	value = 16
WORD	smb_tpscnt;	total number of parameter bytes being sent
WORD	smb_tdscnt;	size of data to be written to pipe (if any)
WORD	smb_mprcnt;	max number of parameter bytes to return
WORD	smb_mdrcnt;	value
BYTE	smb_msrcnt;	value = 0 - max number of setup words to return
BYTE	smb_rsvd;	reserved (pad above to word)
WORD	smb_flags;	additional information
		bit 0 - if set, also disconnect TID in smb_tid
		bit 1 - not set, response is required
DWORD	smb_timeout;	(user defined) number of milliseconds to wait
WORD	smb_rsvd1;	reserved
WORD	smb_pscnt;	number of parameter bytes being sent this buffer
WORD	smb_psoff;	offset (from start of SMB hdr) to data bytes
BYTE	smb_suwcnt;	value = 2
BYTE	smb_rsvd2;	reserved (pad above to word)
WORD	smb_setup;	function (defined below)
		0x54 CallNmPipe - open/write/read/ close pipe
		0x53 WaitNmPipe - wait for pipe to be nonbusy

		0x23 PeekNmPipe - read but don't remove data 0x21 QNmPHandState - query pipe handle modes 0x01- SetNmPHandState - set pipe handle modes 0x22 - QNmPipeInfo - query pipe attributes 0x26 - TransactNmPipe - write/read operation on file 0x11 - RawReadNmPipe - read pipe in "raw" (non message mode) 0x31 - RawWriteNmPipe - write pipe "raw" (non message mode)
WORD	smb_setup;	FID (handle) of pipe (if needed), or priority
WORD	smb_bcc;	total bytes (including pad bytes) following
BYTE	smb_name[]	"\PIPE\<name>0"
BYTE	smb_pad[];	(optional) to pad to word or dword boundary
BYTE	smb_param[*];	param bytes (* = value of smb_prcnt)
BYTE	smb_pad1[];	(optional) to pad to word or dword boundary
BYTE	smb_data[*];	data bytes (* = value of smb_drcnt)

The NT Cancel Command

The NT Cancel command allows a client to cancel an in-progress request. The request to be canceled is identified by matching the UID/TID/PID/MID combination in the header of the original SMB with those in the Cancel SMB.

If the specified request is still in progress at the server when the Cancel request is received, the server cancels the request and sends a response to that request with a distinguished status code. The server does not send a response to the cancel SMB.

The command code for NT Cancel is SMB_COM_NT_CANCEL.

Request Format:

Data type	Field	Value
UCHAR	WordCount;	Must contain the value 0
USHORT	ByteCount;	Must contain the value 0.

Response Format:

Data type	Field	Value
UCHAR	WordCount;	Must contain the value 0.
USHORT	ByteCount;	Must contain the value 0.

The NT Create and X Command

The NT Create And X command is used to create or open a file or a directory. When EAs or an SD must be applied to the file, the NT Transact function Create With SD Or EAs is used.

The command code for NT Create And X is SMB_COM_NT_CREATE_ANDX.

Note: The values for all the tables in this topic are included beneath the table.

Request Format:

Data Type	Field	Value
UCHAR	WordCount;	All values in this table explained below.
UCHAR	AndXCommand;	
UCHAR	AndXReserved;	
USHORT	AndXOffset;	
UCHAR	Reserved1;	
USHORT	NameLength;	
ULONG	Flags;	
ULONG	RootDirectoryFid;	
ACCESS_MASK	DesiredAccess;	
LARGE_INTEGER	AllocationSize;	
ULONG	FileAttributes;	

```
ULONG                   ShareAccess;
ULONG                   CreateDisposition;
ULONG                   CreateOptions;
ULONG                   ImpersonationLevel;
UCHAR                   SecurityFlags;
USHORT                  ByteCount;
ASCII_OR_UNICODE        Name[];
```

WordCount

Must contain the value 24.

AndXCommand

Specifies the command code for the next command in the chain, if any.

AndXReserved

Is a reserved field and should be zero.

AndXOffset

Specifies the offset to the next command in the chain.

Reserved1

Is a reserved field, and should be zero. Its purpose is to align the following fields on natural boundaries.

NameLength

Specifies the length of the Name field, in bytes.

Note that this structure can't handle very long file names (approaching 4K), because we don't allow the name to be sent as data. The NT Create With SD Or EAs transaction must be used for very long names.

Flags

Specifies the following flags:

NT_CREATE_REQUEST_OPLOCK

When set, the client is requesting an oplock.

NT_CREATE_REQUEST_OPBATCH

When set, the client is requesting a batch mode oplock.

NT_CREATE_OPEN_TARGET_DIR

Equivalent to IO_OPEN_TARGET_DIRECTORY in the NT I/O system.

RootDirectoryFid

Specifies the FID for a previously opened directory. If this field is not zero, the file name specified in Name is interpreted relative to the specified root directory. If this field is zero, the file name is interpreted relative to the root of the shared resource.

Note that RootDirectoryFid is subject to change format. While currently 32 bits, it may be changed to a 16 bit field (consistent with Fid fields in other SMBs) and a 16 bit reserved field.

DesiredAccess

Specifies the type of access that the caller requires to the file. The following access types are defined:

DELETE

The file may be deleted.

READ_CONTROL

The SD and ownership information associated with the file may be read.

WRITE_DAC

The discretionary SD associated with the file may be written.

WRITE_OWNER

Ownership information associated with the file may be written.

FILE_READ_DATA

Data may be read from the file.

FILE_WRITE_DATA

Data may be written to the file.

FILE_EXECUTE

Data may be faulted into memory from the file via paging I/O.

FILE_APPEND_DATA

Data may only be appended to the file.

FILE_READ_ATTRIBUTES

File attributes flags may be read.

FILE_WRITE_ATTRIBUTES

File attributes flags may be written.

FILE_READ_EA

Extended attributes associated with the file may be read.

FILE_WRITE_EA

Extended attributes associated with the file may be written.

If the file being created or opened is a directory file, as specified in the CreateOptions field, then the following additional types of access may be requested:

FILE_LIST_DIRECTORY

Files in the directory may be listed.

FILE_TRAVERSE

The directory may be traversed. That is, it may be in the pathname of a file.

FILE_READ_DATA, FILE_WRITE_DATA, FILE_EXECUTE, and FILE_APPEND_DATA accesses are not valid when creating or opening a directory file.

AllocationSize

Specifies the initial allocation size of the file, in bytes. This field is ignored unless the file is created, overwritten, or superseded.

FileAttributes

Specifies the file attributes for the file. Any combination of flags is acceptable except that all other flags override the normal file attribute, FILE_ATTRIBUTE_NORMAL. File attributes are only applied to the file if it is created, superseded, or, in some cases, overwritten. See the description in the text below for more details.

The following attribute flags are defined:

FILE_ATTRIBUTE_NORMAL

A normal file should be created.

FILE_ATTRIBUTE_READONLY

A read-only file should be created.

FILE_ATTRIBUTE_HIDDEN

A hidden file should be created.

FILE_ATTRIBUTE_SYSTEM

A system file should be created.

FILE_ATTRIBUTE_ARCHIVE

The file should be marked so that it will be archived.

FILE_ATTRIBUTE_CONTROL

A control file should be created.

ShareAccess

Specifies the type of share access that the caller would like to the file. The following share access flags are defined:

FILE_SHARE_READ

Other open operations may be performed on the file for read access.

FILE_SHARE_WRITE

Other open operations may be performed on the file for write access.

FILE_SHARE_DELETE

Other open operations may be performed on the file for delete access.

CreateDisposition

Specifies the actions to be taken if the file does or does not already exist. The following disposition values are defined:

FILE_SUPERSEDE

Indicates that if the file already exists then it should be superseded by the specified file. If it does not already exist then it should be created.

FILE_CREATE

Indicates that if the file already exists then the operation should fail. If the file does not already exist then it should be created.

FILE_OPEN

Indicates that if the file already exists it should be opened rather than creating a new file. If the file does not already exist then the operation should fail.

FILE_OPEN_IF

Indicates that if the file already exists, it should be opened. If the file does not already exist then it should be created.

FILE_OVERWRITE

Indicates that if the file already exists it should be opened and overwritten. If the file does not already exist then the operation should fail.

FILE_OVERWRITE_IF

Indicates that if the file already exists it should be opened and overwritten. If the file does not already exist then it should be created.

CreateOptions

Specifies the options that should be used when creating or opening the file. The following options are defined:

FILE_DIRECTORY_FILE

On a create operation, indicates that the file being created is a directory file. On an open operation, indicates that the file being opened must be a directory file.

FILE_NON_DIRECTORY_FILE

On a create operation, indicates that the file being created is not a directory file. On an open operation, indicates that the file being opened must not be a directory file.

FILE_WRITE_THROUGH

Indicates that services that write data to the file must actually write the data to the file before the operation is considered to be complete.

Note that the FILE_NO_INTERMEDIATE_BUFFERING option is not exported via the SMB protocol. The NT redirector

should convert this option to FILE_WRITE_THROUGH.

FILE_SEQUENTIAL_ONLY

Indicates that the file will only be accessed sequentially.

FILE_OPEN_LINK

Indicates that if the last component of the file name is a symbolic link that the link should be opened rather than chased.

FILE_OPEN_UNKNOWN_OBJECT

Indicates that if an object that is unknown to the file system is encountered, that it should be opened rather than skipped over.

FILE_NO_EA_KNOWLEDGE

Indicates that the client does not understand extended attributes.

FILE_EIGHT_DOT_THREE_ONLY

Indicates that the client understands only 8.3 style filenames.

FILE_RANDOM_ACCESS

Hints that the file is going to be accessed randomly, permitting the server to optimize its behavior for that case.

FILE_DELETE_ON_CLOSE

Indicates that the file is to be deleted when closed.

ImpersonationLevel

Specifies the security impersonation level. Must be one of the following values. See the Distributed System Architecture, Subject Security Context Specification for a description of their meanings.

- SecurityAnonymous
- SecurityIdentification
- SecurityImpersonation
- SecurityDelegation

SecurityFlags

Specifies the following security flags. See the Distributed System Architecture, Subject Security Context Specification for a detailed description of their meanings. The following flag values are defined:

SMB_SECURITY_DYNAMIC_TRACKING

SMB_SECURITY_EFFECTIVE_ONLY

ByteCount

Specifies the length of the byte parameters for the request.

Name

Supplies the name of the file to be created or opened. The name is in either ASCII format or Unicode format, depending on the state of flag SMB_FLAGS2_UNICODE of Flags2 in the SMB header. The client may use Unicode only if the server indicates in the Negotiate response that it supports Unicode. Note that the name string is not NULL-terminated.

Response Format:

Data type	Field
UCHAR	WordCount;
UCHAR	AndXCommand;
UCHAR	AndXReserved;
USHORT	AndXOffset;
UCHAR	OplockLevel;

USHORT	Fid;
ULONG	CreateAction;
TIME	CreationTime;
TIME	LastAccessTime;
TIME	LastWriteTime;
TIME	ChangeTime;
ULONG	FileAttributes;
LARGE_INTEGER	AllocationSize;
LARGE_INTEGER	EndOfFile;
USHORT	FileType;
USHORT	DeviceState;
BOOLEAN	Directory;
USHORT	ByteCount;

WordCount

Must contain the value 34.

AndXCommand

Specifies the command code for the next command in the chain, if any.

AndXReserved

Is a reserved field and should be zero.

AndXOffset

Specifies the offset to the next command in the chain.

OplockLevel

Specifies the level of opportunistic lock granted to the opener.

FID

Specifies the identifier for the open file instance.

CreateAction

Specifies the action taken with respect to creating or opening the file. This field is valid only if the operation was successful. The following action codes are defined:

FILE_SUPERSEDED

The file existed and was superseded.

FILE_CREATED

The file did not exist and was created.

FILE_OPENED

The file existed and was opened.

FILE_OVERWRITTEN

The file existed and was overwritten.

FILE_EXISTS

The file existed.

FILE_DOES_NOT_EXIST

The file does not exist.

CreationTime

Specifies the time the file was created.

LastAccessTime

Specifies the time the file was last accessed.

LastWriteTime

Specifies the time the file was last written.

ChangeTime

Specifies the time the file was last changed.

FileAttributes

Specifies the file's attributes. For a description of this field, see the description of FileAttributes in the request format above.

AllocationSize

Specifies the number of bytes allocated to the file.

EndOfFile

Specifies the end-of-file offset for the file.

FileType

Specifies the type of the file. Possible values are the same as for the like-named field in the SMB Open And X command response format:

FileTypeDisk

Indicates a disk file or directory as defined in the FileAttributes field.

FileTypeByteModePipe

Indicates a named pipe.

FileTypeMessageModePipe

Indicates a named pipe in message mode.

FileTypePrinter

Indicates a printer device.

FileTypeCommDevice

Indicates a communications device.

DeviceState

If FileType is FileTypeByteModePipe or FileTypeMessageModePipe, this field indicates the state of the IPC device. Otherwise, this field contains garbage. When valid, possible flag values are the same as for the like-named field in the SMB Open And X command response format:

DeviceStateBlocking

If set, indicates that reads and writes return immediately if no data is available. If clear, reads and writes block if no data is available.

DeviceStateEndPoint

If set, indicates the server end of the pipe. If clear, indicates the consumer end of the pipe.

DeviceStatePipeType

This is a two-bit field. 00 indicates a byte stream pipe. 01 indicates a message pipe. Other values are undefined.

DeviceStateReadMode

This is a two-bit field. 00 indicates that the pipe is to be read as a byte stream. 01 indicates that messages are to be read from the pipe. Other values are undefined.

DeviceStateIcount

This is an eight-bit count to control pipe instancing, and is currently not applicable.

Directory

Indicates whether the file is a directory.

Detailed Description

The NT Create And X command service either causes a new file (or directory) to be created, or it opens an existing file or device. The action taken is dependent on the name of the object being opened, whether the object already existed, and the specified create disposition value. A file handle is returned that can be used by subsequent service calls to manipulate the file itself or the data within the file.

If FILE_APPEND_DATA is the only desired-access flag specified, then the caller can only write to the end of the file. Any offset information on writes to the file is ignored. The file will automatically be extended as necessary for these types of write operations.

Specifying the FILE_WRITE_DATA desired-access flag for a file also allows writes beyond the end of the file to occur. The file is also automatically extended for these types of writes as well.

Access to a file may be shared among separate cooperating processes or threads by requesting that the file system open the file for shared access. This is accomplished through the flags in the ShareAccess field. Provided that both file openers have the privilege to access the file in the specified manner, the file can be successfully opened and shared. If the caller does not specify FILE_SHARE_READ, FILE_SHARE_WRITE, or FILE_SHARE_DELETE, then no other open operations may be performed on the file.

In order for the file to be successfully opened, the requested access mode to the file must be compatible with the way in which other opens to the file have been made. That is, the desired access mode to the file must not conflict with the accesses that other openers of the file have disallowed.

The **FILE_SUPERSEDE** disposition value specifies that if the file does not already exist, it is to be created. If the file already exists, then it should be superseded. Superseding a file requires that the accessor have delete access to the existing file. That is, the existing file is effectively deleted and then recreated. This implies that if someone else already has the file open, they have specified that the file may be deleted by another file opener. This is done by specifying ShareAccess with the **FILE_SHARE_DELETE** flag set.

The **FILE_OVERWRITE_IF** disposition value is much like the **FILE_SUPERSEDE** disposition value. If the file exists, then it will be overwritten; if it does not already exist then it will be created. Overwriting a file is semantically equivalent to a supersede operation except that it requires write access to the file rather than delete access. That is, the requester must have write access to the file and if someone else already has the file open, they must have specified that the file may be written by another file opener. This is done by specifying a ShareAccess parameter with the **FILE_SHARE_WRITE** flag set. Another difference between an overwrite and a supersede is that the specified file attributes are logically OR'd with those already on the file. That is, the caller may not turn off any flags already set in the attributes but may turn others on.

The **FILE_OVERWRITE** disposition value performs exactly the same operation as a **FILE_OVERWRITE_IF**, except that if the file does not already exist the operation will fail.

The **FILE_OPEN_LINK** option specifies that if the last component of a pathname is a symbolic link, then the link file itself should be opened rather than chasing it. A symbolic link is a file identified by the following characteristics:

- The file's **FILE_ATTRIBUTE_CONTROL** attribute flag is set.
- The file has an EA whose name is **".FAMILY_IDS"** and whose value has a type of *EAT_FAMILY*, a family count of one, and a family ID of 42.
- The file has an EA whose name is **"SYMBOLIC_LINK_VALUE"** and whose value is the fully qualified pathname to the target file.

The **FILE_OPEN_LINK** option may be used to open a file that represents a DFS exit path as well, provided that the file is the last component of the pathname. A DFS exit path is a file identified by the following characteristics:

- The file's **FILE_ATTRIBUTE_CONTROL** attribute flag is set.
- The file has an EA whose name is **".FAMILY_IDS"** and whose value has a type of **EAT_FAMILY**, a family count of one, and a family ID of 69.

The **FILE_OPEN_UNKNOWN_OBJECT** option specifies that if the file system encounters a file in the file name path that has its **FILE_ATTRIBUTE_CONTROL** attribute flag set, but whose other characteristics do not match one of the symbolic link or DFS exit path descriptions above, then the file should be opened rather than continuing the pathname search. If this type of file is

encountered and the option is specified, then the response field *CreateAction* is set to **FILE_OPENED_UNKNOWN_OBJECT**. The *Query File Information* command may be used to obtain the remainder of the unparsed file name string.

When opening a named pipe file, the Name field of the request should contain only the name of the relative name of the pipe. This contrasts the OpenAndX and Open2 SMBs in which the pipe name is prepended by the string "\PIPE\".

ERRbadfunc

Error code 1.

Invalid function. The server OS did not recognize or could not perform a system call generated by the server, e.g., set the DIRECTORY attribute on a data file, invalid seek mode. [EINVAL]

ERRbadfile

Error code 2.

File not found. The last component of a file's pathname could not be found. [ENOENT]

ERRbadpath

Error code 3.

Directory invalid. A directory component in a pathname could not be found. [ENOENT]

ERRnofids

Error code 4.

Too many open files. The server has no file handles (fids) available. [EMFILE]

ERRnoaccess

Error code 5.

Access denied, the requester's context does not permit the requested function. This includes the following conditions. [EPERM]

- duplicate name errors
- invalid rename command
- write to fid open for read only
- read on fid open for write only
- attempt to open read-only file for write
- attempt to delete read-only file
- attempt to set attributes of a read only file
- attempt to create a file on a full server
- directory full
- attempt to delete a non-empty directory
- invalid file type (e.g., file commands on a directory)

ERRbadfid

Error code 6.

Invalid file handle. The file handle specified was not recognized by the server. [EBADF]

ERRbadmcb

Error code 7.

Memory control blocks destroyed. [EREMOTEIO]

ERRnomem

Error code 8.

Insufficient server memory to perform the requested function. [ENOMEM]

ERRbadmem

Error code 9.

Invalid memory block address. [EFAULT]

ERRbadenv

Error code 10.

Invalid environment. [EREMOTEIO]

ERRbadformat

Error code 11.

Invalid format. [EREMOTEIO]

ERRbadaccess

Error code 12.

Invalid open mode.

ERRbaddata

Error code 13.

Invalid data (generated only by IOCTL calls within the server). [E2BIG]

ERR 14

Reserved.

ERRbaddrive

Error code 15.

Invalid drive specified. [ENXIO]

ERRremcd

Error code 16.

A Delete Directory request attempted to remove the server's current directory. [EREMOTEIO]

ERRdiffdevice

Error code 17.

Not same device (e.g., a cross volume rename was attempted) [EXDEV]

ERRnofiles

Error code 18.

A File Search command can find no more files matching the specified criteria.

ERRbadshare

Error code 32.

The sharing mode specified for a non-compatibility mode Open conflicts with existing FIDs on the file. [ETXTBSY]

ERRlock

Error code 33.

A Lock request conflicted with an existing lock or specified an invalid mode, or an Unlock request attempted to remove a lock held by another process. [EDEADLOCK]

ERRfilexists

Error code 80.

The file named in a Create Directory or Make New File request already exists. The error may also be generated in the Create and Rename transactions. [EEXIST]

ERRerror

Error code 1.

Non-specific error code. It is returned under the following conditions:

- resource other than disk space exhausted (e.g., TIDs)
- first command on VC was not negotiate
- multiple negotiates attempted
- internal server error [ENFILE]

ERRbadpw

Error code 2.

Bad password - name/password pair in a Tree Connect is invalid.

ERRbadtype

Error code 3.

Reserved.

ERRaccess

Error code 4.

The requester does not have the necessary access rights within the specified TID context for the requested function. [EACCES]

ERRinvnid

Error code 5.

The tree ID (tid) specified in a command was invalid.

ERRinvnetname

Error code 6.

Invalid name supplied with tree connect.

ERRinvdevice

Error code 7.

Invalid device - printer request made to non-printer connection or non-printer request made to printer connection.

ERRqfull

Error code 49.

Print queue full (files) -- returned by open print file.

ERRqtoobig

Error code 50.

Print queue full -- no space.

ERRqeof

Error code 51.

EOF on print queue dump.

ERRinvpfid

Error code 52.

Invalid print file FID.

ERRpaused

Error code 81.

Server is paused.

ERRmsgoff

Error code 82.

Not receiving messages.

ERRnoroom

Error code 83.

No room to buffer message.

ERRnosupport

Error code 87.

Too many remote user names.

ERRnowrite

Error code 19.

Attempt to write on write-protected diskette. [EROFS]

ERRbadunit

Error code 20.

Unknown unit. [ENODEV]

ERRnotready

Error code 21.

Drive not ready. [EUCLEAN]

ERRbadcmd

Error code 22.

Invalid disk command.

ERRdata

Error code 23.

Data error (CRC). [EIO]

ERRbadreq

Error code 24.

Bad request structure length. [ERANGE]

ERRseek

Error code 25.

Seek error.

ERRbadmedia

Error code 26.

Unknown media type.

ERRbadsector

Error code 27.

Sector not found.

ERRnopaper

Error code 28.

Printer out of paper.

ERRwrite

Error code 29.

Write fault.

ERRread

Error code 30.

Read fault.

ERRgeneral

Error code 31.

General failure.

Click Help Topics for a list of Help topics.