Network Working Group
Request for Comments: DRAFT
Category: Informational
Title: draft-heizer-cifs-v1-spec-01.txt

I. Heizer
P. Leach
D. Perry
Microsoft
June 30, 1996

Common Internet File System Protocol (CIFS/1.0)

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or made obsolete by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress".

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet-Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Distribution of this document is unlimited. Please send comments to the authors at <cifs@microsoft.com>. Discussion of CIFS is on the mailing list <cifs@lists.msn.com>; subscribe by sending a message to <ListAdmin@lists.msn.com> with a body of "subscribe CIFS you@your.company.com". There is a CIFS home page at <ftp://ftp.microsoft.com/developr/drg/SMB-info/cifs.html>.

Abstract

This document describes the CIFS file sharing protocol, version 1.0. Client systems use this protocol to request file and print services from server systems over a network. It is based on the Server Message Block protocol widely in use by personal computers and workstations running a wide variety of operating systems.

Table of Contents

1. Introduction

This document describes the file and print sharing protocol for a proposed Common Internet File System (CIFS). CIFS is intended to provide an open cross-platform mechanism for client systems to request file and print services from server systems over a network. It is based on the standard Server Message Block (SMB) protocol widely in use by personal computers and workstations running a wide variety of operating systems. An earlier version of this protocol was documented as part of the X/OPEN (now Open Group) CAE series of standards [7]; this document updates the specification to include the latest shipping versions, and is published to allow the creation of implementations that interoperate with those implementations.

Use of the Internet and the World Wide Web has been characterized by read-only access. Existing protocols such as FTP are good solutions for one-way file transfer. However, new read/write interfaces will become increasingly necessary as the Internet becomes more interactive and collaborative. Adoption of a common file sharing protocol having modern semantics such as shared files, byte-range locking, coherent caching, change notification, replicated storage, etc. would provide important benefits to the Internet community.

1.1 Summary of features

The protocol supports the following features:

- File and printer access
- File and record locking
- Safe caching, read-ahead, and write-behind
- File change notification
- Protocol version negotiation
- Extended attributes
- Distributed replicated virtual volumes
- Server name resolution using DNS
- Batched requests
- Operates over connection-oriented or connection-less transports
- Unicode file names

1.1.1 File and printer access

The protocol supports the usual set of file operations: open, close, read, write, and seek. Opening a printer resources as a file and writing to it causes a print job to be queued.

1.1.2 File and record locking

The protocol supports file and record locking, as well as unlocked access to files. Applications that lock files can not be improperly interfered with by applications that do not; once a file or record is locked, non-locking applications are denied access to the file.

1.1.3 Safe caching, read-ahead, and write-behind

The protocol supports caching, read-ahead, and write-behind, even for unlocked files, as long as they are safe. All these optimizations are safe as long as only one client is accessing a file; read-caching and read-ahead are safe with many clients accessing a file as long as all are just reading. If many clients are writing a file simultaneously, then none are safe, and all file operations have to go to the server. The protocol notifies all clients accessing a file of changes in the number and access mode of clients accessing the file, so that they can use the most optimized safe access method.

1.1.4 File change notification

Applications can register with a server to be notified if and when file or directory contents are modified. They can use this to (for example) know when a display needs to be refreshed, without having to constantly poll the server.

1.1.5 Protocol version negotiation

There are several different versions and sub-versions of this protocol; a particular version is referred to as a *dialect*. When two machines first come into network contact they negotiate the dialect to be used. Different dialects can include both new messages as well as changes to the fields and semantics of existing messages in other dialects.

1.1.6 Extended attributes

In addition to many built-in file attributes, such as creation and modification times, non-file system attributes can be added by applications, such as the author's name, content description, *etc.*

1.1.7 Distributed replicated virtual volumes

The protocol supports file system subtrees which look like to clients as if they are on a single volume and server, but which actually span multiple volumes and servers. The files and directories of such a subtree can be physically moved to different servers, and their names do not have to change, isolating clients from changes in the server configuration. These subtrees can also be transparently replicated for load sharing and fault tolerance. When a client requests a file, the protocol uses referrals to transparently direct a client to the server that stores it.

1.1.8 Server name resolution via DNS

The protocol supports resolving server names using the DNS, permitting access to the file systems of other organizations over the Internet, or hierarchical organization of servers' names within an organization. Earlier versions of the protocol only supported a flat server name space.

1.1.9 Batched requests

The protocol supports the batching of multiple requests into a single message, in order to minimize round trip latencies, even when a later request depends on the results of an earlier one.

2. Protocol Operation Overview

In order to access a file on a server, a client has to:

- Parse the full file name to determine the server name, and the relative name within that server.
- Resolve the server name to a transport address (this may be cached)
- Make a connection to the server (if using a connection-oriented transport and no connection has yet been made)
- Exchange SMB messages (see below for an example)

This process may be repeated as many times as desired. Once the connection has been idle for a while, it may be torn down.

2.1 Server Name Determination

How the client determines the name of the server and the relative name within the server is outside of the scope of this specification. However, just for expository purposes, here are three examples.

In the URL "file://fs.megacorp.com/users/fred/stuff.txt", the client could take the part between the leading double slashes and the next slash as the server name and the remainder as the relative name -- in this example "fs.megacorp.com" and "/users/fred/stuff.txt", respectively.

In the path name "\\corpserver\public\policy.doc" the client could take the part between the leading double backslashes and the next slash as the server name, and the remainder as the relative name -- in this example, "corpserver" and "\public\policy.doc" respectively.

In the path name "x:\policy.doc" the client could use "x" as an index into a table that contains a server name and a file name prefix. If the contents of such a table for "x" were "corpserver" and "\public", then the server name and relative name would be the same as in the previous example.

2.2 Server Name Resolution

Once the server name has been determined, then the client needs to resolve that name to a transport address. This specification defines three ways of doing so: using the Domain Name System (DNS) [1,2], NETBIOS name resolution (see RFC 1001 and RFC 1002 [3,4]), or IPX naming (see appendix B). Which method is used is configuration dependent; the default is DNS to encourage interoperability over the Internet. The name resolution mechanism used will place constraints on the form of the server name. In the case of NETBIOS, the server name must be 15 characters or less, and be upper case.

The server name can also be specified as the string form an IPv4 address in the usual dotted notation, *e.g.*, "157.33.135.101" In this case, "resolution" consists of converting to the 32 bit IPv4 address.

2.3 Sample Message Flow

The following illustrates a typical message exchange sequence for a client connecting to a user level server, opening a file, reading its data, closing the file, and disconnecting from the server. Note: using the SMB request batching mechanism (called the

"AndX" mechanism), the second to sixth messages in this sequence can be combined into one, so there are really only three round trips in the sequence, and the last one can be done asynchronously by the client.

Client Command ==========	Server Response ==========
SMB_COM_NEGOTIATE	Must be the first message sent by client to the server. Includes a list of SMB dialects supported by the client. Server response indicates which SMB dialect should be used.
SMB_COM_SESSION_SETUP_ANDX	Transmits the user's name and credentials to the server for verification. Successful server response has Uid field set in SMB header used for subsequent SMBs on behalf of this user.
SMB_COM_TREE_CONNECT	Transmits the name of the disk share the client wants to access. Successful server response has Tid field set in SMB header used for subsequent SMBs referring to this resource.
SMB_COM_OPEN	Transmits the name of the file, relative to Tid, the client wants to open. Successful server response includes a file id (Fid) the client should supply for subsequent operations on this file.
SMB_COM_READ	Client supplies Tid, Fid, file offset, and number of bytes to read. Successful server response includes the requested file data.
SMB_COM_CLOSE	Client closes the file represented by Tid and Fid. Server responds with success code.
SMB_COM_TREE_DISCONNECT	Client disconnects from resource represented by Tid.

2.4 Message Format

Clients exchange messages with a server to access resources on that server. These messages are called Server Message Blocks (SMBs), and every SMB message has a common format. Multi-byte values are always transmitted least significant byte first.

```
typedef unsigned char UCHAR;           // 8 unsigned bits
typedef unsigned short USHORT;         // 16 unsigned bits
typedef unsigned long ULONG;           // 32 unsigned bits

typedef struct {
    ULONG LowPart;
    LONG HighPart;
} LARGE_INTEGER;                       // 64 bits of data

typedef struct {
    ULONG LowTime;
    LONG HighTime;
} TIME;

typedef struct  {
    UCHAR Protocol[4];                 // Contains 0xFF,'SMB'
    UCHAR Command;                     // Command code
    union {
        struct {
            UCHAR ErrorClass;          // Error class
            UCHAR Reserved;            // Reserved for future use
            USHORT Error;                 // Error code
        } DosError;
        ULONG NtStatus;                // NT-style 32-bit error code
    } Status;
    UCHAR Flags;                       // Flags
    USHORT Flags2;              // More flags
    union {
        USHORT Pad[6];                    // Ensure this section is 12
                                                  // bytes long
        struct {
            USHORT PidHigh;            // High part of PID
                                                  // (NT Create And X)
            ULONG  Unused;             // Not used
            USHORT Sid;                // Session ID
            USHORT SequenceNumber;     // Sequence number
          } Connectionless;               // IPX
          };
    USHORT Tid;                           // Tree identifier
    USHORT Pid;                           // Caller's process id
    USHORT Uid;                           // Unauthenticated user id
    USHORT Mid;                           // multiplex id
    UCHAR  WordCount;                     // Count of parameter words
    USHORT ParameterWords[ WordCount ];   // The parameter words
    USHORT ByteCount;                     // Count of bytes
    UCHAR  Buffer[ ByteCount ];           // The bytes
} SMB_HEADER;
```

All SMBs have identical format up to the *PARAMETERWORDS* fields.
Different SMBs have a different number and interpretation of *PARAMETERWORDS* and *BUFFER*. All reserved fields in the SMB header must be zero. All quantities are sent in native Intel format.

- *COMMAND* is the operation code which this SMB is requesting, or responding to.

- *STATUS.DOSERROR.ERRORCLASS* and *STATUS.DOSERROR.ERROR* are set by the server and combine to give the error code of any failed server operation. If the client is capable of receiving 32 bit error returns, the status is returned in *STATUS.NTSTATUS* instead. When an error is returned, the server may choose to return only the header portion of the response SMB.

- *FLAGS* and *FLAGS2* contain bits which, depending on the negotiated protocol dialect, indicate various client capabilities.

- *PIDHIGH* is used in the *NTCREATEANDX* request SMB

- O *CONNECTIONLESS. SID*, and *CONNECTIONLESS.SEQUENCENUMBER* are used when the client to server connection is on a datagram oriented protocol such as IPX.

- *TID* identifies the subdirectory, or "tree", on the server which the client is accessing. SMBs which do not reference a particular tree should set *TID* to 0xFFFF

- *PID* is the caller's process id, and is generated by the client to uniquely identify a process within the client computer.

- *MID* is reserved for multiplexing multiple messages on a single Virtual Circuit (VC). A response message will always contain the same value as the corresponding request message.

2.5 SMB Protocol Dialect Negotiation

The first message sent from an SMB client to an SMB server must be one whose *COMMAND* field is SMB_COM_NEGOTIATE. The format of this client request includes an array of NULL terminated strings indicating the dialects of the SMB protocol which the client supports. The server compares this list against the list of dialects the server supports and returns the index of the chosen dialect in the response message.

2.6 Message Transport

Clients and servers can exchange messages over a NETBIOS reliable connection oriented transport, or a connectionless transport.

2.6.1 Reliable Connection Oriented Transports

When using a reliable connection oriented transport, the SMB protocol makes no higher level attempts to ensure sequenced delivery of messages between the client and server. The transport must have some mechanism to detect failures of either the client or server node, and to deliver such an indication to the client or server software so they can clean up state. When a reliable transport connection from a client terminates, all work in progress by that client is terminated by the server and all resources open by that client on the server are closed.

2.6.1.1 Connection Establishment

How the connection gets established depends on how the server name was resolved to the transport address: with DNS, with an explicit IP address, or with NETBIOS.

2.6.1.1.1 DNS

When using DNS, the server name is mapped onto an IP address and the connection is established by using the session establishment protocol as outlined in RFC 1001 and RFC 1002. The client should initiate the session setup using a called name which is obtained by taking the first component of the server name, converting it to upper case, and padding it up to a length of 16 with banks (hex 20 value).

2.6.1.1.2 Explicit IP Address

The connection is established by using the session establishment protocol as outlined in RFC 1001 and RFC 1002; the client should use "*SMBSERVER " as the called name in the session establishment protocol (since it does not know the server name).

2.6.1.1.3 NETBIOS

When using NETBIOS name resolution, the NETBIOS session establishment protocol as outlined in RFC 1001 and RFC 1002 must also be used. The NETBIOS name used for session establishment is the server name converted to upper case and padded to a length of 16 with blanks (hex 20 value).

Server-side Connection Procedures

The server should register a listen on at least one of the following names on the network using the NETBIOS name registration services. If the server wishes to support clients that use NETBIOS name resolution, it registers a 16 character name that is obtained by padding the server machine name with additional blanks if required. If the server wishes to support clients that use DNS name resolution, the name to register is obtained by taking the first component of the server name and padding it up to a length of 16 with blanks, and the 16th character of the name must be a blank (20 hex). Note: while the local server name and the registered DNS server name may differ, it usually makes administration easier to have them the same.

If servers wish to allow access via explicit IP address, they should register the name "*SMBSERVER " (padded to 16 characters with blanks) as a local name in NETBIOS. This name must not be defended on the network.

2.6.1.2 Connection Management

Once a connection is established, the rules for reliable transport connection dissolution are:

- If a server receives a transport establishment request from a client with which it is already conversing, the server may terminate all other transport connections to that client. This is to recover from the situation where the client was suddenly rebooted and was unable to cleanly terminate its resource sharing activities with the server.

- A server may drop the transport connection to a client at any time if the client is generating malformed or illogical requests. However, wherever possible the server should first return an error code to the client indicating the cause of the abort.

- If a server gets a hard error on the transport (such as a send failure) the transport connection to that client may be aborted.

- A server may terminate the transport connection when the client has no open resources on the server, however, we recommend that the termination be performed only after some time has passed or if resources are scarce on the server. This will help performance in that the transport connection will not need to be reestablished if activity soon begins anew. Client software is expected to be able to automatically reconnect to the server if this happens..

2.6.2 Connectionless Transports

The SMB protocol can be run over connectionless transports such as IPX and UDP/IP. Since connectionless transports do not support reliable delivery, this has to be implemented in the SMB protocol itself when utilizing such transports.

Unlike a traditional transport protocol, the connectionless SMB protocol is asymmetric. Wherever possible, processing overhead has been moved from the server to the client so that the server can scale to a large number of clients efficiently. For example, the server does not initiate retransmission of lost responses. It is entirely up to the client to resend the request in the case of lost packets in either direction.

The SMB header includes two fields specifically designed for use on connectionless transports. "*Sid"* is the server's session ID and "*SequenceNumber"* is the message sequence number. The *Sid* value is generated by the server, and returned to the client in the NegotiateProtocol response. The client must use this *Sid* value in all future SMB exchanges with this server during this resource sharing session. *SequenceNumber* is supplied by the client. A valid *SequenceNumber* is either zero or one greater than the previous sequence number sent by the client.

For sequenced commands, the server requires that the sequence numbers progress in order, S, S+1, S+2, ... The sequence number wraps to one (1) not zero. The wrap around progression is: 65534, 65535, 1, 2, ... Out of sequence commands are ignored by the server.

For unsequenced commands (i.e. *SequenceNumber* is 0) the redirector must use the *Mid* field to identify SMB responses. The redirector should take steps to generate relatively unique values for *Mid* for each request. In particular, the client must ensure that it never has two or more distinct requests outstanding to the server whose *SequenceNumbers* are 0 and whose *Mid*s are identical.

The client must limit the negotiated buffer size to the maximum MTU of the connectionless transport. If desired, the client could dynamically determine the maximum packet size by sending echo SMBs to the server using various packet sizes and then selecting the largest size which worked correctly.

For SMB operation over connectionless transports, commands are divided into two classes: sequenced commands and unsequenced commands. Sequenced commands are used for operations which cause a state change on the server that cannot be repeated, and which have relatively few bytes in the response. For example, file open/close or record locking. Unsequenced commands are used for operations which can be performed as many times as necessary with the same result each time or which have multi-packet responses. For example, reading or writing to a disk file. The client should must send all commands with a large response size as unsequenced; such commands include file read and file search.

2.6.2.1 Errors specific to connectionless transport operation

If the response to a sequenced command is too large, the server will fail the request with a *Status.DosError.ErrorClass* set to SMB_ERR_CLASS_SERVER and *Status.DosError.Error* set to ERRerror. If the *Sid* value is incorrect, the server will fail the request with a *Status.DosError.ErrorClass* set to SMB_ERR_CLASS_SERVER and *Status.DosError.Error* set to SMB_ERR_BAD_SID. If the server has an SMB in progress which matches either *SequenceNumber* for sequenced commands or *Mid* for unsequenced commands, it will respond with *Status.DosError.ErrorClass* set to SMB_ERR_CLASS_SERVER and *Status.DosError.Error* set to SMB_ERR_WORKING.

2.6.2.2 Transaction SMBs

The exceptions to the "large response requires unsequenced" rule are *transaction SMBs*. These SMBs are used both to retrieve bulk data from the server (EG: enumerate shares, etc.) and to change the server's state (EG: add a new share, change file permissions, etc.) Transaction requests are also unusual because they can have a multiple part request and/or a multiple part response. For this reason, transactions are handled as a set of sequenced commands to the server. Each part of a request is sent as a sequenced command using the same *Mid* value and an increasing *Seq* value. The server responds to each request piece except the last one with a response indicating that the server is ready for the next piece. The last piece is responded to with the first piece of the result data. The client then sends a transaction secondary SMB with *ParameterDisplacement* set to the number of parameter bytes received so far and *DataDisplacement* set to the number of data bytes received so far and *ParameterCount*, *ParameterOffset*, *DataCount*, and *DataOffset* set to zero (0). The server responds with the next piece of the transaction result. The process is repeated until all of the response information has been received. When the transaction has been completed, the redirector must send another sequenced command (an echo SMB will do fine) to the server to allow the server to know that the final piece was received and that resources allocated to the transaction command may be released.

The flow is as follows, where (S) is the *SequenceNumber*, (N) is the number of request packets to be sent from the client to the server, and (M) is the number of response packets to be sent by the server to the client:

Client	<->	Server
SMB(S) Transact	->	
	<-	OK (S) send more data
[repeat N-1 times:		
SMB(S+1) Transact secondary	->	
	<-	OK (S+1) send more data
SMB(S+N-1)		
]		
	<-	OK (S+N-1) transaction response (1)
[repeat M-1 times:		
SMB(S+N) Transact secondary	->	
	<-	OK (S+N) transaction response (2)
SMB(S+N+M-2) Transact secondary	->	
	<-	OK (S+N+M-2] transaction response (M)
]		
SMB(S+N+M-1) Echo	->	
	<-	OK (S+N+M-1) echoed

In order to allow the server to detect clients which have been powered off, have crashed, etc., the client must send commands to the server periodically if it has resources open on the server. If nothing has been received from a client for awhile, the server will assume that the client is no longer running and disconnect the client. This includes closing any files that the client had open at the time and releasing any resources being used on behalf of the client. Clients should at least send an echo SMB to the server every four (4) minutes if there is nothing else to send. The server will disconnect clients after a configurable amount of time which cannot be less than five (5) minutes. (Note: the NT server has a default timeout value of 15 minutes.)

2.7 Opportunistic Locks

Network performance can be increased if the client can locally buffer file data. For example, the client does not have to write information into a file on the server if the client knows that no other process is accessing the data. Likewise, the client can buffer read-ahead data from the file if the client knows that no other process is writing the data.

The mechanism which allows clients to dynamically alter their buffering strategy in a consistent manner is knows as "opportunistic locks", or *OPLOCKS* for short. Versions of the SMB file sharing protocol including and newer than the LANMAN1.0 dialect support oplocks.

There are three different types of oplocks:

- An *EXCLUSIVE* oplock allows a client to open a file for exclusive access and allows the client to perform arbitrary buffering
- A *BATCH* oplock allows a client to keep a file open on the server even though the local accessor on the client machine has closed the file.
- A *LEVEL II* oplock indicates there are multiple readers of a file, and no writers. Level II oplocks are supported if the negotiated dialect is `NT LM 0.12` or later.

When a client opens a file, it requests the server to grant it a particular type of oplock on the file. The response from the server indicates the type of oplock granted to the client. The client uses the granted oplock type to adjust its buffering policy.

The `SMB_COM_LOCKING_ANDX` SMB is used to convey oplock break and response information.

Oplocks are not supported over connectionless transports.

2.7.1 Exclusive Oplocks

If a client is granted an exclusive oplock, it may buffer lock information, read-ahead data, and write data on the client because the client knows that it is the only accessor to the file. The basic protocol is that the redirector on the client opens the file requesting that an oplock be given to the client. If the file is open by anyone else, then the client is refused the oplock and no local buffering may be performed on the local client. This also means that no readahead may be performed to the file, unless the redirector knows that it has the read ahead range locked. If the server grants the exclusive oplock, the client can perform certain optimizations for the file such as buffering lock, read, and write data.

The exclusive oplock protocol is:

Client		<->	Server
A ==========	B ==========	===	================================
Open ("foo")		->	
		<-	Open OK. Exclusive oplock granted.
	Open("foo")	->	
		<-	oplock break to A
lock(s)		->	
		<-	lock(s) response(s)
write(s)		->	
		<-	write(s) response(s)
close or done		->	
		<-	open response to B

As can be seen, when client A opens the file, it can request an exclusive oplock. Provided no one else has the file open on the server, then the oplock is granted to client A. If, at some point in the future, another client, such as client B, requests an open to the same file, then the server must have client A break its oplock. Breaking the oplock involves client A sending the server any lock or write data that it has buffered, and then letting the server know that it has acknowledged that the oplock has been broken. This synchronization message informs the server that it is now permissible to allow client B to complete its open.

Client A must also purge any readahead buffers that it has for the file. This is not shown in the above diagram since no network traffic is needed to do this.

2.7.2 Batch Oplocks

Batch oplocks are used where common programs on a client behave in such a way that causes the amount of network traffic on a wire to go beyond an acceptable level for the functionality provided by the program.

For example, the command processor executes commands from within a command procedure by performing the following steps:

- Opening the command procedure.
- Seeking to the "next" line in the file.
- Reading the line from the file.
- Closing the file.
- Executing the command.

This process is repeated for each command executed from the command procedure file. As is obvious, this type of programming model causes an inordinate amount of processing of files, thereby creating a lot of network traffic that could otherwise be curtailed if the program were to simply open the file, read a line, execute the command, and then read the next line.

Batch oplocking curtails the amount of network traffic by allowing the client to skip the extraneous open and close requests. When the command processor then asks for the next line in the file, the client can either ask for the next line from the server, or it may have already read the data from the file as readahead data. In either case, the amount of network traffic from the client is greatly reduced.

If the server receives either a rename or a delete request for the file that has a batch oplock, it must inform the client that the oplock is to be broken. The client can then change to a mode where the file is repeatedly opened and closed.

The batch oplock protocol is:

Client		<->	Server
A ==========	B ==========	====	==========
Open("foo")		->	
		<-	Open OK. Batch oplock granted.
Read		->	
		<-	data
<close>			
<open>			
<seek>			
		->	read
		<-	data
<close>			
	Open("foo")	->	
		<-	Oplock break to A
Close		->	
		<-	Close OK to A
		<-	Open OK to B

When client A opens the file, it can request an oplock. Provided no one else has the file open on the server, then the oplock is granted to client A. Client A, in this case, keeps the file open for its caller across multiple open/close operations. Data may be read ahead for the caller and other optimizations, such as buffering locks, can also be performed.

When another client requests an open, rename, or delete operation to the server for the file, however, client A must cleanup its buffered data and synchronize with the server. Most of the time this involves actually closing the file, provided that client A's caller actually believes that he has closed the file. Once the file is actually closed, client B's open request can be completed.

2.7.3 Level II Oplocks

Level II oplocks allow multiple clients to have the same file open, providing that no client is performing write operations to the file. This is important for many environments because most compatibility mode opens from down-level clients map to an open request for shared read/write access to the file. While it makes sense to do this, it also tends to break oplocks for other clients even though neither client actually intends to write to the file.

The Level II oplock protocol is:

Client		<->	Server
A ==========	B ==========	 ====	 ==========
Open("foo")		->	
		<-	Open OK. Exclusive oplock granted.
Read		->	
		<-	data
	Open("foo")	->	
		<-	Break to Level II oplock to A
lock(s)		->	
		<-	lock(s) response(s)
done		->	
		<-	Open OK. Oplock II oplock granted to B

This sequence of events is very much like an exclusive oplock. The basic difference is that the server informs the client that it should break to a level II lock when no one has been writing the file. That is, client A, for example, may have opened the file for a desired access of READ, and a share access of READ/WRITE. This means, by definition, that client A will not performed any writes to the file.

When client B opens the file, the server must synchronize with client A in case client A has any buffered locks. Once it is synchronized, client B's open request may be completed. Client B, however, is informed that he has a level II oplock, rather than an exclusive oplock to the file.

In this case, no client that has the file open with a level II oplock may buffer any lock information on the local client machine. This allows the server to guarantee that if any write operation is performed, it need only notify the level II clients that the lock should be broken without having to synchronize all of the accessors of the file.

The level II oplock may be *BROKEN TO NONE*, meaning that some client that had the file opened has now performed a write operation to the file. Because no level II client may buffer lock information, the server is in a consistent state. The writing client, for example, could not have written to a locked range, by definition. Read ahead data may be buffered in the client machines, however, thereby cutting down on the amount of network traffic required to the file. Once the level II oplock is broken, however, the buffering client must flush its buffers and degrade to performing all operations on the file across the network. No oplock break response is expected from a client when the server breaks a client from *LEVEL II* to *NONE*.

2.8 Security Model

Each server makes a set of resources available to clients on the network. A resource being shared may be a directory tree, named pipe, printer, etc. So far as clients are concerned, the server has no storage or service dependencies on any other servers; a client considers the server to be the sole provider of the file (or other resource) being accessed.

The SMB protocol requires server authentication of users before file accesses are allowed, and each server authenticates its own users. A client system must send authentication information to the server before the server will allow access to its resources.

The SMB protocol defines two methods which can be selected by the server for security: *share level* and *user level*:

- A *share level* server makes some directory on a disk device (or other resource) available. An optional password may be required to gain access. Thus any user on the network who knows the name of the server, the name of the resource and the password has access to the resource. Share level security servers may use different passwords for the same shared resource with different passwords allowing different levels of access.

- A *user level* server makes some directory on a disk device (or other resource) available but in addition requires the client to provide a user name and corresponding user password to gain access. User level servers are preferred over share level servers for any new server implementation, since organizations generally find *user level* servers easier to administer as employees come and go. User level servers may use the account name to check access control lists on individual files, or may have one access control list that applies to all files in the directory.

When a *user level* server validates the account name and password presented by the client, an identifier representing that authenticated instance of the user is returned to the client in the *Uid* field of the response SMB. This *Uid* must be included in all further requests made on behalf of the user from that client. A *share level* server returns no useful information in the *Uid* field.

The user level security model was added after the original dialect of the SMB protocol was issued, and subsequently some clients may not be capable of sending account name and passwords to the server. A server in user level security mode communicating with one of these clients will allow a client to connect to resources even if the client has not sent account name and password information:

1. If the client's computer name is identical to an account-name known on the server, and if the password supplied to connect to the shared resource matches that account’s password, an implicit "user logon" will be performed using those values.

If the above fails, the server may fail the request or assign a default account name of its choice.

2. The value of *Uid* in subsequent requests by the client will be ignored and all access will be validated assuming the account name selected above.

2.9 Resource Share/Access Example

The following examples illustrate a possible command line user interface for a server to offer a disk resource, and for a client to connect to and use that resource.

a) NET SHARE

The *NET SHARE* command, when executed on the server, specifies a directory name to be made available to clients on the network. A share name must be given, and this name is presented by clients wishing to access the directory.

Examples:

```
NET SHARE  src=c:\dir1\src  "bonzo"
```

assigns password *BONZO* to all files within directory *C:\DIR1\SRC* and its subdirectories with the share name *SRC* being the name used to connect to this resource.

```
NET SHARE  c=c:\        " "        RO

NET SHARE  work=c:\work  "flipper"  RW
```

offers read-only access to everything on the *C* drive. Offers read-write access to all files within the *C:\WORK* directory and its subdirectories.

The above example is appropriate for servers operating as a *SHARE LEVEL* server. A *USER LEVEL* server would not require the permissions or password, since the combination of the client's account name and specific access control lists on files is sufficient to govern access.

b) NET USE

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

Examples:

```
1.  NET USE  d: \\Server1\src  "bonzo"
```

gains full access to the files and directories on Server1 matching the offer defined by the netname *SRC* with the password of *BONZO*. The user may now address files on *SERVER1 C:\DIR1\SRC* by referencing d:. E.g. "type d:srcfile1.c".

```
2.  NET USE  e: \\Server1\c

3.  NET USE  f: \\Server1\work   "flipper"
```

Now any read request to any file on that node (drive c) is valid (e.g. "type e:\bin\foo.bat"). Read-write requests only succeed to files whose pathnames start with f: (e.g. "copy foo f:foo.tmp" copies foo to Server1 c:\work\foo.tmp).

For *USER LEVEL* servers, the client would not provide a password with the *NET USE* command.

The client software must remember the drive identifier supplied with the NET USE request and associate it with the *TID* value returned by the server in the SMB header. Subsequent requests using this *TID* must include only the pathname relative to the connected subtree as the server treats the subtree as the root directory (virtual root). When the user references one of the remote drives, the client software looks through its list of drives for that node and includes the tree id associated with this drive in the *TID* field of each request.

Note that one shares a directory and all files underneath that directory are then affected. If a particular file is within the range of multiple shares, connecting to any of the share 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.

2.10 Authentication

An SMB server keeps an encrypted form of a client's password. To gain authenticated access to server resources, the server sends a challenge to the client, which the client responds to in a way that proves it knows the client's password.

Authentication makes use of DES encryption [5] in block mode. We denote the DES encryption function as E(K,D), which accepts a seven byte key (K) and an eight byte data block (D) and produces an eight byte encrypted data block as its value. If the data to be encrypted is longer than eight bytes, the encryption function is applied to each block of eight bytes in sequence and the results are appended together. If the key is longer than seven bytes, the data is first completely encrypted using the first seven bytes of the key, then the second seven bytes, etc., appending the results each time. In other words, to encrypt the 16 byte quantity D0D1 with the 14 byte key K0K1,

```
E(K0K1,D0D1) = E(K0,D0)E(K0,D1)E(K1,D0)E(K1,D1)
```

The *EncryptionKey* field in the SMB_COM_NEGPROT response contains an 8 byte challenge denoted below as "C8", chosen to be unique to prevent replay attacks; the client responds with a 24 byte response denoted "P24", and computed as described below. (Note: the name "*EncryptionKey*" is historical -- it doesn't actually hold an encryption key.)

Clients send the response to the challenge in the SMB_COM_TREE_CONNECT, SMB_COM_TREE_CONNECT_ANDX, and/or SMB_COM_SESSION_SETUP_ANDX request which follows the SMB_COM_NEGPROT message exchange. The server must validate the response by performing the same computations the client did to create it, and ensuring the strings match.

If the comparisons fail, the client system may be incapable of encryption; if so the string may be the user password in clear text. The server should try to validating the string as though it were the unencrypted password.

The SMB field used to store the response depends upon the request:

- *Password* in SMB_COM_TREE_CONNECT
- *Password* in SMB_COM_TREE_CONNECT_ANDX
- *AccountPassword* in SMB_COM_SESSION_SETUP_ANDX

(Note: again, the names are historical, and do not reflect this usage.)

The contents of the response to the challenge depends on the SMB dialect, as outlined in the following sections:

2.10.1 Pre NT LM 0.12

- The client and server both compute

  ```
  P16 = E(P14,S8)
  ```

  and

  ```
  P24 = E(P21,C8)
  ```

where:

- P14 is a 14 byte string containing the user's password in clear text, upper cased, padded with spaces

- S8 is an eight byte string whose value is available from Microsoft upon request.
- P21 is a twenty one byte string obtained by appending five null bytes to the string P16, just computed
- C8 is the value of the challenge sent in the EncryptionKey field in the SMB_COM_NEGPROT response for this connection.

2.10.2 NT LM 0.12

The client and server both compute

```
P16 = MD4(U(PN))
```

and

```
P24 = E(P21, C8)
```

where:

- PN is a string containing the user's password in clear text, case sensitive, no maximum length
- U(x) of an ASCII string "x" is that string converted to Unicode
- MD4(x) of an octet string "x" is the 16 byte MD4 message digest [6] of that string
- P21 and C8 are as above.

2.11 Distributed Filesystem (DFS) Support

Protocol dialects of NT LM 0.12 and later support distributed filesystem operations. The distributed filesystem gives a way for this protocol to use a single consistent file naming scheme which may span a collection of different servers and shares. The distributed filesystem model employed is a referral - based model. This protocol specifies the manner in which clients receive referrals.

The client can set a flag in the request SMB header indicating that the client wants the server to resolve this SMB's paths within the DFS known to the server. The server attempts to resolve the requested name to a file contained within the local directory tree indicated by the TID of the request and proceeds normally. If the request pathname resolves to a file on a different system, the server returns the following error:

> STATUS_DFS_PATH_NOT_COVERED - the server does not support the part of the DFS namespace needed to resolved the pathname in the request. The client should request a referral from this server for further information.

A client asks for a referral with the TRANS2_DFS_GET_REFERRAL request containing the DFS pathname of interest. The response from the server indicates how the client should proceed.

The method by which the topological knowledge of the DFS is stored and maintained by the servers is not specified by this protocol.

3. SMB Message Formats and Data Types

This section describes the entire set of SMB commands and responses exchanged between SMB clients and servers. It also details which SMBs are introduced into the protocol as higher dialect levels are negotiated.

3.1 SMB Header

While each SMB command has specific encodings, there are some fields in the SMB header which have meaning to all SMBs. These fields and considerations are described in the following sections.

3.1.1 Flags field

This field contains 8 individual flags, numbered from least significant to most significant, and have the following meanings:

Bit ===	Meaning ==========	Earliest Dialect ==========
0	When set (returned) from the server in the SMB_COM_NEGOTIATE response SMB, this bit indicates that the server supports the "sub dialect" consisting of the LockandRead and WriteandUnlock protocols defined later in this document.	LANMAN1.0
1	When on (on an SMB request being sent to the server), the client guarantees that there is a receive buffer posted such that a send without acknowledgment can be used by the server to respond to the client's request.	
2	Reserved (must be zero).	
3	When on, all pathnames in this SMB must be treated as caseless. When off, the pathnames are case sensitive.	LANMAN1.0
4	When on (in SMB_COM_SESSION_SETUP_ANDX defined later in this document), all paths sent to the server by the client are already canonicalized. This means that file/directory names are in upper case, are valid characters, . and .. have been removed, and single backslashes are used as separators.	LANMAN1.0
5	When on (in SMB_COM_OPEN, SMB_COM_CREATE and SMB_COM_CREATE_NEW), this indicates that the client is requesting that the file be "opportunistically" locked if this process is the only process which has the file open at the time of the open request. If the server "grants" this oplock request, then this bit should remain set in the corresponding response SMB to indicate to the client that the oplock request was granted. See the discussion of "oplock" in the sections defining the SMB_COM_OPEN_ANDX and SMB_COM_LOCKING_ANDX protocols later in this document (this bit has the same function as bit 1 of Flags if the SMB_COM_OPEN_ANDX SMB).	LANMAN1.0
6	When on (in core protocols SMB_COM_OPEN_ANDX, SMB_COM_CREATE and SMB_COM_CREATE_NEW), this indicates that the server should notify the client on any action which can modify the file (delete, setattrib, rename, etc.) by another client. If not set, the server need only notify the client about another open request by a different client. See the discussion of "oplock" in the sections defining the SMB_COM_OPEN_ANDX and SMB_COM_LOCKING_ANDX SMBs later in this document (this bit has the same function as bit 2 of smb_flags of the SMB_COM_OPEN_ANDX SMB). Bit6 only has meaning if bit5 is set..	LANMAN1.0
7	When on, this SMB is being sent from the server in response to a client request. The Command field usually contains the same value in a protocol request from the client to the server as in the matching response from the server to the client. This bit unambiguously distinguishes the command request from the command response.	PC NETWORK PROGRAM 1.0

3.1.2 Flags2 Field

This field contains six individual flags, numbered from least significant bit to most significant bit, which are defined below. Flags which not defined must be set to zero.

Bit ===	Meaning ==========	Earliest Dialect ==========
0	If set, the client knows how to handle names which do not conform to the MS-DOS 8.3 naming convention.	
1	If set, the client is aware of extended attributes	
2	If set, SMB_FLAGS2_IS_LONG_NAME	
12	If set, any request pathnames in this SMB should be resolved in the Distributed File System	NT LM 0.12
13	If set, indicates that a read will be permitted if the client does not have read permission but does have execute permission. This flag is only useful on a read request.	
14	If set, specifies that the returned error code is a 32 bit error code in Status.NtStatus. Otherwise the Status.DosError.ErrorClass and Status.DosError.Error fields contain the DOS-style error information. When passing NT status codes is negotiated, this flag should be set for every SMB.	NT LM 0.12
15	If set, any strings in this SMB message are encoded as UNICODE. Otherwise, all strings are in ASCII.	NT LM 0.12

3.1.3 Tid Field

TID represents an instance of an authenticated connection to a server resource. *TID* is returned by the server to the client when the client successfully connects to a resource, and the client uses *TID* in subsequent requests referring to the resource.

If the server is executing in a *SHARE LEVEL* security mode, *TID* is the only thing used to allow access to the shared resource. Thus if the user is able to perform a successful connection to the server specifying the appropriate netname and passwd (if any) the resource may be accessed according to the access rights associated with the shared resource (same for all who gained access this way).

If however the server is executing in *USER LEVEL* security mode, access to the resource is based on the *UID* (validated on the SMB_COM_SESSION_SETUP_ANDX request) and the *TID* is NOT associated with access control but rather merely defines the resource (such as the shared directory tree).

In most SMB requests, *TID* must contain a valid value. Exceptions include prior to getting a *TID* established including SMB_COM_NEGOTIATE, SMB_COM_TREE_CONNECT, SMB_COM_ECHO, and SMB_COM_SESSION_SETUP_ANDX. 0xFFFF should be used for Tid for these situations. The server is always responsible for enforcing use of a valid *TID* where appropriate.

3.1.4 Pid Field

PID uniquely identifies a client process. Clients inform servers of the creation of a new process by simply introducing a new *PID* value into the dialogue for new processes.

In the core protocol, the SMB_COM_PROCESS_EXIT SMB was used to indicate the catastrophic termination of a process on the client. In the single tasking DOS system, it was possible for hard errors to occur causing the destruction of the process with files remaining open. Thus a SMB_COM_PROCESS_EXIT SMB was sent for this occurrence to allow the server to close all files opened by that process.

In the `LANMAN 1.0` and newer dialects, no SMB_COM_PROCESS_EXIT SMB is sent. The client operating system must ensure that the appropriate close and cleanup SMBs will be sent when the last process referencing the file closes it. From the server's point of view, there is no concept of FIDs "belonging to" processes. A FID returned by the server to one process may be used by any other process using the same transport connection and *TID*. There is no process creation SMB sent to the server; it is up to the client to ensure only valid client processes gain access to *FID*s (and *TID*s). On SMB_COM_TREE_DISCONNECT (or when the client and server session is terminated) the server will invalidate any files opened by any process on that client.

3.1.5 Mid Field

Clients using the `LANMAN 1.0` and newer dialects will typically be multitasked and allow multiple asynchronous input/output requests per task. Therefore a multiplex ID (*MID*) is used along with *PID* to allow multiplexing the single client and server connection among the client's multiple processes, threads, and requests per thread.

Regardless of negotiated dialect, the server is responsible for ensuring that every response contains the same *MID* and *PID* values as its request. The client may then use the *MID* and *PID* values for associating requests and responses and may have up to the negotiated number of requests outstanding at any time to a particular server.

3.1.6 Status Field

An SMB returns error information to the client in the *STATUS* field. Protocol dialects prior to `NT LM 0.12` return status to the client using the combination of *STATUS.DOSERROR.ERRORCLASS* and *STATUS.DOSERROR.ERROR*. Beginning with `NT LM 0.12` SMB servers can return 32 bit error information to clients using *STATUS.NTSTATUS* if the incoming client SMB has bit 14 set in the *FLAGS2* field of the SMB header. Any valid NT status code may be returned in this case. The contents of response parameters is not guaranteed in the case of an error return, and must be ignored. For write behind activity, a subsequent write or close of the file may return the fact that a previous write failed. Normally write behind failures are limited to hard disk errors and device out of space.

3.1.7 Timeouts

In general, SMBs are not expected to block at the server; they should return "immediately". But some SMB requests do indicate timeout periods for the completion of the request on the server. If a server implementation can not support timeouts, then an error can be returned just as if a timeout had occurred if the resource is not available immediately upon request.

3.1.8 Data Buffer (*BUFFER*) and String Formats

The data portion of SMBs 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.

Identifier	Description	Value
Data Block	See Below	1
Dialect	Null terminated String	2
Pathname	Null terminated String	3
ASCII	Null terminated String	4
Variable block	See Below	5

When the identifier indicates a data block or variable block then the format is a word indicating the length followed by the data.

In all dialects prior to `NT LM 0.12`, all strings are encoded in ASCII. If the agreed dialect is `NT LM 0.12` or later, Unicode strings may be exchanged. Unicode strings include file names, resource names, and user names. This applies to null-terminated strings, length specified strings and the type-prefixed strings. In all cases where a string is passed in Unicode format, the Unicode string must be word-aligned with respect to the beginning of the SMB. Should the string not naturally fall on a two-byte boundary, a null byte of padding will be inserted, and the Unicode string will begin at the next address. In the description of the SMBs, items that may be encoded in Unicode or ASCII are labeled as STRING. If the encoding is ASCII, even if the negotiated string is Unicode, the quantity is labeled as UCHAR.

For type-prefixed Unicode strings, the padding byte is found after the type byte. The type byte is 4 (indicating SMB_FORMAT_ASCII) independent of whether the string is ASCII or Unicode. For strings whose start addresses are found using offsets within the fixed part of the SMB (as opposed to simply being found at the byte following the preceding field,) it is guaranteed that the offset will be properly aligned.

Strings that are never passed in Unicode are:

- The protocol strings in the Negotiate SMB request.
- The service name string in the Tree Connect And X SMB.

When Unicode is negotiated, bit 15 should be set in the *FLAGS2* field of every SMB header.

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

3.2 File Names

File names in the SMB protocol consist of components separated by a backslash ('\'). Early clients of the SMB protocol required that the name components adhere to an 8.3 format name. These names consist of two parts: a basename of no more than 8 characters, and an extension of no more than 3 characters. The basename and extension are separated by a '.'. All characters are legal in the basename and extension *EXCEPT* the space character (0x20) and:

" . / \[]:+|<>=;,*?

If the client has indicated long name support by setting *BIT2* in the *FLAGS2* field of the SMB header, this indicates that the client is not bound by the 8.3 convention. Specifically this indicates that any SMB which returns file names to the client may return names which do not adhere to the 8.3 convention, and have a total length of up to 255 characters. This capability was introduced with the `LM1.2X002` protocol dialect.

3.3 Wildcards

Some SMB requests allow wildcards to be given for the filename. The wildcard allows a number of files to be operated on as a unit without having to separately enumerate the files and individually operate on each one from the client.

If the client is using 8.3 names, each part of the name (base (8) or extension (3)) is treated separately. For long filenames the . in the name is significant even though there is no longer a restriction on the size of each of the components.

The ? character is a wild card for a single character. If a filename part commences with one or more "?"s then exactly that number of characters will be matched by the wildcards, e.g., "??x" equals "abx" but not "abcx" or "ax". When a filename part has trailing "?"s then it matches the specified number of characters or less, e.g., "x??" matches "xab", "xa" and "x", but not "xabc". If only "?"s are present in the filename part, then it is handled as for trailing "?"s

The * character matches an entire part of the name, as does an empty specification for that part. A part consisting of * means that the rest of the component should be filled with ? and the search should be performed with this wildcard character. For example, "*.abc" or ".abc" match any file with an extension of "abc". "*.*", "*" or "null" match all files in a directory.

If the negotiated dialect is "`NT LM 0.12`" or later, and the client requires MS-DOS wildcard matching semantics, UNICODE wildcards should be translated according to the following rules:

Translate the ? literal to >

Translate the . literal to " if it is followed by a ? or a *

Translate the * literal to < if it is followed by a .

The translation can be performed in-place.

3.4 DFS Pathnames

A DFS pathname adheres to the standard described in the FileNames section. A DFS enabled client accessing a DFS share should set the *Flags2* bit 12 in all name based SMB requests indicating to the server that the enclosed pathname should be resolved in the Distributed File System namespace. The pathname should always have the full file name, including the server name and share name. If the server can resolve the DFS name to a piece of local storage, the local storage will be accessed. If the server determines that the DFS name actually maps to a different server share, the access to the name will fail with the distinguished error STATUS_PATH_NOT_COVERED (SMB Status code 0xC0000257).

On receiving this error, the DFS enabled client should ask the server for a *referral* (see TRANS2_GET_DFS_REFERRAL). The referral request should contain the full file name.

The response to the request will contain a list of server and share names to try, and the part of the request file name that junctions to the list of server shares. If the ServerType field of the referral is set to 1 (SMB server), then the client should resubmit the request with the *ORIGINAL* file name to one of the server shares in the list, once again setting the Flags2 bit 12 bit in the SMB. If the ServerType field is not 1, then the client should strip off the part of the file name that junctions to the server share before resubmitting the request to one of servers in the list.

A response to a referral request may elicit a response that does *NOT* have the StorageServers bit set. In that case, the client should resubmit the *REFERRAL REQUEST* to one of the servers in the list, until it finally obtains a referral response that has the StorageServers bit set, at which point the client can resubmit the request SMB to one of the listed server shares.

If, after getting a referral with the StorageServers bit set and resubmitting the request to one of the server shares in the list, the server fails the request with STATUS_PATH_NOT_COVERED, it must be the case that there is an inconsistency between the view of the DFS namespace held by the server granting the referral and the server listed in that referral. In this case, the client may inform the server granting the referral of this inconsistency via the *TRANS2_REPORT_DFS_INCONSISTENCY* SMB.

3.5 Time And Date Encoding

When SMB requests or responses encode time values, the following describes the various encodings used.

```
struct {
        USHORT Day : 5;
        USHORT Month : 4;
        USHORT Year : 7;
} SMB_DATE;
```

The Year field has a range of 0-119, which represents years 1980 - 2099. The Month is encoded as 1-12, and the day ranges from 1-31.

```
struct {
        USHORT TwoSeconds : 5;
        USHORT Minutes : 6;
        USHORT Hours : 5;
} SMB_TIME;
```

Hours ranges from 0-23, Minutes range from 0-59, and TwoSeconds ranges from 0-29 representing two second increments within the minute.

```
typedef struct {
          ULONG LowTime;
          LONG HighTime;
} TIME;
```

TIME indicates a signed 64-bit integer representing either an absolute time or a time interval. Times are specified in units of 100ns. A positive value expresses an absolute time, where the base time (the 64-bit integer with value 0) is the beginning of the year 1601 AD in the Gregorian calendar. A negative value expresses a time interval relative to some base time, usually the current time.

```
typedef unsigned long UTIME;
```

UTIME is the number of seconds since Jan 1, 1970, 00:00:00.0 GMT.

3.6 Access Mode Encoding

Various client requests and server responses, such as SMB_COM_OPEN, pass file access modes encoded into a USHORT. The encoding of these is as follows:

```
    1111 11
    5432 1098 7654 3210
    rWrC rLLL rSSS rAAA

 where:

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

    S - Sharing mode:
        0 - Compatibility mode
        1 - Deny read/write/execute (exclusive)
        2 - Deny write
        3 - Deny read/execute
        4 - Deny none

    A - Access mode
        0 - Open for reading
        1 - Open for writing
        2 - Open for reading and writing
        3 - Open for execute

    rSSSrAAA = 11111111 (hex FF) indicates FCB open (???)

    C - Cache mode
        0 - Normal file
        1 - Do not cache this file

    L - Locality of reference
        0 - Locality of reference is unknown
        1 - Mainly sequential access
        2 - Mainly random access
        3 - Random access with some locality
        4 to 7 - Currently undefined
```

3.7 Open Function Encoding

OPENFUNCTION specifies the action to be taken depending on whether or not the file exists. This word has the following format:

```
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.
          1 -- Create file.

    r - reserved (must be zero).

    O - Open (action to be taken if file exists).
          0 - Fail.
          1 - Open file.
          2 - Truncate file.
```

3.8 Open Action Encoding

ACTION in the response to an open request specifies the action as a result of the Open request. It has the following format:

```
bits:

    1111 11
    5432 1098 7654 3210
    Lrrr rrrr rrrr rrOO

where:

    L - Lock (single user total file lock status).
          0 -- file opened by another user (or mode not supported by server).
          1 -- file is opened only by this user at the present  time.

    r - 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.
```

3.9 Device State Encoding

DEVICESTATE is encoded as follows:

```
    1111 11
    5432 1098 7654 3210
    BE** TTRR ---- ----

where:

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

    E - Endpoint
          0 => client end of pipe
          1 => server end of pipe

    TT - Type of pipe
          00 => pipe is a byte stream pipe
          01  => pipe is a message pipe

    RR - Read Mode
          00 => Read pipe as a byte stream
          01 => Read messages from pipe
```

3.10 File Attribute Encoding

When SMB messages exchange file attribute information, it is encoded in 16 bits as:

Value =======	Description ====================
0x01	Read only file
0x02	Hidden file
0x04	System file
0x08	Volume
0x10	Directory
0x20	Archive file
others	Reserved - must be 0

3.11 Batching Requests ("AndX" Messages)

`LANMAN1.0` and later dialects of the SMB protocol allow multiple SMB requests to be sent in one message to the server. Messages of this type are called AndX SMBs, and they obey the following rules:

- The embedded command does not repeat the SMB header information. Rather the next SMB starts at the *WORDCOUNT* field.

- All multiple (chained) requests must fit within the negotiated transmit size. For example, if SMB_COM_TREE_CONNECT_ANDX included OPENandX SMB_COM_OPEN_ANDX which included SMB_COM_WRITE were sent, they would all have to fit within the negotiated buffer size. This would limit the size of the write.

- There is one message sent containing the chained requests and there is one response message to the chained requests. The server may NOT elect to send separate responses to each of the chained requests.

- All chained responses must fit within the negotiated transmit size. This limits the maximum value on an embedded SMB_COM_READ for example. It is the client's responsibility to not request more bytes than will fit within the multiple response.

- The server will implicitly use the result of the first command in the "X" command. For example the *TID* obtained via SMB_COM_TREE_CONNECT_ANDX would be used in the embedded SMB_COM_OPEN_ANDX and the *FID* obtained in the SMB_COM_OPEN_ANDX would be used in the embedded SMB_COM_READ.

- Each chained request can only reference the same *FID* and *TID* as the other commands in the combined request. The chained requests can be thought of as performing a single (multi-part) operation on the same resource.

- The first *COMMAND* to encounter an error will stop all further processing of embedded commands. The server will not back out commands that succeeded. Thus if a chained request contained SMB_COM_OPEN_ANDX and SMB_COM_READ and the server was able to open the file successfully but the read encountered an error, the file would remain open. This is exactly the same as if the requests had been sent separately.

- If an error occurs while processing chained requests, the last response (of the chained responses in the buffer) will be the one which encountered the error. Other unprocessed chained requests will have been ignored when the server encountered the error and will not be represented in the chained response. Actually the last valid *ANDXCOMMAND* (if any) will represent the SMB on which the error occurred. If no valid *ANDXCOMMAND* is present, then the error occurred on the first request/response and *COMMAND* contains the command which failed. In all cases the error information are returned in the SMB header at the start of the response buffer.

- Each chained request and response contains the offset (from the start of the SMB header) to the next chained request/response (in the *ANDXOFFSET* field in the various "and X" protocols defined later e.g. SMB_COM_OPEN_ANDX). This allows building the requests unpacked. There may be space between the end of the previous request (as defined by *WORDCOUNT* and *BYTECOUNT*) and the start of the next chained request. This simplifies the building of chained protocol requests. Note that because the client must know the size of the data being returned in order to post the correct number of receives (e.g. SMB_COM_TRANSACTION, SMB_COM_READ_MPX), the data in each response SMB is expected to be truncated to the maximum number of 512 byte blocks (sectors) which will fit (starting at a DWORD boundary) in the negotiated buffer size with the odd bytes remaining (if any) in the final buffer.

3.12 "Transaction" Style Subprotocols

`SMB_COM_TRANSACTION` performs a symbolically named transaction. This transaction is known only by a name (no file handle used). `SMB_COM_TRANSACTION2` likewise performs a transaction, but a word parameter is used to identify the transaction instead of a name. `SMB_COM_NT_TRANSACTION` is used for commands that potentially need to transfer a large amount of data (greater than 64K bytes).

3.12.1 SMB_COM_TRANSACTION and SMB_COM_TRANSACTION2 Formats

Primary Client Request	Description
Command	SMB_COM_TRANSACTION or SMB_COM_TRANSACTION2
UCHAR WordCount;	Count of parameter words; value = (14 + SetupCount)
USHORT TotalParameterCount;	Total parameter bytes being sent
USHORT TotalDataCount;	Total data bytes being sent
USHORT MaxParameterCount;	Max parameter bytes to return
USHORT MaxDataCount;	Max data bytes to return
UCHAR MaxSetupCount;	Max setup words to return
UCHAR Reserved;	
USHORT Flags;	Additional information: bit 0 - also disconnect TID in *TID* bit 1 - one-way transaction (no resp)
ULONG Timeout;	
USHORT Reserved2;	
USHORT ParameterCount;	Parameter bytes sent this buffer
USHORT ParameterOffset;	Offset (from header start) to Parameters
USHORT DataCount;	Data bytes sent this buffer
USHORT DataOffset;	Offset (from header start) to data
UCHAR SetupCount;	Count of setup words
UCHAR Reserved3;	Reserved (pad above to word)
USHORT Setup[SetupCount];	Setup words (# = SetupWordCount)
USHORT ByteCount;	Count of data bytes
STRING Name[];	Name of transaction (NULL if SMB_COM_TRANSACTION2)

UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Parameters[ParameterCount];	Parameter bytes (# = ParameterCount)
UCHAR Pad1[];	Pad to SHORT or LONG
UCHAR Data[DataCount];	Data bytes (# = DataCount)

Interim Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

Secondary Client Request ==========	Description ==========
Command	SMB_COM_TRANSACTION_SECONDARY
UCHAR WordCount;	Count of parameter words = 8
USHORT TotalParameterCount;	Total parameter bytes being sent
USHORT TotalDataCount;	Total data bytes being sent
USHORT ParameterCount;	Parameter bytes sent this buffer
USHORT ParameterOffset;	Offset (from header start) to Parameters
USHORT ParameterDisplacement;	Displacement of these Parameter bytes
USHORT DataCount;	Data bytes sent this buffer
USHORT DataOffset;	Offset (from header start) to data
USHORT DataDisplacement;	Displacement of these data bytes
USHORT Fid;	*FID* for handle based requests, else 0xFFFF. This field is present only if this is an SMB_COM_TRANSACTION2 request.
USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Parameters[ParameterCount];	Parameter bytes (# = ParameterCount)
UCHAR Pad1[];	Pad to SHORT or LONG
UCHAR Data[DataCount];	Data bytes (# = DataCount)

Server Response	Description
UCHAR WordCount;	Count of data bytes; value = 10 + *SETUPCOUNT*
USHORT TotalParameterCount;	Total parameter bytes being sent
USHORT TotalDataCount;	Total data bytes being sent
USHORT Reserved;	
USHORT ParameterCount;	Parameter bytes sent this buffer
USHORT ParameterOffset;	Offset (from header start) to Parameters
USHORT ParameterDisplacement;	Displacement of these Parameter bytes
USHORT DataCount;	Data bytes sent this buffer
USHORT DataOffset;	Offset (from header start) to data
USHORT DataDisplacement;	Displacement of these data bytes
UCHAR SetupCount;	Count of setup words
UCHAR Reserved2;	Reserved (pad above to word)
USHORT Setup[SetupWordCount];	Setup words (# = SetupWordCount)
USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Parameters[ParameterCount];	Parameter bytes (# = ParameterCount)
UCHAR Pad1[];	Pad to SHORT or LONG
UCHAR Data[DataCount];	Data bytes (# = DataCount)

3.12.2 SMB_COM_NT_TRANSACTION Formats

Primary Client Request	Description
UCHAR WordCount;	Count of parameter words; value = (19 + SetupCount)
UCHAR MaxSetupCount;	Max setup words to return
USHORT Reserved;	

ULONG TotalParameterCount;	Total parameter bytes being sent
ULONG TotalDataCount;	Total data bytes being sent
ULONG MaxParameterCount;	Max parameter bytes to return
ULONG MaxDataCount;	Max data bytes to return
ULONG ParameterCount;	Parameter bytes sent this buffer
ULONG ParameterOffset;	Offset (from header start) to Parameters
ULONG DataCount;	Data bytes sent this buffer
ULONG DataOffset;	Offset (from header start) to data
UCHAR SetupCount;	Count of setup words
USHORT Function;	The transaction function code
UCHAR Buffer[1];	
USHORT Setup[SetupWordCount];	Setup words
USHORT ByteCount;	Count of data bytes
UCHAR Pad1[];	Pad to LONG
UCHAR Parameters[ParameterCount];	Parameter bytes
UCHAR Pad2[];	Pad to LONG
UCHAR Data[DataCount]; Data bytes	

Interim Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

Secondary Client Request	Description
UCHAR WordCount;	Count of parameter words = 18
UCHAR Reserved[3];	MBZ
ULONG TotalParameterCount;	Total parameter bytes being sent
ULONG TotalDataCount;	Total data bytes being sent
ULONG ParameterCount;	Parameter bytes sent this buffer
ULONG ParameterOffset;	Offset (from header start) to Parameters
ULONG ParameterDisplacement;	Specifies the offset from the start of the overall parameter block to the parameter bytes that are contained in this message
ULONG DataCount;	Data bytes sent this buffer
ULONG DataOffset;	Offset (from header start) to data
ULONG DataDisplacement;	Specifies the offset from the start of the overall data block to the data bytes that are contained in this message.
UCHAR Reserved1;	
USHORT ByteCount;	Count of data bytes
UCHAR Pad1[];	Pad to LONG
UCHAR Parameters[ParameterCount];	Parameter bytes
UCHAR Pad2[];	Pad to LONG
UCHAR Data[DataCount];	Data bytes

Server Response	Description
UCHAR WordCount;	Count of data bytes; value = 18 + SetupCount
UCHAR Reserved[3];	
ULONG TotalParameterCount;	Total parameter bytes being sent
ULONG TotalDataCount;	Total data bytes being sent
ULONG ParameterCount;	Parameter bytes sent this buffer

ULONG ParameterOffset;	Offset (from header start) to Parameters
ULONG ParameterDisplacement;	Specifies the offset from the start of the overall parameter block to the parameter bytes that are contained in this message
ULONG DataCount;	Data bytes sent this buffer
ULONG DataOffset;	Offset (from header start) to data
ULONG DataDisplacement;	Specifies the offset from the start of the overall data block to the data bytes that are contained in this message.
UCHAR SetupCount;	Count of setup words
USHORT Setup[SetupWordCount];	Setup words
USHORT ByteCount;	Count of data bytes
UCHAR Pad1[];	Pad to LONG
UCHAR Parameters[ParameterCount];	Parameter bytes
UCHAR Pad2[];	Pad to SHORT or LONG
UCHAR Data[DataCount];	Data bytes

3.12.3 Functional Description

The `SMB_COM_TRANSACTION` command's scope includes named pipes and mailslots. Where the resource is unidirectional (such as class 2 writes to mailslots), *bit1* of *Flags* in the request can be set indicating that no response is needed. The other transactions accommodate `IOCTL` requests and file system requests which require the transfer of an extended attribute list.

The transaction *Setup* 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 client and server implementations. The transaction 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 client knows the number of parameter and data bytes expected to be sent and passes this information to the server via the primary request (*TotalParameterCount* and *TotalDataCount*). This may be reduced by lowering the total number of bytes expected (*TotalParameterCount* and *TotalDataCount*) in each (if any) secondary request.

When the amount of parameter bytes received (total of each *ParameterCount*) equals the total amount of parameter bytes expected (smallest *TotalParameterCount*) received, then the server has received all the parameter bytes.

Likewise, when the amount of data bytes received (total of each *DATACOUNT*) equals the total amount of data bytes expected (smallest *TOTALDATACOUNT*) 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 (*PARAMETEROFFSET* and *DATAOFFSET*) and the length fields (*PARAMETERCOUNT* and *DATACOUNT*). The displacement of the bytes (relative to start of each) is also known (*PARAMETERDISPLACEMENT* and

DATADISPLACEMENT). Thus the server is able to reassemble the parameter and data bytes should the individual requests 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 client knows the maximum amount of data bytes and parameter bytes which the server may return (from *MAXPARAMETERCOUNT* and *MAXDATACOUNT* of the request). Thus the client initializes its bytes expected variables to these values. The server then informs the client of the actual amounts being returned via each message of the server response (*TOTALPARAMETERCOUNT* and *TOTALDATACOUNT*). The server may reduce the expected bytes by lowering the total number of bytes expected (*TOTALPARAMETERCOUNT* and/or *TOTALDATACOUNT*) in each (any) response.

When the amount of parameter bytes received (total of each *PARAMETERCOUNT*) equals the total amount of parameter bytes expected (smallest *TOTALPARAMETERCOUNT*) received, then the client has received all the parameter bytes.

Likewise, when the amount of data bytes received (total of each *DATACOUNT*) equals the total amount of data bytes expected (smallest *TOTALDATACOUNT*) received, then the client has received all the data bytes.

The parameter bytes should normally be returned first followed by the data bytes. However, the client knows where each begins and ends in each buffer by the offset fields (*PARAMETEROFFSET* and *DATAOFFSET*) and the length fields (*PARAMETERCOUNT* and *DATACOUNT*). The displacement of the bytes (relative to start of each) is also known (*PARAMETERDISPLACEMENT* and *DATADISPLACEMENT*). The client is able to reassemble the parameter and data bytes should the server responses be received out of sequence.

If a connectionless transport is being used, the transaction requests must be properly sequenced in the *CONNECTIONLESS.SEQUENCENUMBER* SMB header field. The *MID* of any secondary client requests must match the *MID* of the primary client request. The server responds to each request piece except the last one with a response indicating that the server is ready for the next piece. The last piece is responded to with the first piece of the result data. The client then sends an `SMB_COM_TRANSACTION_SECONDARY` SMB with *PARAMETERDISPLACEMENT* set to the number of parameter bytes received so far and *DATADISPLACEMENT* set to the number of data bytes received so far and *PARAMETERCOUNT*, *PARAMETEROFFSET*, *DATACOUNT*, and *DATAOFFSET* set to zero (0). The server responds with the next piece of the transaction result. The process is repeated until all of the response information has been received. When the transaction has been completed, the client must send another sequenced command (such as an `SMB_COM_ECHO`) to the server to allow the server to know that the final piece was received and that resources allocated to the transaction command may be released.

The flow for these transactions over a connection oriented transport is:

1. The client sends the primary client request identifying 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 the 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 client then sends another buffer full of bytes to the server. This step is repeated until all of the bytes are sent and received.

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 result bytes have been returned.

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

Client	<->	Server
Primary TRANSACTION request	->	
	<-	Interim Server Response
Secondary TRANSACTION request 1	->	
Secondary TRANSACTION request 2	->	
Secondary TRANSACTION request N	->	
	<-	TRANSACTION response 1
	<-	TRANSACTION response 2
	<-	TRANSACTION response m

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

Client	<->	Server
Primary TRANSACTION request	->	
	<-	TRANSACTION response 1
	<-	TRANSACTION response 2
	<-	TRANSACTION response m

The flow for the transaction protocol over a connectionless transport is:

1. The client sends the primary client request identifying 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 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 client then sends another buffer full of bytes to the server. The server responds with an interim server response. This step is repeated until all of the bytes are sent and received.

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).

6. The client responds with a transaction secondary request. The server sends back more response data. This step is repeated until all result bytes have been returned.

7. The client sends a sequenced request to the server such as `SMB_COM_ECHO`

The primary transaction request through the final response make up the complete transaction exchange, thus the *TID*, *PID*, *UID* and *MID* must remain constant and can be used as appropriate by both the server and the client. Of course, other SMB requests may intervene as well.

There are (at least) three ways that actual server responses have been observed to differ from what might be expected. First, some servers will send Pad bytes to move the DataOffset to a 2- or 4-byte boundary even if there are no data bytes; the point here is that the ByteCount must be used instead of ParameterOffset plus ParameterCount to infer the actual message length. Second, some servers always return MaxParameterCount bytes even if the particular Transact2 has no parameter response. Finally, in case of an error, some servers send the "traditional WordCount==0/ByteCount==0" response while others generate a Transact response format.

3.13 Valid SMB Requests by Negotiated Dialect

The following SMB messages may be exchanged by SMB clients and servers if the "`PC NETWORK PROGRAM 1.0`" dialect is negotiated:

SMB_COM_CREATE_DIRECTORY	SMB_COM_DELETE_DIRECTORY
SMB_COM_OPEN	SMB_COM_CREATE
SMB_COM_CLOSE	SMB_COM_FLUSH
SMB_COM_DELETE	SMB_COM_RENAME
SMB_COM_QUERY_INFORMATION	SMB_COM_SET_INFORMATION
SMB_COM_READ	SMB_COM_WRITE
SMB_COM_LOCK_BYTE_RANGE	SMB_COM_UNLOCK_BYTE_RANGE
SMB_COM_CREATE_TEMPORARY	SMB_COM_CREATE_NEW
SMB_COM_CHECK_DIRECTORY	SMB_COM_PROCESS_EXIT
SMB_COM_SEEK	SMB_COM_TREE_CONNECT
SMB_COM_TREE_DISCONNECT	SMB_COM_NEGOTIATE
SMB_COM_QUERY_INFORMATION_DISK	SMB_COM_SEARCH
SMB_COM_OPEN_PRINT_FILE	SMB_COM_WRITE_PRINT_FILE
SMB_COM_CLOSE_PRINT_FILE	SMB_COM_GET_PRINT_QUEUE

If the "LANMAN 1.0" dialect is negotiated, all of the messages in the previous list must be supported. Clients negotiating LANMAN 1.0 and higher dialects will probably no longer send SMB_COM_PROCESS_EXIT, and the response format for SMB_COM_NEGOTIATE is modified as well. New messages introduced with the LANMAN 1.0 dialect are:

SMB_COM_LOCK_AND_READ	SMB_COM_WRITE_AND_UNLOCK
SMB_COM_READ_RAW	SMB_COM_READ_MPX
SMB_COM_WRITE_MPX	SMB_COM_WRITE_RAW
SMB_COM_WRITE_COMPLETE	SMB_COM_WRITE_MPX_SECONDARY
SMB_COM_SET_INFORMATION2	SMB_COM_QUERY_INFORMATION2
SMB_COM_LOCKING_ANDX	SMB_COM_TRANSACTION
SMB_COM_TRANSACTION_SECONDARY	SMB_COM_IOCTL
SMB_COM_IOCTL_SECONDARY	SMB_COM_COPY
SMB_COM_MOVE	SMB_COM_ECHO
SMB_COM_WRITE_AND_CLOSE	SMB_COM_OPEN_ANDX
SMB_COM_READ_ANDX	SMB_COM_WRITE_ANDX
SMB_COM_SESSION_SETUP_ANDX	SMB_COM_TREE_CONNECT_ANDX
SMB_COM_FIND	SMB_COM_FIND_UNIQUE
SMB_COM_FIND_CLOSE	

The "LM1.2X002" dialect introduces these new SMBs:

SMB_COM_TRANSACTION2	SMB_COM_TRANSACTION2_SECONDARY
SMB_COM_FIND_CLOSE2	SMB_COM_LOGOFF_ANDX

"NT LM 0.12" dialect introduces:

SMB_COM_NT_TRANSACT	SMB_COM_NT_TRANSACT_SECONDARY
SMB_COM_NT_CREATE_ANDX	SMB_COM_NT_CANCEL
SMB_COM_NT_RENAME	SMB_COM_READ_BULK
SMB_COM_WRITE_BULK	SMB_COM_WRITE_BULK_DATA

4. SMB Requests

This section lists the "best practice" SMB requests -- ones that would permit a client to exercise full CIFS functionality and optimum performance when interoperating with a server speaking the latest dialect as of this writing ("NT LM 0.12").

Note that, as of this writing, no existing client restricts itself to only these requests, so no useful server can be written that supports just them. The classification is provided so that future clients will be written to permit future servers to be simpler.

4.1 Session Requests

4.1.1 NEGOTIATE: Negotiate Protocol

Client Request	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes; min = 2
struct {	
UCHAR BufferFormat;	0x02 -- Dialect
UCHAR DialectName[];	ASCII null-terminated string
} Dialects[];	

The Client sends a list of dialects that it 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. Implementers of particular protocols may choose to include, for example, version numbers in the string.

If the server does not understand any of the dialect strings, or if PC NETWORK PROGRAM 1.0 is the chosen dialect, the response format is

Server Response ==========	Description ==========
UCHAR WordCount; USHORT DialectIndex; USHORT ByteCount;	Count of parameter words = 1 Index of selected dialect Count of data bytes = 0

If the chosen dialect is greater than core up to and including `LANMAN2.1`, the protocol response format is

Server Response	Description
UCHAR WordCount;	Count of parameter words = 13
USHORT DialectIndex;	Index of selected dialect
USHORT SecurityMode;	Security mode: bit 0: 0 = share, 1 = user bit 1: 1 = use challenge/response authentication
USHORT MaxBufferSize;	Max transmit buffer size (>= 1024)
USHORT MaxMpxCount;	Max pending multiplexed requests
USHORT MaxNumberVcs;	Max VCs between client and server
USHORT RawMode;	Raw modes supported: bit 0: 1 = Read Raw supported bit 1: 1 = Write Raw supported
ULONG SessionKey;	Unique token identifying this session
SMB_TIME ServerTime;	Current time at server
SMB_DATE ServerDate;	Current date at server
USHORT ServerTimeZone;	Current time zone at server
USHORT EncryptionKeyLength;	MBZ if this is not LM2.1
USHORT Reserved;	MBZ
USHORT ByteCount	Count of data bytes
UCHAR EncryptionKey[];	The challenge encryption key
STRING PrimaryDomain[];	The server's primary domain

MaxBufferSize is the size of the largest message which the client can legitimately send to the server

If *bit0* of the *Flags* field is set in the negotiate response, this indicates the server supports the SMB_COM_LOCK_AND_READ and SMB_COM_WRITE_AND_UNLOCK client requests.

If the *SecurityMode* field indicates the server is running in *user mode*, the client must send appropriate SMB_COM_SESSION_SETUP_ANDX requests before the server will allow the client to access resources. If the *SecurityMode* fields indicates the client should use challenge/response authentication, the client should use the authentication mechanism specified in section 2.10.

Clients should submit no more than *MaxMpxCount* distinct unanswered SMBs to the server when using multiplexed reads or writes (see sections 5.13 and 5.24)

Clients using the "MICROSOFT NETWORKS 1.03" dialect use a different form of raw reads than documented here, and servers are better off setting *RawMode* in this response to 0 for such sessions.

If the negotiated dialect is "DOS LANMAN2.1" or "LANMAN2.1", then *PrimaryDomain* string should be included in this response.

If the negotiated dialect is `NT LM 0.12`, the response format is

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 17
USHORT DialectIndex;	Index of selected dialect
UCHAR SecurityMode;	Security mode: bit 0: 0 = share, 1 = user bit 1: 1 = encrypt passwords
USHORT MaxMpxCount;	Max pending multiplexed requests
USHORT MaxNumberVcs;	Max VCs between client and server
ULONG MaxBufferSize;	Max transmit buffer size
ULONG MaxRawSize;	Maximum raw buffer size
ULONG SessionKey;	Unique token identifying this session
ULONG Capabilities;	Server capabilities
ULONG SystemTimeLow;	System (UTC) time of the server (low).
ULONG SystemTimeHigh;	System (UTC) time of the server (high).
USHORT ServerTimeZone;	Time zone of server (min from UTC)
UCHAR EncryptionKeyLength;	Length of encryption key.
USHORT ByteCount;	Count of data bytes
UCHAR EncryptionKey[];	The challenge encryption key
UCHAR OemDomainName[];	The name of the domain (in OEM chars)

In addition to the definitions above, *MaxBufferSize* is the size of the largest message which the client can legitimately send to the server. If the client is using a connectionless protocol, *MaxBufferSize* must be set to the smaller of the server's internal buffer size and the amount of data which can be placed in a response packet.

MaxRawSize specifies the maximum message size the server can send or receive for SMB_COM_WRITE_RAW or SMB_COM_READ_RAW.

Connectionless clients must set *SID* to 0 in the SMB request header.

Capabilities allows the server to tell the client what it supports. The bit definitions are:

Capability Name	Encoding	Meaning
CAP_RAW_MODE	0x0001	The server supports SMB_COM_READ_RAW and SMB_COM_WRITE_RAW
CAP_MPX_MODE	0x0002	The server supports SMB_COM_READ_MPX and SMB_COM_WRITE_MPX
CAP_UNICODE	0x0004	The server supports Unicode strings
CAP_LARGE_FILES	0x0008	The server supports large files with 64 bit offsets
CAP_NT_SMBS	0x0010	The server supports the SMBs particular to the `NT LM 0.12` dialect
CAP_RPC_REMOTE_APIS	0x0020	The sever supports remote API requests via RPC
CAP_NT_STATUS	0x0040	The server can respond with 32 bit status codes in Status.NtStatus
CAP_LEVEL_II_OPLOCKS	0x0080	The server supports level 2 oplocks
CAP_LOCK_AND_READ	0x0100	The server supports the SMB_COM_LOCK_AND_READ SMB
CAP_NT_FIND	0x0200	
CAP_DFS	0x1000	This server is DFS aware
CAP_LARGE_READX	0x4000	The server supports SMB_COM_READ_ANDX requests which exceed the negotiated buffer size

4.1.1.1 Errors

SUCCESS/SUCCESS

ERRSRV/ERRerror

4.1.2 SESSION_SETUP_ANDX: Session Setup

This SMB is used to further "Set up" the session normally just established via the negotiate protocol.

One primary function is to perform a "user logon" in the case where the server is in *USER LEVEL* security mode. The *UID* in the SMB header is set by the client to be the userid desired for the *ACCOUNTNAME* and validated by the *ACCOUNTPASSWORD*.

If the negotiated protocol is prior to `NT LM 0.12`, the format of SMB_COM_SESSION_SETUP_ANDX is:

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 10
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT MaxBufferSize;	Client maximum buffer size
USHORT MaxMpxCount;	Actual maximum multiplexed pending requests
USHORT VcNumber;	0 = first (only), nonzero=additional VC number
ULONG SessionKey;	Session key (valid iff VcNumber != 0)
USHORT PasswordLength;	Account password size
ULONG Reserved;	Must be 0
USHORT ByteCount;	Count of data bytes; min = 0
UCHAR AccountPassword[];	Account Password
STRING AccountName[];	Account Name
STRING PrimaryDomain[];	Client's primary domain
STRING NativeOS[];	Client's native operating system
STRING NativeLanMan[];	Client's native LAN Manager type

and the response is:

Server Response ==========	Description ==========

UCHAR WordCount;	Count of parameter words = 3
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Action;	Request mode: bit0 = logged in as GUEST
USHORT ByteCount;	Count of data bytes
STRING NativeOS[];	Server's native operating system
STRING NativeLanMan[];	Server's native LAN Manager type
STRING PrimaryDomain[];	Server's primary domain

If the server is in "share level security mode", the account name and passwd should be ignored by the server.

If challenge/response authentication is not being used, *AccountPassword* should be a null terminated ASCII string with *PasswordLength* set to the string size including the null; the password will case insensitive. If challenge/response authentication is being used (see section 2.10), then *AccountPassword* will be the response to the server's challenge, and *PasswordLength* should be set to its length.

The server validates the name and password supplied and if valid, it registers the user identifier on this session as representing the specified *AccountName*. The *Uid* field in the SMB header will then be used to validate access on subsequent SMB requests. The SMB requests where permission checks are required are those which refer to a symbolically named resource such as SMB_COM_OPEN, SMB_COM_RENAME, SMB_COM_DELETE, etc.. The value of the *Uid* is relative to a specific client/server session so it is possible to have the same *Uid* value represent two different users on two different sessions at the server.

Multiple session setup commands may be sent to register additional users on this session. If the server receives an additional SMB_COM_SESSION_SETUP_ANDX, only the *Uid*, *AccountName* and *AccountPassword* fields need contain valid values (the server MUST ignore the other fields).

The client writes the name of its domain in *PrimaryDomain* if it knows what the domain name is. If the domain name is unknown, the client either encodes it as a NULL string, or as a question mark.

If *BIT0* of *Action* is set, this informs the client that although the server did not recognize the *AccountName*, it logged the user in as a guest. This is optional behavior by the server, and in any case one would ordinarily expect guest privileges to limited.

Another function of the Session Set Up protocol is to inform the server of the maximum values which will be utilized by this client. Here *MaxBufferSize* is the maximum message size which the client can receive. Thus although the server may support 16k buffers (as returned in the SMB_COM_NEGOTIATE response), if the client only has 4k buffers, the value of *MaxBufferSize* here would be 4096. The minimum allowable value for *MaxBufferSize* is 1024. The SMB_COM_NEGOTIATE response includes the server buffer size supported. Thus this is the maximum SMB message size which the client can send to the server. This size may be larger than the size returned to the server from the client via the SMB_COM_SESSION_SETUP_AND X

protocol which is the maximum SMB message size which the server may send to the client. Thus if the server's buffer size were 4k and the client's buffer size were only 2K, the client could send up to 4k (standard) write requests but must only request up to 2k for (standard) read requests.

The field, *MaxMpxCount* informs the server of the maximum number of requests which the client will have outstanding to the server simultaneously (see sections 5.13 and 5.24).

The *VcNumber* field specifies whether the client wants this to be the first VC or an additional VC.

The values for *MaxBufferSize*, *MaxMpxCount*, and *VcNumber* must be less than or equal to the maximum values supported by the server as returned in the SMB_COM_NEGOTIATE response.

If the server gets a SMB_COM_SESSION_SETUP_ANDX request with *VcNumber* of 0 and other VCs are still connected to that client, they will be aborted thus freeing any resources held by the server. This condition could occur if the client was rebooted and reconnected to the server before the transport level had informed the server of the previous VC termination.

If the negotiated SMB dialect is "NT LM 0.12" or later, the format of the response SMB is unchanged, but the request is:

Client Request	Description
UCHAR WordCount;	Count of parameter words = 13
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT MaxBufferSize;	Client's maximum buffer size
USHORT MaxMpxCount;	Actual maximum multiplexed pending requests
USHORT VcNumber;	0 = first (only), nonzero=additional VC number
ULONG SessionKey;	Session key (valid iff VcNumber != 0)
USHORT CaseInsensitivePasswordLength;	Account password size, ANSI
USHORT CaseSensitivePasswordLength;	Account password size, Unicode
ULONG Reserved;	must be 0
ULONG Capabilities;	Client capabilities
USHORT ByteCount;	Count of data bytes; min = 0
UCHAR CaseInsensitivePassword[];	Account Password, ANSI
UCHAR CaseSensitivePassword[];	Account Password, Unicode
STRING AccountName[];	Account Name, Unicode
STRING PrimaryDomain[];	Client's primary domain, Unicode
STRING NativeOS[];	Client's native operating system, Unicode
STRING NativeLanMan[];	Client's native LAN Manager type, Unicode

The client expresses its capabilities to the server encoded in the *Capabilities* field:

Capability Name	Encoding	Description
CAP_UNICODE	0x0004	The client can use UNICODE strings

CAP_LARGE_FILES	0x0008	The client can deal with files having 64 bit offsets
CAP_NT_SMBS	0x0010	The client understands the SMBs introduced with the `NT LM 0.12` dialect. Implies CAP_NT_FIND.
CAP_NT_FIND	0x0200	
CAP_NT_STATUS	0x0040	The client can receive 32 bit errors encoded in *STATUS.NTSTATUS*
CAP_LEVEL_II_OPLOCKS	0x0080	The client understands Level II oplocks

The entire message sent and received including the optional ANDX SMB must fit in the negotiated maximum transfer size. The following are the only valid SMB commands for *AndXCommand* for SMB_COM_SESSION_SETUP_ANDX

`SMB_COM_TREE_CONNECT_ANDX`	`SMB_COM_OPEN`
`SMB_COM_OPEN_ANDX`	`SMB_COM_CREATE`
`SMB_COM_CREATE_NEW`	`SMB_COM_CREATE_DIRECTORY`
`SMB_COM_DELETE`	`SMB_COM_DELETE_DIRECTORY`
`SMB_COM_FIND`	`SMB_COM_FIND_UNIQUE`
`SMB_COM_COPY`	`SMB_COM_RENAME`
`SMB_COM_NT_RENAME`	`SMB_COM_CHECK_DIRECTORY`
`SMB_COM_QUERY_INFORMATION`	`SMB_COM_SET_INFORMATION`
`SMB_COM_NO_ANDX_COMMAND`	`SMB_COM_OPEN_PRINT_FILE`
`SMB_COM_GET_PRINT_QUEUE`	`SMB_COM_TRANSACTION`

4.1.2.1 Errors

ERRSRV/ERRerror - no NEG_PROT issued

ERRSRV/ERRbadpw - password not correct for given username

ERRSRV/ERRtoomanyuids - maximum number of users per session exceeded

ERRSRV/ERRnosupport - chaining of this request to the previous one is not supported

4.1.3 LOGOFF_ANDX: User Logoff

This SMB is the inverse of SMB_COM_SESSION_SETUP_ANDX.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 2
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT ByteCount;	Count of data bytes = 0

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 2
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT ByteCount;	Count of data bytes = 0

The user represented by *UID* in the SMB header is logged off. The server closes all files currently open by this user, and invalidates any outstanding requests with this *UID*.

SMB_COM_SESSION_SETUP_ANDX is the only valid *AndXCommand*. for this SMB.

4.1.3.1 Errors

ERRSRV/invnid - TID was invalid

ERRSRV/baduid - UID was invalid

4.1.4 TREE_CONNECT_ANDX: Tree Connect

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 4
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Flags;	Additional information bit 0 set = disconnect Tid
USHORT PasswordLength;	Length of Password[]
USHORT ByteCount;	Count of data bytes; min = 3
UCHAR Password[];	Password
STRING Path[];	Server name and share name
STRING Service[];	Service name

This message generally functions just as SMB_COM_TREE_CONNECT, except it allows an AndXCommand to follow. Because *PASSWORD* may be encrypted, it is a variable length field with the length specified by *PASSWORDLENGTH*. If password encryption is not being used, *PASSWORD* should be a null terminated ASCII string with *PASSWORDLENGTH* set to the string size including the terminating null.

SERVICE is as described for SMB_COM_TREE_CONNECT.

If *BIT0* of *FLAGS* is set, the tree connection to *TID* in the SMB header should be disconnected. If this tree disconnect fails, the error should be ignored.

If the negotiated dialect is earlier than DOS `LANMAN2.1`, the response to this SMB is:

Server Response	Description
UCHAR WordCount;	Count of parameter words = 2
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT ByteCount;	Count of data bytes; min = 3

If the negotiated is DOS `LANMAN2.1` or later, the response to this SMB is:

Server Response	Description
UCHAR WordCount;	Count of parameter words = 3
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT OptionalSupport;	Optional support bits
USHORT ByteCount;	Count of data bytes; min = 3
UCHAR Service[];	Service type connected to. Always ANSII
STRING NativeFileSystem[];	Native file system for this tree

NativeFileSystem is the name of the filesystem; values to be expected include FAT, NTFS, etc.

OptionalSupport bits has the encoding:

Name ==========	Encoding =========	Description ==========
SMB_SUPPORT_SEARCH_BITS	0x0001	
SMB_SHARE_IS_IN_DFS	0x0002	

Some servers negotiate "DOS LANMAN2.1" dialect or later and still send the "downlevel" (i.e. wordcount==2) response. Valid AndX following commands are

```
SMB_COM_OPEN                    SMB_COM_OPEN_ANDX

SMB_COM_CREATE                  SMB_COM_CREATE_NEW

SMB_COM_CREATE_DIRECTORY        SMB_COM_DELETE

SMB_COM_DELETE_DIRECTORY        SMB_COM_FIND

SMB_COM_FIND_UNIQUE             SMB_COM_COPY

SMB_COM_RENAME                  SMB_COM_NT_RENAME

SMB_COM_CHECK_DIRECTORY         SMB_COM_QUERY_INFORMATION

SMB_COM_SET_INFORMATION         SMB_COM_GET_PRINT_QUEUE

SMB_COM_OPEN_PRINT_FILE         SMB_COM_NO_ANDX_COMMAND

SMB_COM_TRANSACTION
```

4.1.4.1 Errors

ERRDOS/ERRnomem

ERRDOS/ERRbadpath

ERRDOS/ERRinvdevice

ERRSRV/ERRaccess

ERRSRV/ERRbadpw

ERRSRV/ERRinvnetname

4.1.5 TREE_DISCONNECT: Tree Disconnect

This message informs the server that the client no longer wishes to access the resource connected to with a prior SMB_COM_TREE_CONNECT or SMB_COM_TREE_CONNECT_ANDX.

Client Request ================================	Description ================================
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

The resource sharing connection identified by *Tid* in the SMB header is logically disconnected from the server. *Tid* is invalidated; it will not be recognized if used by the client for subsequent requests. All locks, open files, etc. created on behalf of *Tid* are released.

Server Response ================================	Description ================================
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

4.1.5.1 Errors

ERRSRV/invnid
ERRSRV/baduid

4.1.6 TRANS2_QUERY_FS_INFORMATION: Get File System Information

This transaction requests information about a filesystem on the server.

Client Request ================================	Value ================================
WordCount; TotalParameterCount; MaxSetupCount; SetupCount; Setup[0];	15 2 or 4 0 1 or 2 TRANS2_QUERY_FS_INFORMATION

Parameter Block Encoding ================================	Description ================================
USHORT Information Level;	Level of information requested

The filesystem is identified by *Tid* in the SMB header.

MaxDataCount in the transaction request must be large enough to accommodate the response.

The encoding of the response parameter block depends on the *InformationLevel* requested. Information levels whose values are greater than 0x102 are mapped to corresponding calls to `NtQueryVolumeInformationFile` calls by the server. The two levels below 0x102 are described below. The requested information is placed in the *Data* portion of the transaction response.

InformationLevel ============================	Value ======	NtQueryVolumeInformationFile equivalent ============================
SMB_INFO_ALLOCATION	1	
SMB_INFO_VOLUME	2	
SMB_QUERY_FS_VOLUME_INFO	0x102	FileFsVolumeInformation
SMB_QUERY_FS_SIZE_INFO	0x103	FileFsSizeInformation
SMB_QUERY_FS_DEVICE_INFO	0x104	FileFsDeviceInformation
SMB_QUERY_FS_ATTRIBUTE_INFO	0x105	FileFsAttributeInformation

The following sections describe the *InformationLevel* dependent encoding of the data part of the transaction response for the non-NT-equivalent information levels.

4.1.6.1 SMB_INFO_ALLOCATION

Data Block Encoding	Description
ULONG idFileSystem;	File system identifier. NT server always returns 0
ULONG cSectorUnit;	Number of sectors per allocation unit
ULONG cUnit;	Total number of allocation units
ULONG cUnitAvail;	Total number of available allocation units
USHORT cbSector;	Number of bytes per sector

4.1.6.2 SMB_INFO_VOLUME

Data Block Encoding	Description
ULONG ulVsn;	Volume serial number
UCHAR cch;	Number of characters in Label
STRING Label;	The volume label

4.1.6.3 Errors

ERRSRV/invnid - TID was invalid

ERRSRV/baduid - UID was invalid

ERRHRD/ERRnotready - the file system has been removed

ERRHRD/ERRdata - disk I/O error

ERRSRV/ERRaccess - user does not have the right to perform this operation

ERRSRV/ERRinvdevice - resource identified by TID is not a file system

4.1.7 ECHO: Ping the Server

This request is used to test the connection to the server, and to see if the server is still responding.

Client Request	Description

UCHAR WordCount;	Count of parameter words = 1
USHORT EchoCount;	Number of times to echo data back
USHORT ByteCount;	Count of data bytes; min = 1
UCHAR Buffer[1];	Data to echo

Server Response	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT SequenceNumber;	Sequence number of this echo
USHORT ByteCount;	Count of data bytes; min = 4
UCHAR Buffer[1];	Echoed data

Each response echoes the data sent, though ByteCount may indicate no data If *EchoCount* is zero, no response is sent.

Tid in the SMB header is ignored, so this request may be sent to the server even if there are no valid tree connections to the server.

The flow for the ECHO protocol is:

Client Request	<->	Server Response
Echo Request (EchoCount == n)	->	
	<-	Echo Response 1
	<-	Echo Response 2
	<-	Echo Response n

If a client is communicating to the server over a connectionless transport, this SMB can be used to ensure there is some activity on the connection as required in the "Connectionless Transports" section elsewhere in this document.

4.1.7.1 Errors

ERRSRV/ERRbaduid - UID was invalid
ERRSRV/ERRnoaccess - session has not been established
ERRSRV/ERRnosupport - ECHO function is not supported

4.1.8 NT_CANCEL: Cancel request

This SMB allows a client to cancel a request currently pending at the server.

Client Request ================================	Description ================================
UCHAR WordCount; USHORT ByteCount;	No words are sent (== 0) No bytes (==0)

The *Sid*, *Uid*, *Pid*, *Tid*, and *Mid* fields of the SMB are used to locate an pending server request from this session. If a pending request is found, it is "hurried along" which may result in success or failure of the original request. No other response is generated for this SMB.

4.2 File Requests

4.2.1 NT_CREATE_ANDX: Create or Open File **

This command is used to create or open a file or a directory.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 24
UCHAR AndXCommand;	Secondary command; 0xFF = None
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
UCHAR Reserved;	Reserved (must be 0)
USHORT NameLength;	Length of Name[] in bytes
ULONG Flags;	Create bit set: 0x02 - Request an oplock 0x04 - Request a batch oplock 0x08 - Target of open must be directory
ULONG RootDirectoryFid;	If non-zero, open is relative to this directory
ACCESS_MASK DesiredAccess;	access desired
LARGE_INTEGER AllocationSize;	Initial allocation size
ULONG FileAttributes;	File attributes for creation **
ULONG ShareAccess;	Type of share access **
ULONG CreateDisposition;	Action to take if file exists or not
ULONG CreateOptions;	Options to use if creating a file
ULONG ImpersonationLevel;	Security QOS information
UCHAR SecurityFlags;	Security tracking mode flags: 0x1 - SECURITY_CONTEXT_TRACKING 0x2 - SECURITY_EFFECTIVE_ONLY
USHORT ByteCount;	Length of byte parameters
STRING Name[];	File to open or create

The *FIleAttributes* parameter specifies the file attributes and flags for the file. The parameter's value is the sum of allowed attributes and flags.

Any combination of the following attributes is acceptable, except all other file attributes override FILE_ATTRIBUTE_NORMAL:

FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It is not to be included in an ordinary directory listing.
FILE_ATTRIBUTE_READONLY	The file is read only. Applications can read the file but cannot write to it or delete it.
FILE_ATTRIBUTE_SYSTEM	The file is part of or is used exclusively by the operating system.

Any combination of the following flags is acceptable:

FILE_FLAG_WRITE_THROUGH

Instructs the operating system to write through any intermediate cache and go directly to the file. The operating system can still cache write operations, but cannot lazily flush them.

FILE_FLAG_NO_BUFFERING

Instructs the operating system to open the file with no intermediate buffering or caching. This can provide performance gains in some situations. An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

- File access must begin at offsets within the file that are integer multiples of the volume's sector size.
- File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations must be aligned on addresses in memory that are integer multiples of the volume's sector size. An application can determine a volume's sector size by calling the **GetDiskFreeSpace** function.

FILE_FLAG_RANDOM_ACCESS

Indicates that the file is accessed randomly. Windows uses this flag to optimize file caching.

FILE_FLAG_SEQUENTIAL_SCAN

Indicates that the file is to be accessed sequentially from beginning to end. Windows uses this flag to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur; however, correct operation is still guaranteed.

Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.

FILE_FLAG_DELETE_ON_CLOSE

Indicates that the operating system is to delete the file immediately after all of its handles have been closed.If you use this flag when you call **CreateFile**, then open the file again, and then close the handle for which you specified FILE_FLAG_DELETE_ON_CLOSE, the file will not be deleted until after you have closed the second and any other handle to the file.

FILE_FLAG_BACKUP_SEMANTICS

Windows NT only: Indicates that the file is being opened or created for a backup or restore operation. The operating system ensures that the calling process overrides file security checks, provided it has the necessary permission to do so. The relevant permissions are SE_BACKUP_NAME and SE_RESTORE_NAME.A Windows NT application can also set this flag to obtain a handle to a directory. A directory handle can be passed to some Win32 functions in place of a file handle.

FILE_FLAG_POSIX_SEMANTICS

Indicates that the file is to be accessed according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support such naming. Use care when using this option because files created with this flag may not be accessible by applications written for MS-DOS, Windows 3.*x*, or Windows NT.

The *desiredAccess* parameter can have one or more of the following flags:

GENERIC_READ	Specifies read access to the file. Data can be read from the file and the file pointer can be moved.
GENERIC_WRITE	Specifies write access to the file. Data can be written to the file and the file pointer can be moved.

If neither value is specified, it still allows an application to query attributes without actually accessing the file.

The *CreateDisposition* parameter can contain one of the following values:

CREATE_NEW	Creates a new file. The function fails if the specified file already exists.
CREATE_ALWAYS	Creates a new file. The function overwrites the file if it exists.
OPEN_EXISTING	Opens the file. The function fails if the file does not exist.
OPEN_ALWAYS	Opens the file, if it exists. If the file does not exist, act like CREATE_NEW.
TRUNCATE_EXISTING	Opens the file. Once opened, the file is truncated so that its size is zero bytes. The calling process must open the file with at least GENERIC_WRITE access. The function fails if the file does not exist.

The *ImpersonationLevel* parameter can contain one or more of the following values:

SECURITY_ANONYMOUS	Specifies to impersonate the client at the Anonymous impersonation level.
SECURITY_IDENTIFICATION	Specifies to impersonate the client at the Identification impersonation level.
SECURITY_IMPERSONATION	Specifies to impersonate the client at the Impersonation impersonation level.

SECURITY_DELEGATION Specifies to impersonate the client at the Delegation impersonation level.

The SecurityFlags parameter can have either of the following two flags set:

SECURITY_CONTEXT_TRACKING Specifies that the security tracking mode is dynamic. If this flag is not specified, Security Tracking Mode is static.

SECURITY_EFFECTIVE_ONLY Specifies that only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available. This flag allows the client to limit the groups and privileges that a server can use while impersonating the client.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 26
UCHAR AndXCommand; Secondary command;	0xFF = None
UCHAR AndXReserved;	MBZ
USHORT AndXOffset;	Offset to next command WordCount
UCHAR OplockLevel;	The oplock level granted
USHORT Fid;	The file ID
ULONG CreateAction;	The action taken
TIME CreationTime;	The time the file was created
TIME LastAccessTime;	The time the file was accessed
TIME LastWriteTime;	The time the file was last written
TIME ChangeTime;	The time the file was last changed
ULONG FileAttributes;	The file attributes
LARGE_INTEGER AllocationSize;	The number of byes allocated
LARGE_INTEGER EndOfFile;	The end of file offset
USHORT FileType;	
USHORT DeviceState;	state of IPC device (e.g. pipe)
BOOLEAN Directory;	TRUE if this is a directory
USHORT ByteCount;	= 0

The following SMBs may follow SMB_COM_NT_CREATE_ANDX:

SMB_COM_READ SMB_COM_READ_ANDX

SMB_COM_IOCTL

4.2.2 NT_TRANSACT_CREATE: Create or Open File with EAs or SD

This command is used to create or open a file or a directory, when EAs or an SD must be applied to the file.

Request Parameter Block Encoding ==========	Description ==========
ULONG Flags;	Creation flags (see below)
ULONG RootDirectoryFid;	Optional directory for relative open
ACCESS_MASK DesiredAccess;	Desired access (NT format)
LARGE_INTEGER AllocationSize;	The initial allocation size in bytes, if file created
ULONG FileAttributes;	The file attributes, (NT format)
ULONG ShareAccess;	The share access (NT format)
ULONG CreateDisposition;	Action to take if file exists or not (NT format)
ULONG CreateOptions;	Options for creating a new file (NT format)
ULONG SecurityDescriptorLength;	Length of SD in bytes
ULONG EaLength;	Length of EA in bytes
ULONG NameLength;	Length of name in characters
ULONG ImpersonationLevel;	Security QOS information (NT format)
UCHAR SecurityFlags;	Security QOS information (NT format)
STRING Name[NameLength];	The name of the file (not NULL terminated)
Data Block Encoding ==========	Description ==========
UCHAR SecurityDescriptor[SecurityDescriptorLength]; UCHAR ExtendedAttributes[EaLength];	

Creation Flag Name	Value	Description
NT_CREATE_REQUEST_OPLOCK	0x02	Level I oplock requested
NT_CREATE_REQUEST_OPBATCH	0x04	Batch oplock requested
NT_CREATE_OPEN_TARGET_DIR	0x08	Target for open is a directory

Output Parameter Block Encoding	Description
UCHAR OplockLevel;	The oplock level granted 0 - No oplock granted 1 - Exclusive oplock granted 2 - Batch oplock granted 3 - Level II oplock granted
UCHAR Reserved;	
USHORT Fid;	The file ID
ULONG CreateAction;	The action taken
ULONG EaErrorOffset;	Offset of the EA error
TIME CreationTime;	The time the file was created
TIME LastAccessTime;	The time the file was accessed
TIME LastWriteTime;	The time the file was last written
TIME ChangeTime;	The time the file was last changed
ULONG FileAttributes;	The file attributes
LARGE_INTEGER AllocationSize;	The number of byes allocated
LARGE_INTEGER EndOfFile;	The end of file offset
USHORT FileType;	
USHORT DeviceState;	state of IPC device (e.g. pipe)
BOOLEAN Directory;	TRUE if this is a directory

The above parameters are in native NT format.

4.2.3 CREATE_TEMPORARY: Create Temporary File

The server creates a data file in *DIRECTORY* relative to *TID* in the SMB header and assigns a unique name to it.

Client Request ================================	Server Response ================================
UCHAR WordCount;	Count of parameter words = 3
USHORT reserved;	Ignored by the server
UTIME CreationTime;	New file's creation time stamp
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING DirectoryName[];	Directory name

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 1
USHORT Fid;	File handle
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING Filename[];	File name

FID is the returned handle for future file access.

Filename is the name of the file which was created within the requested *Directory*. It is opened in compatibility mode with read/write access for the client.

Support of *CreationTime* by the server is optional.

4.2.4 READ_ANDX: Read Data

Client Request ================================	Description ==================================
UCHAR WordCount;	Count of parameter words = 10
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin read
USHORT MaxCount;	Max number of bytes to return
USHORT MinCount;	Min number of bytes to return
ULONG Reserved;	Must be 0
USHORT Remaining;	Bytes remaining to satisfy request
USHORT ByteCount;	Count of data bytes = 0

Large File Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 12
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin read
USHORT MaxCount;	Max number of bytes to return
USHORT MinCount;	Min number of bytes to return
ULONG Reserved;	Must be 0
USHORT Remaining;	Bytes remaining to satisfy request
ULONG OffsetHigh;	Upper 32 bits of offset
USHORT ByteCount;	Count of data bytes = 0

Server Response ================================	Description ==================================
UCHAR WordCount;	Count of parameter words = 12
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Remaining;	Bytes remaining to be read
USHORT DataCompactionMode;	
USHORT Reserved;	Reserved (must be 0)
USHORT DataLength;	Number of data bytes (min = 0)
USHORT DataOffset;	Offset (from header start) to data
USHORT Reserved[5];	Reserved (must be 0)
USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	
UCHAR Data[DataLength];	Data from resource

If the negotiated dialect is `NT LM 0.12` or later, the client may use the Large File version of the request. This version allows specification of 64 bit file offsets. If CAP_LARGE_READX was indicated by the server in the negotiate protocol response, the request's *MAXCOUNT* field may exceed the negotiated buffer size if *FID* refers to a disk file. The server may arbitrarily elect to return fewer than *MAXCOUNT* bytes in response.

MINCOUNT in the request is valid only if *FID* refers to a named pipe. *MINCOUNT* informs the server that at least *MINCOUNT* bytes should be returned, if possible.

REMAINING in the response is valid for pipes only. It is used to return the number of bytes currently available in the pipe excluding the bytes returned in this response. This information can then be used by the client to know when a subsequent (non blocking) read of the pipe may return some data. When a future read request is actually received by the server there may be more or less actual data in the pipe (more data has been written to the pipe or another reader drained it). If the information is currently not available or the request is NOT for a pipe, a -1 value should be returned.

The following SMBs may follow SMB_COM_READ_ANDX:

SMB_COM_CLOSE

4.2.5 READ_RAW: Read Raw

The SMB_COM_READ_RAW protocol is used to maximize the performance of reading a large block of data from the server to the client. This request can be applied to files and named pipes.

Client Request ================================	Description ==================================
UCHAR WordCount;	Count of parameter words = 8
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin read
USHORT MaxCount;	Max bytes to return (maximum 65535)
USHORT MinCount;	Min bytes to return (normally 0)
ULONG Timeout;	Wait time if named pipe
USHORT Reserved;	
USHORT ByteCount;	Count of data bytes = 0

FID identifies the resource being read, and may refer to a disk file or a named pipe.

TIMEOUT is the number of milliseconds to wait for completion *FID* refers to a named pipe.

When the client issues this request, the client must guarantee that there is (and will be) no other request to the server for the duration of the SMB_COM_READ_RAW. The server will respond, in one send, with the raw data being read. Thus the client is able to request up to 65,535 bytes of data and receive it directly into the user's buffer, since the server response has no header or trailer. 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 client's connection to the server for the duration of the request is that if other receives are present, 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 client receives as reported by the transport layer. 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 responds with a zero byte send. Upon receipt of a zero length response, the client should send a different type of request to the server. The response to that read will then tell the client that EOF was hit or identify the error condition.

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 results 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 client. Thus no SMB level confirmation protocol is required. If an error should occur at the clients end, all bytes must be received and thrown away. There is no need to inform the server of the error.

This message was introduced with `the LANMAN1.0` SMB dialect. Whether or not this request is supported is returned in the response to `SMB_COM_NEGOTIATE`.

The flow for reading a sequential file using `SMB_COM_READ_BOCK_RAW` is:

Client Request	Server Response
SMB_COM_OPEN file	Success
SMB_COM_READ_RAW	
	raw data returned
SMB_COM_READ_RAW	
	more raw data returned
SMB_COM_READ_RAW	
	short (or 0 length) response returned
SMB_COM_READ	
	0 bytes returned indicating EOF
SMB_COM_CLOSE	Success

SMB_COM_READ_RAW has no way to return errors. Because the response is raw data only, a zero length response indicates EOF, a read error or that the server is temporarily out of large buffers. The client should then retry using a different type of 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 a temporary server condition.

If the negotiated dialect is `NT LM 0.12` or later, and the response to the SMB_COM_NEGOTIATE SMB has CAP_LARGE_FILES set in the *CAPABILITIES* field, a new format of the SMB_COM_READ_RAW request is allowed which accommodates very large files having 64 bit offsets.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 10
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin read

USHORT MaxCount;	Max bytes to return (maximum 65535)
USHORT MinCount;	Min bytes to return (normally 0)
ULONG Timeout;	Wait time if named pipe
USHORT Reserved;	
ULONG OffsetHigh;	Upper 32 bits of offset
USHORT ByteCount;	Count of data bytes = 0

This form of the request is differentiated from the previous form of the request by the *WORDCOUNT* field. In this case, the final offset to read from is used by combining *OFFSETHIGH* and *OFFSET*, the resulting value can not be negative or the request will be rejected by the server.

SMB_COM_READ_RAW can not be used over connectionless transports.

4.2.6 WRITE_ANDX: Write Bytes to file or resource

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 12
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin write
ULONG Reserved;	Must be 0
USHORT WriteMode;	Write mode: 0 - write through 1 - return Remaining 2 - use WriteRawNamedPipe (n. pipes) 3 - "this is the start of the msg"
USHORT Remaining;	Bytes remaining to satisfy request
USHORT Reserved;	
USHORT DataLength;	Number of data bytes in buffer (>=0)
USHORT DataOffset;	Offset to data bytes
USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Data[DataLength];	Data to write

Large File Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 14
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin write
ULONG Reserved;	Must be 0
USHORT WriteMode;	Write mode bits: 0 - write through 1 - return Remaining 2 - use WriteRawNamedPipe (n. pipes) 3 - "this is the start of the msg"
USHORT Remaining;	Bytes remaining to satisfy request
USHORT Reserved;	
USHORT DataLength;	Number of data bytes in buffer (>=0)
USHORT DataOffset;	Offset to data bytes
ULONG OffsetHigh;	Upper 32 bits of offset
USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Data[DataLength];	Data to write

Server Response ==============================	Description ====================================
UCHAR WordCount;	Count of parameter words = 6
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Count;	Number of bytes written
USHORT Remaining;	Bytes remaining to be read in pipe
ULONG Reserved;	
USHORT ByteCount;	Count of data bytes = 0

A *BYTECOUNT* of 0 does not truncate the file. Rather a zero length write merely transfers zero bytes of information to the file. A request such as SMB_COM_WRITE must be used to truncate the file.

If *WRITEMODE* has bit0 set in the request and *FID* refers to a disk file, the response is not sent from the server until the data is on stable storage.

If *FID* refers to a named pipe, it is possible that the client wishes to transfer more data to the named pipe than the negotiated client and server buffer sizes permit. In this case, the data will arrive at the server in multiple SMB_COM_WRITE_ANDX messages. If *WRITEMODE BIT2* and *BIT3* are set, this is the first SMB of the sequence, and the total number of bytes which will be written are the sum of *DATALENGTH* and *REMAINING*. Subsequent SMB_COM_WRITE_ANDX messages having *WRITEMODE BIT2* set and possessing the same *PID* and *FID* will be gathered up in the server until *DATALENGTH* + *REMAINING* bytes have been received, at which time all the data is written to the named pipe in one message.

The return field *REMAINING* is valid only if *FID* refers to a named pipe, and *WRITEMODE* has *BIT1* set in the request. It is used to return the number of bytes currently available in the pipe. This information can then be used by the client to know when a subsequent (non blocking) read of the pipe may return some data. When the read request is actually received by the server there may be more or less actual data in the pipe (more data has been written to the pipe / device or another reader drained it).

If the negotiated dialect is `NT LM 0.12` or later, the Large File format of this SMB may be used to access portions of files requiring offsets expressed as 64 bits.

The following are the only valid *ANDXCOMMAND* values for this SMB:

SMB_COM_READ SMB_COM_READ_ANDX

SMB_COM_LOCK_AND_READ SMB_COM_WRITE_ANDX

SMB_COM_CLOSE

4.2.7 WRITE_RAW: Write Raw Bytes

The Write Block Raw protocol is used to maximize the performance of writing a large block of data from the client to the server. The Write Block Raw command's scope includes files, Named Pipes, and spooled output (can be used in place COM_WRITE_PRINT_FILE).

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 12
USHORT Fid;	File handle
USHORT Count;	Total bytes, including this buffer
USHORT Reserved;	
ULONG Offset;	Offset in file to begin write
ULONG Timeout;	
USHORT WriteMode;	Write mode: bit 0 - complete write to disk and send final result response bit 1 - return Remaining (pipe/dev) (see WriteAndX for #defines)
ULONG Reserved2;	
USHORT DataLength;	Number of data bytes this buffer
USHORT DataOffset;	Offset (from header start) to data
USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Data[];	Data (# = DataLength)

First Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 1
USHORT Remaining;	Bytes remaining to be read if pipe
USHORT ByteCount;	Count of data bytes = 0

Final Server Response ================================	Description ================================
UCHAR Command (in SMB header)	SMB_COM_WRITE_COMPLETE
UCHAR WordCount;	Count of parameter words = 1
USHORT Count;	Total number of bytes written
USHORT ByteCount;	Count of data bytes = 0

The first response format will be that of the final server response in the case where the server gets an error while writing the data sent along with the request. Thus *COUNT* is the number of bytes which did get written any time an error is returned. If an error occurs after the first response has been sent allowing the client to send the remaining data, the final response should not be sent unless write through is set. Rather the server should return this "write behind" error on the next access to the *FID*.

The client must guarantee that there is (and will be) no other request on the connection for the duration of this request. The server will reserve enough resources to receive the data and respond with a response SMB as defined above. The client then sends the raw data in one send. Thus the server is able to receive up to 65,535 bytes of data directly into the server buffer. 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 connection for the duration of the request is that if other receives are present on the connection, there is normally no way to guarantee that the data will be received into the correct server buffer, rather the data may fill one (or more) of the other buffers. Also if the client is sending other requests on the connection, a request may land in the buffer that the server has allocated for the this SMB's data.

Whether or not SMB_COM_WRITE_RAW is supported is returned in the response to SMB_COM_NEGOTIATE. SMB_COM_WRITE_RAW is not supported for connectionless clients.

When write through is not specified ((*WRITEMODE* & 01) == 0) this SMB is assumed to be a form of write behind. The transport layer guarantees delivery of all secondary requests from the client. Thus no "got the data you sent" SMB 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 ((*WRITEMODE* & 01) != 0), the server will receive the data, write it to disk and then send a final response indicating the result of the write. The total number of bytes written is also returned in this response in the *COUNT* field.

The flow for the SMB_COM_WRITE_RAW SMB is:

```
client -----> SMB_COM_WRITE_RAW request (optional data) >-------> server
client <-------------------< OK send (more) data <----------------- server
client  ---------------------> raw data >----------------------> server
client  <---< data on disk or error (write through only) <------- server
```

This protocol is set up such that the SMB_COM_WRITE_RAW request may also carry data. This is an optimization in that up to the server's buffer size (*MAXCOUNT* from SMB_COM_NEGOTIATE response), minus the size of the

SMB_COM_WRITE_RAW SMB 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 request (and wait for it to be on the disk or device if write through is set), prior to sending the response.

The specific responses error class ERRSRV, error codes ERRusempx and ERRusestd, indicate that the server is temporarily out of the resources needed to support the raw write of the remaining data, but that any data sent along with the request has been successfully written. The client should then write the remaining data using a different type of SMB write request, or delay and retry using SMB_COM_WRITE_RAW. If a write error occurs writing the initial data, it will be returned and the write raw request is implicitly denied.

The return field *REMAINING* is returned for named pipes only. It is used to return the number of bytes currently available in the pipe. This information can then be used by the client to know when a subsequent (non blocking) read of the pipe may return some data. Of course when the read request is actually received by the server there may be more or less actual data in the pipe (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 the server does not support this feature, a -1 value should be returned.

If the negotiated dialect is `NT LM 0.12` or later, and the response to the SMB_COM_NEGOTIATE SMB has CAP_LARGE_FILES set in the *CAPABILITIES* field, an additional request format is allowed which accommodates very large files having 64 bit offsets:

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 14
USHORT Fid;	File handle
USHORT Count;	Total bytes, including this buffer
USHORT Reserved;	
ULONG Offset;	Offset in file to begin write
ULONG Timeout;	
USHORT WriteMode;	Write mode: bit 0 - complete write to disk and send final result response bit 1 - return Remaining (pipe/dev)
ULONG Reserved2;	
USHORT DataLength;	Number of data bytes this buffer
USHORT DataOffset;	Offset (from header start) to data
ULONG OffsetHigh;	Upper 32 bits of offset

USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Data[];	Data (# = DataLength)

In this case the final offset in the file is formed by combining *OFFSETHIGH* and *OFFSET*, the resulting offset must not be negative.

4.2.8 LOCKING_ANDX: Lock or Unlock Byte Ranges

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

Client Request	Description
UCHAR WordCount;	Count of parameter words = 8
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Fid;	File handle
UCHAR LockType;	See LockType table below
UCHAR OplockLevel;	The new oplock level
ULONG Timeout;	Milliseconds to wait for unlock
USHORT NumberOfUnlocks;	Num. unlock range structs following
USHORT NumberOfLocks;	Num. lock range structs following
USHORT ByteCount;	Count of data bytes
LOCKING_ANDX_RANGE Unlocks[];	Unlock ranges
LOCKING_ANDX_RANGE Locks[];	Lock ranges

LockType Flag Name	Value	Description
LOCKING_ANDX_SHARED_LOCK	0x01	Read-only lock

LOCKING_ANDX_OPLOCK_RELEASE	0x02	Oplock break notification
LOCKING_ANDX_CHANGE_LOCKTYPE	0x04	Change lock type
LOCKING_ANDX_CANCEL_LOCK	0x08	Cancel outstanding request
LOCKING_ANDX_LARGE_FILES	0x10	Large file locking format

LOCKING_ANDX_RANGE Format ==========	
USHORT Pid;	PID of process "owning" lock
ULONG Offset;	Offset to bytes to [un]lock
ULONG Length;	Number of bytes to [un]lock

Large File LOCKING_ANDX_RANGE Format ==========	
USHORT Pid;	PID of process "owning" lock
USHORT Pad;	Pad to DWORD align (mbz)
ULONG OffsetHigh;	Offset to bytes to [un]lock (high)
ULONG OffsetLow;	Offset to bytes to [un]lock (low)
ULONG LengthHigh;	Number of bytes to [un]lock (high)
ULONG LengthLow;	Number of bytes to [un]lock (low)

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 2
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT ByteCount;	Count of data bytes = 0

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 permitted. Any process using the *FID* specified in this request's *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 locks 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.

If *NUMBEROFUNLOCKS* is non-zero, the *UNLOCKS* vector contains *NUMBEROFUNLOCKS* elements. Each element requests that a lock at *OFFSET* of *LENGTH* be released. If *NUMBEROFLOCKS* is nonzero, the *LOCKS* vector contains *NUMBEROFLOCKS* elements. Each element requests the acquisition of a lock at *OFFSET* of *LENGTH*.

TIMEOUT is the maximum amount of time to wait for the byte range(s) specified to become unlocked. A timeout value of 0 indicates that the server 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 of 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 client or none).

If *LOCKTYPE* has the `LOCKING_ANDX_SHARED_LOCK` flag set, the lock is specified as a shared lock. Locks for both read and write (where `LOCKING_ANDX_SHARED_LOCK` is clear) should be prohibited, but other shared locks should be permitted. If shared locks can not be supported by a server, the server should map the lock to a lock for both read and write. Closing a file with locks still in force causes the locks to be released in no defined order.

If *LOCKTYPE* has the `LOCKING_ANDX_LARGE_FILES` flag set and if the negotiated protocol is `NT LM 0.12` or later, then the Locks and Unlocks vectors are in the Large File `LOCKING_ANDX_RANGE` format. This allows specification of 64 bit offsets for very large files.

If the one and only member of the *LOCKS* vector has the `LOCKING_ANDX_CANCEL_LOCK` flag set in the *LOCKTYPE* field, the client is requesting the server to cancel a previously requested, but not yet responded to, lock.

If LockType has the `LOCKING_ANDX_CHANGE_LOCKTYPE` flag set, the client is requesting that the server atomically change the lock type from a shared lock to an exclusive lock or vice versa. If the server can not do this in an atomic fashion, the server must reject this request. NT and W95 servers do not support this capability.

Oplocks are described in the "Opportunistic Locks" section elsewhere in this document. A client requests an oplock by setting the appropriate bit in the SMB_COM_OPEN_ANDX request when the file is being opened in a mode which is not exclusive. The server responds by setting the appropriate bit in the response SMB indicating whether or not the oplock was granted. By granting the oplock, the server tells the client the file is currently only being used by this one client process at the current time. The client 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 client will be notified when any other process attempts to open or modify the oplocked file.

When another user attempts to open or otherwise modify the file which a client has oplocked, the server delays the second attempt and notifies the client via an `SMB_LOCKING_ANDX SMB` asynchronously sent from the server to the client. This message

has the LOCKING_ANDX_OPLOCK_RELEASE flag set indicating to the client that the oplock is being broken. *OPLOCKLEVEL* indicates the type of oplock the client now owns. If *OPLOCKLEVEL* is 0, the client possesses no oplocks on the file at all, if *OPLOCKLEVEL* is 1 the client possesses a Level II oplock. The client is expected to flush any dirty buffers to the server, submit any file locks and respond to the server with either an SMB_LOCKING_ANDX SMB having the LOCKING_ANDX_OPLOCK_RELEASE flag set, or with a file close if the file is no longer in use by the client. If the client sends an SMB_LOCKING_ANDX SMB with the LOCKING_ANDX_OPLOCK_RELEASE flag set and *NUMBEROFLOCKS* is zero, the server does not send a response. Since a close being sent to the server and break oplock notification from the server could cross on the wire, if the client gets an oplock notification on a file which it does not have open, that notification should be ignored.

Due to timing, the client could get an "oplock broken" notification in a user's data buffer as a result of this notification crossing on the wire with a SMB_COM_READ_RAW request. The client must detect this (use length of msg, "FFSMB", MID of -1 and *COMMAND* of SMB_COM_LOCKING_ANDX) and honor the "oplock broken" notification as usual. The server must also note on receipt of an SMB_COM_READ_RAW request that there is an outstanding (unanswered) "oplock broken" notification to the client and return a zero length response denoting failure of the read raw request. The client 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.

The entire message sent and received including the optional second protocol must fit in the negotiated maximum transfer size. The following are the only valid SMB commands for *ANDXCOMMAND* for SMB_COM_LOCKING_ANDX:

SMB_COM_READ	SMB_COM_READ_ANDX
SMB_COM_WRITE	SMB_COM_WRITE_ANDX
SMB_COM_FLUSH	

4.2.9 SEEK: Seek in File

The seek message is sent to set the current file pointer for *FID*.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 4
USHORT Fid;	File handle
USHORT Mode;	Seek mode: 0 = from start of file 1 = from current position 2 = from end of file
LONG Offset;	Relative offset
USHORT ByteCount;	Count of data bytes = 0

The starting point of the seek is set by *MODE*:

0 seek from start of file
1 seek from current file pointer
2 seek from end of file

The "current position" 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.

Server Response ================================	Description ================================
UCHAR WordCount; ULONG Offset; USHORT ByteCount;	Count of parameter words = 2 Offset from start of file Count of data bytes = 0

The response returns the new file pointer in *OFFSET* which is 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 sets the current file pointer to start of the file.

This request should generally only be issued by clients wishing to find the size of a file, since all read and write requests include the read or write file position as part of the SMB. This request is inappropriate for very large files, as the offsets specified are only 32 bits. A seek which results in an Offset which can not be expressed in 32 bits returns the least significant .

4.2.10 FLUSH: Flush File

The flush SMB is sent to ensure all data and allocation information for the corresponding file has been written to stable storage. When the *FID* has a value -1 (hex FFFF) the server performs a flush for all file handles associated with the client and *PID*. The response is not sent until the writes are complete.

Client Request ================================	Description ================================
UCHAR WordCount; USHORT Fid; USHORT ByteCount;	Count of parameter words = 1 File handle Count of data bytes = 0

This client request is probably expensive to perform at the server, since the server's operating system is generally scheduling disk writes is a way which is optimal for the system's read and write activity integrated over the entire population of clients. This message from a client "interferes" with the server's ability to optimally schedule the disk activity; clients are discouraged from overuse of this SMB request.

Server Response	Description
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

4.2.11 CLOSE: Close File

The close message is sent to invalidate a file handle for the requesting process. All locks or other resources held by the requesting process on the file should be released by the server. The requesting process can no longer use *FID* for further file access requests.

Client Request	Description
UCHAR WordCount; USHORT Fid; UTIME LastWriteTime USHORT ByteCount;	Count of parameter words = 3 File handle Time of last write Count of data bytes = 0

If *LASTWRITETIME* and *LASTWRITEDATE* are 0, the server should allow its local operating system to set the file's times. Otherwise, the server should set the time to the values requested. Failure to set the times, even if requested by the client in the request message, should not result in an error response from the server.

If *FID* refers to a print spool file, the file should be spooled to the printer at this time.

Server Response	Description
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

4.2.12 DELETE: Delete File

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

Client Request	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT SearchAttributes;	
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING FileName[];	File name

Multiple files may be deleted in response to a single request as SMB_COM_DELETE supports wildcards

SEARCHATTRIBUTES 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. Attributes are described in the "Attribute Encoding" section of this document.

If *BIT0* of the *FLAGS2* field of the SMB header is set, a pattern is passed in, and the file has a long name, then the passed pattern much match the long file name for the delete to succeed. If *BIT0* is clear, a pattern is passed in, and the file has a long name, then the passed pattern must match the file's short name for the deletion to succeed.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

4.2.13 RENAME: Rename File

The rename file message is sent to change the name of a file.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT SearchAttributes;	Target file attributes
USHORT ByteCount;	Count of data bytes; min = 4
UCHAR BufferFormat1;	0x04
STRING OldFileName[];	Old file name

UCHAR BufferFormat2; STRING NewFileName[];	0x04 New file name

Files *OldFileName* must exist and *NewFileName* 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 wildcards in the file name (last component of the pathname).

SearchAttributes indicates the attributes that the target file(s) must have. If *SearchAttributes* 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. The encoding of *SearchAttributes* is described in the "Attribute Encoding" section of this document.

Server Response ==========	Description ==========
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

4.2.14 MOVE: Rename File

The source file is copied to the destination and the source is subsequently deleted.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 3
USHORT Tid2;	Second (target) file id
USHORT OpenFunction;	what to do if target file exists
USHORT Flags;	Flags to control move operations: 0 - target must be a file 1 - target must be a directory 2 - reserved (must be 0) 3 - reserved (must be 0) 4 - verify all writes
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR Format1;	0x04
STRING OldFileName[];	Old file name
UCHAR FormatNew;	0x04
STRING NewFileName[];	New file name

OldFileName is copied to *NewFileName*, then *OldFileName* is deleted. Both *OldFileName* and *NewFileName* must refer to paths on the same server. *NewFileName* can refer to either a file or a directory. All file components except the last must exist; directories will not be created.

NewFileName can be required to be a file or a directory by the Flags field.

The *TID* in the header is associated with the source while *TID2* is associated with the destination. These fields may contain the same or differing valid values. *TID2* can be set to -1 indicating that this is to be the same *TID* as in the SMB header. This allows use of the move protocol with `SMB_TREE_CONNECT_ANDX`.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 1

USHORT Count;	Number of files moved
USHORT ByteCount;	Count of data bytes; min = 0
UCHAR ErrorFileFormat;	0x04 (only if error)
STRING ErrorFileName[];	Pathname of file where error occurred

The source path must refer to an existing file or files. Wildcards are permitted. Source files specified by wildcards are processed until an error is encountered. If an error is encountered, the expanded name of the file is returned in ErrorFileName. Wildcards are not permitted in *NEWFILENAME*.

OpenFunction controls what should happen if the destination file exists. If (*OpenFunction* & 0x30) == 0, the operation should fail if the destination exists. If (*OpenFunction* & 0x30) == 0x20, the destination file should be overwritten.

4.2.15 COPY: Copy File

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 3
USHORT Tid2;	Second (target) path TID
USHORT OpenFunction;	What to do if target file exists
USHORT Flags;	Flags to control copy operation: bit 0 - target must be a file bit 1 - target must be a dir. bit 2 - copy target mode: 0 = binary, 1 = ASCII bit 3 - copy source mode: 0 = binary, 1 = ASCII bit 4 - verify all writes bit 5 - tree copy
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR SourceFileNameFormat;	0x04

STRING SourceFileName;	Pathname of source file
UCHAR TargetFileNameFormat;	0x04
STRING TargetFileName;	Pathname of target file

The file at *SourceName* is copied to *TargetFileName*, both of which must refer to paths on the same server.

The *TID* in the header is associated with the source while *TID2* is associated with the destination. These fields may contain the same or differing valid values. *TID2* can be set to -1 indicating that this is to be the same *TID* as in the SMB header. This allows use of the move protocol with `SMB_TREE_CONNECT_ANDX`.

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 1
USHORT Count;	Number of files copied
USHORT ByteCount;	Count of data bytes; min = 0
UCHAR ErrorFileFormat;	0x04 (only if error)
STRING ErrorFileName;	

The source path must refer to an existing file or files. Wildcards are permitted. Source files specified by wildcards are processed until an error is encountered. If an error is encountered, the expanded name of the file is returned in ErrorFileName. Wildcards are not permitted in *TargetFileName*. *TargetFileName* can refer to either a file or a directory.

The destination can be required to be a file or a directory by the bits in *FLAGS*. If neither *BIT0* nor *BIT1* are set, the destination may be either a file or a directory. *Flags* also controls the copy mode. In a binary copy for the source, the copy stops the first time an EOF (control-Z) is encountered. In a binary copy for the target, the server must make sure that there is exactly one EOF in the target file and that it is the last character of the file.

If the destination is a file and the source contains wildcards, the destination file will either be truncated or appended to at the start of the operation depending on bits in *OpenFunction* (see section 3.7). Subsequent files will then be appended to the file.

If the negotiated dialect is `LM1.2X002` or later, *BIT5* of *Flags* is used to specify a tree copy on the remote server. When this option is selected the destination must not be an existing file and the source mode must be binary. A request with *BIT5* set and either *BIT0* or *BIT3* set is therefore an error. When the tree copy mode is selected, the *Count* field in the server response is undefined.

4.2.16 TRANS2_QUERY_PATH_INFORMATION: Get File Attributes given Path

This request is used to get information about a specific file or subdirectory.

Client Request ==========	Value ==========
WordCount MaxSetupCount SetupCount Setup[0]	15 0 1 TRANS2_QUERY_PATH_INFORMATION
Parameter Block Encoding ==========	Description ==========
USHORT InformationLevel; ULONG Reserved; STRING FileName;	Level of information requested Must be zero File or directory name

The following InformationLevels may be requested:

Information Level ==========	Value =====	NtQueryInformationFile Equivalent ==========
SMB_INFO_STANDARD SMB_INFO_QUERY_EA_SIZE SMB_INFO_QUERY_EAS_FROM_LIST SMB_INFO_QUERY_ALL_EAS SMB_INFO_IS_NAME_VALID SMB_QUERY_FILE_BASIC_INFO SMB_QUERY_FILE_STANDARD_INFO	1 2 3 4 6 0x101 0x102	 FileBasicInformation FileStandardInformation

SMB_QUERY_FILE_EA_INFO	0x103	FileEaInformation
SMB_QUERY_FILE_NAME_INFO	0x104	FileNameInformation
SMB_QUERY_FILE_ALL_INFO	0x107	FileAllInformation
SMB_QUERY_FILE_ALT_NAME_INFO	0x108	FileAlternateNameInformation
SMB_QUERY_FILE_STREAM_INFO	0x109	FileStreamInformation
SMB_QUERY_FILE_COMPRESSION_INFO	0x10B	FileCompressionInformation

Information levels whose values are greater than 0x101 are mapped to corresponding calls to NtQueryInformationFile calls by the server. The five levels below 0x101 are described below. The requested information is placed in the Data portion of the transaction response. For the NT equivalent responses, the transaction response has 1 parameter word which should be ignored by the client.

4.2.16.1 SMB_INFO_STANDARD & SMB_INFO_QUERY_EA_SIZE

Data Block Encoding	Description
SMB_DATE CreationDate;	Date when file was created
SMB_TIME CreationTime;	Time when file was created
SMB_DATE LastAccessDate;	Date of last file access
SMB_TIME LastAccessTime;	Time of last file access
SMB_DATE LastWriteDate;	Date of last write to the file
SMB_TIME LastWriteTime;	Time of last write to the file
ULONG DataSize;	File Size
ULONG AllocationSize;	Size of filesystem allocation unit
USHORT Attributes;	File Attributes
ULONG EaSize;	Size of file's EA information (`SMB_INFO_QUERY_EA_SIZE`)

4.2.16.2 SMB_INFO_QUERY_EAS_FROM_LIST & SMB_INFO_QUERY_ALL_EAS

Response Field	Value

MaxDataCount	Length of FEAlist found (minimum value is 4)
Parameter Block Encoding ==========	Description ==========
USHORT EaErrorOffset	Offset into EAList of EA error
Data Block Encoding ==========	Description ==========
ULONG ListLength; UCHAR EaList[]	Length of the remaining data The extended attributes list

4.2.16.3 SMB_INFO_IS_NAME_VALID

This requests checks to see if the name of the file contained in the request's *Data* field has a valid path syntax. No parameters or data are returned on this information request. An error is returned if the syntax of the name is incorrect. *SUCCESS* indicates the server accepts the path syntax, but it does not ensure the file or directory actually exists.

4.2.16.4 SMB_QUERY_FILE_BASIC_INFO

```
typedef struct {
    TIME CreationTime;
    TIME LastAccessTime;
    TIME LastWriteTime;
    TIME ChangeTime;
    ULONG FileAttributes;
} FILE_BASIC_INFORMATION;
```

4.2.16.5 SMB_QUERY_FILE_STANDARD_INFO

```
typedef struct {
    LARGE_INTEGER AllocationSize;
    LARGE_INTEGER EndOfFile;
    ULONG NumberOfLinks;
    BOOLEAN DeletePending;
    BOOLEAN Directory;
} FILE_STANDARD_INFORMATION;
```

4.2.16.6 SMB_QUERY_FILE_EA_INFO

```
typedef struct {
    ULONG EaSize;
} FILE_EA_INFORMATION;
```

4.2.16.7 SMB_QUERY_FILE_NAME_INFO

```
typedef struct {
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NAME_INFORMATION;
```

4.2.16.8 SMB_QUERY_FILE_ALL_INFO

```
typedef struct {
    LARGE_INTEGER IndexNumber;
} FILE_INTERNAL_INFORMATION;

typedef ULONG ACCESS_MASK;
```

The **ACCESS_MASK** structure is one 32 bit value containing standard, specific, and generic rights. These rights are used in access-control entries (ACEs) and are the primary means of specifying the requested or granted access to an object.

The bits in this value are allocated as follows:

Bits	**Meaning**
0 through 15	Specific rights. Contains the access mask specific to the object type associated with the mask.
16 through 23	Standard rights. Contains the object's standard access rights and can be a combination of the following predefined flags:

Bit	**Flag**	**Meaning**
16	DELETE	Delete access
17	READ_CONTROL	Read access to the owner, group, and discretionary access-control list (ACL) of the security descriptor
18	WRITE_DAC	Write access to the discretionary access-control list (ACL)
19	WRITE_OWNER	Write access to owner
20	SYNCHRONIZE	Windows NT: Synchronize access

Bits	Meaning
24	Access system security (ACCESS_SYSTEM_SECURITY). This flag is not a typical access type. It is used to indicate access to a system ACL. This type of access requires the calling process to have a specific privilege.
25	Maximum allowed (MAXIMUM_ALLOWED)
26 through 27	Reserved
28	Generic all (GENERIC_ALL)
29	Generic execute (GENERIC_EXECUTE)
30	Generic write (GENERIC_WRITE)
31	Generic read (GENERIC_READ)

```
typedef struct {
    ACCESS_MASK AccessFlags;
} FILE_ACCESS_INFORMATION;

typedef struct {
    LARGE_INTEGER CurrentByteOffset;
} FILE_POSITION_INFORMATION;

typedef struct {
    ULONG Mode;
} FILE_MODE_INFORMATION;

typedef struct {
    ULONG AlignmentRequirement;
} FILE_ALIGNMENT_INFORMATION;

typedef struct _FILE_ALL_INFORMATION {
    FILE_BASIC_INFORMATION BasicInformation;
    FILE_STANDARD_INFORMATION StandardInformation;
    FILE_INTERNAL_INFORMATION InternalInformation;
    FILE_EA_INFORMATION EaInformation;
    FILE_ACCESS_INFORMATION AccessInformation;
    FILE_POSITION_INFORMATION PositionInformation;
    FILE_MODE_INFORMATION ModeInformation;
    FILE_ALIGNMENT_INFORMATION AlignmentInformation;
    FILE_NAME_INFORMATION NameInformation;
} FILE_ALL_INFORMATION;
```

4.2.16.9 SMB_QUERY_FILE_ALT_NAME_INFO

This infornation level returns a FILE_NAME_INFORMATION structure.

4.2.16.10 SMB_QUERY_FILE_STREAM_INFO

```
typedef struct {
    ULONG NextEntryOffset;
    ULONG StreamNameLength;
    LARGE_INTEGER StreamSize;
    LARGE_INTEGER StreamAllocationSize;
    WCHAR StreamName[1];
} FILE_STREAM_INFORMATION;
```

4.2.16.11 SMB_QUERY_FILE_COMPRESSION_INFO

```
typedef struct {
    LARGE_INTEGER CompressedFileSize;
    USHORT CompressionFormat;
    UCHAR CompressionUnitShift;
    UCHAR ChunkShift;
    UCHAR ClusterShift;
    UCHAR Reserved[3];
} FILE_COMPRESSION_INFORMATION;
```

4.2.17 TRANS2_SET_PATH_INFORMATION: Set File Attributes given Path

This request is used to set information about a specific file or subdirectory.

Client Request	Value
WordCount	15
MaxSetupCount	0
SetupCount	1
Setup[0]	TRANS2_SET_PATH_INFORMATION
Parameter Block Encoding	Description
USHORT InformationLevel;	Level of information to set
ULONG Reserved;	Must be zero
STRING FileName;	File or directory name

The following *INFORMATIONLEVELS* may be set:

Information Level	Value
SMB_INFO_STANDARD	1
SMB_INFO_QUERY_EA_SIZE	2
SMB_INFO_QUERY_ALL_EAS	4

The response formats are:

4.2.17.1 SMB_INFO_STANDARD & SMB_INFO_QUERY_EA_SIZE

Parameter Block Encoding ==========	Description ==========
USHORT Reserved	0
Data Block Encoding ==========	Description ==========
SMB_DATE CreationDate; SMB_TIME CreationTime; SMB_DATE LastAccessDate; SMB_TIME LastAccessTime; SMB_DATE LastWriteDate; SMB_TIME LastWriteTime; ULONG DataSize; ULONG AllocationSize; USHORT Attributes; ULONG EaSize;	Date when file was created Time when file was created Date of last file access Time of last file access Date of last write to the file Time of last write to the file File Size Size of filesystem allocation unit File Attributes Size of file's EA information (SMB_INFO_QUERY_EA_SIZE)

4.2.17.2 SMB_INFO_QUERY_ALL_EAS

Response Field ==========	Value ==========
MaxDataCount	Length of FEAlist found (minimum value is 4)
Parameter Block Encoding	Description

==========	==========
USHORT EaErrorOffset	Offset into EAList of EA error
Data Block Encoding ==========	Description ==========
ULONG ListLength; UCHAR EaList[]	Length of the remaining data The extended attributes list

4.2.18 TRANS2_QUERY_FILE_INFORMATION: Get File Attributes Given FID

This request is used to get information about a specific file or subdirectory given a handle to it.

Client Request ==========	Value ==========
WordCount MaxSetupCount SetupCount Setup[0]	15 0 1 TRANS2_QUERY_FILE_INFORMATION
Parameter Block Encoding ==========	Description ==========
USHORT Fid; USHORT InformationLevel;	Handle of file for request Level of information requested

The available information levels, as well as the format of the response are identical to `TRANS2_QUERY_PATH_INFORMATION`.

4.2.19 TRANS2_SET_FILE_INFORMATION: Set File Attributes Given FID

This request is used to set information about a specific file or subdirectory given a handle to the file or subdirectory.

Client Request	Value
WordCount MaxSetupCount SetupCount Setup[0]	15 0 1 TRANS2_SET_FILE_INFORMATION
Parameter Block Encoding	**Description**
USHORT Fid; USHORT InformationLevel; USHORT Reserved;	Handle of file for request Level of information requested Ignored by the server

The following *INFORMATIONLEVELS* may be set:

Information Level	Value	NtSetFileInformation equiv.
SMB_INFO_STANDARD SMB_INFO_QUERY_EA_SIZE SMB_SET_FILE_BASIC_INFO SMB_SET_FILE_DISPOSITION_INFO SMB_SET_FILE_ALLOCATION_INFO SMB_SET_FILE_END_OF_FILE_INFO	1 2 0x101 0x102 0x103 0x104	 FileBasicInformation FileDispositionInformation FileAllocationInformation FileEndOfFileInformation

Information levels whose values are greater than 0x100 are mapped to corresponding calls to NtSetInformationFile calls by the server. The two levels below 0x100 are as described in the `NT_SET_PATH_INFORMATION` transaction. The requested information is placed in the Data portion of the transaction response. For the NT equivalent responses, the transaction response has 1 parameter word which should be ignored by the client.

4.3 Directory Requests

4.3.1 TRANS2_CREATE_DIRECTORY: Create Directory (optional EAs)

This requests the server to create a directory relative to *TID* in the SMB header, optionally assigning extended attributes to it.

Client Request ==========	Value ==========
WordCount MaxSetupCount SetupCount Setup[0]	15 0 1 TRANS2_CREATE_DIRECTORY
Parameter Block Encoding ==========	Description ==========
ULONG Reserved; STRING Name[]; UCHAR Data[];	Reserved--must be zero Directory name to create Optional FEAList for the new directory

Response Parameter Block ==========	Description ==========
USHORT EaErrorOffset	Offset into FEAList of first error which occurred while setting EAs

4.3.2 DELETE_DIRECTORY: Delete Directory

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.

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 0

USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING DirectoryName[];	Directory name

The directory to be deleted cannot be the root of the share specified by *TID*.

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

4.3.3 CHECK_DIRECTORY: Check Directory

This SMB is used to verify that a path exists and is a directory. No error is returned if the given path exists and the client has read access to it. Client 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 for a particular client. The client must always supply full pathnames relative to the *TID* in the SMB header.

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING DirectoryPath[];	Directory path

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

DOS clients, in particular, depend on the SMB_ERR_BAD_PATH return code if the directory is not found.

4.3.4 TRANS2_FIND_FIRST2: Search Directory using Wildcards

Client Request	Value
WordCount	15
TotalDataCount	Total size of extended attribute list
SetupCount	1
Setup[0]	TRANS2_FIND_FIRST2
Parameter Block Encoding	**Description**
USHORT SearchAttributes;	
USHORT SearchCount;	Maximum number of entries to return
USHORT Flags;	Additional information: Bit 0 - close search after this request Bit 1 - close search if end of search reached Bit 2 - return resume keys for each entry found Bit 3 - continue search from previous ending place Bit 4 - find with backup intent
USHORT InformationLevel;	
ULONG SearchStorageType;	
STRING FileName;	Pattern for the search
UCHAR Data[TotalDataCount]	FEAList if InformationLevel is `QUERY_EAS_FROM_LIST`

Response Parameter Block	Description
USHORT Sid;	Search handle

USHORT SearchCount;	Number of entries returned
USHORT EndOfSearch;	Was last entry returned?
USHORT EaErrorOffset;	Offset into EA list if EA error
USHORT LastNameOffset;	Offset into data to file name of last entry, if server needs it to resume search; else 0
UCHAR Data[TotalDataCount]	Level dependent info about the matches found in the search

This request allows the client to search for the file(s) which match the file specification. The search can be continued if necessary with `TRANS2_FIND_NEXT2`. There are numerous levels of information which may be obtained for the returned files, the desired level is specified in the *InformationLevel* field of the request.

InformationLevel Name ==========	Value ==========
`SMB_INFO_STANDARD`	1
`SMB_INFO_QUERY_EA_SIZE`	2
`SMB_INFO_QUERY_EAS_FROM_LIST`	3
`SMB_FIND_FILE_DIRECTORY_INFO`	0x101
`SMB_FIND_FILE_FULL_DIRECTORY_INFO`	0x102
`SMB_FIND_FILE_NAMES_INFO`	0x103
`SMB_FIND_FILE_BOTH_DIRECTORY_INFO`	0x104

Information levels whose values are greater than 0x101 are mapped to corresponding calls to `NtQueryInformationFile` calls by the server. The three levels below 0x101 are described below. The requested information is placed in the *DATA* portion of the transaction response.

A client which does not support long names can only request `SMB_INFO_STANDARD`. The following sections detail the data returned for each InformationLevel.

4.3.4.1 SMB_INFO_STANDARD

Response Field	Description
SMB_DATE CreationDate;	Date when file was created
SMB_TIME CreationTime;	Time when file was created
SMB_DATE LastAccessDate;	Date of last file access
SMB_TIME LastAccessTime;	Time of last file access
SMB_DATE LastWriteDate;	Date of last write to the file
SMB_TIME LastWriteTime;	Time of last write to the file
ULONG DataSize;	File Size
ULONG AllocationSize;	Size of filesystem allocation unit
USHORT Attributes;	File Attributes
UCHAR FileNameLength;	Length of filename in bytes
STRING FileName;	Name of found file

4.3.4.2 SMB_INFO_QUERY_EA_SIZE

Response Field	Description
SMB_DATE CreationDate;	Date when file was created
SMB_TIME CreationTime;	Time when file was created
SMB_DATE LastAccessDate;	Date of last file access
SMB_TIME LastAccessTime;	Time of last file access
SMB_DATE LastWriteDate;	Date of last write to the file
SMB_TIME LastWriteTime;	Time of last write to the file
ULONG DataSize;	File Size
ULONG AllocationSize;	Size of filesystem allocation unit
USHORT Attributes;	File Attributes

ULONG EaSize;	Size of file's EA information
UCHAR FileNameLength;	Length of filename in bytes
STRING FileName;	Name of found file

4.3.4.3 SMB_INFO_QUERY_EAS_FROM_LIST

This request returns the same information as `SMB_INFO_QUERY_EA_SIZE`, but only for files which have an EA list which match the EA information in the *DATA* part of the request.

4.3.4.4 SMB_FIND_FILE_DIRECTORY_INFO

Response Field	Description
ULONG NextEntryOffset;	Offset from this structure to beginning of next one
ULONG FileIndex;	
LARGE_INTEGER CreationTime;	file creation time
LARGE_INTEGER LastAccessTime;	last access time
LARGE_INTEGER LastWriteTime;	last write time
LARGE_INTEGER ChangeTime;	last attribute change time
LARGE_INTEGER EndOfFile;	file size
LARGE_INTEGER AllocationSize;	size of filesystem allocation information
ULONG FileAttributes;	NT style encoding of file attributes
ULONG FileNameLength;	Length of filename in bytes
STRING FileName;	Name of the file

4.3.4.5 SMB_FIND_FILE_FULL_DIRECTORY_INFO

Response Field	Description
ULONG NextEntryOffset;	Offset from this structure to beginning of next one
ULONG FileIndex;	
LARGE_INTEGER CreationTime;	file creation time
LARGE_INTEGER LastAccessTime;	last access time
LARGE_INTEGER LastWriteTime;	last write time
LARGE_INTEGER ChangeTime;	last attribute change time
LARGE_INTEGER EndOfFile;	file size
LARGE_INTEGER AllocationSize;	size of filesystem allocation information
ULONG FileAttributes;	NT style encoding of file attributes

ULONG FileNameLength;	Length of filename in bytes
ULONG EaSize;	Size of file's extended attributes
STRING FileName;	Name of the file

4.3.4.6 SMB_FIND_FILE_BOTH_DIRECTORY_INFO

Response Field	Description
ULONG NextEntryOffset;	Offset from this structure to beginning of next one
ULONG FileIndex;	
LARGE_INTEGER CreationTime;	file creation time
LARGE_INTEGER LastAccessTime;	last access time
LARGE_INTEGER LastWriteTime;	last write time
LARGE_INTEGER ChangeTime;	last attribute change time
LARGE_INTEGER EndOfFile;	file size
LARGE_INTEGER AllocationSize;	size of filesystem allocation information
ULONG FileAttributes;	NT style encoding of file attributes
ULONG FileNameLength;	Length of FileName in bytes
ULONG EaSize;	Size of file's extended attributes
UCHAR ShortNameLength;	Length of file's short name in bytes
WCHAR ShortName[12];	File's 8.3 conformant name in Unicode
STRING FileName;	Files full length name

4.3.4.7 SMB_FIND_FILE_NAMES_INFO

Response Field	Description
ULONG NextEntryOffset;	Offset from this structure to beginning of next one
ULONG FileIndex;	

ULONG FileNameLength;	Length of FileName in bytes
STRING FileName;	Files full length name

4.3.5 TRANS2_FIND_NEXT2: Resume Directory Search Using Wildcards

This request resumes a search which was begun with a previous `TRANS2_FIND_FIRST2` request.

Client Request ==========	Value ==========
WordCount	15
SetupCount	1
Setup[0]	TRANS2_FIND_NEXT2
Parameter Block Encoding ==========	Description ==========
USHORT Sid;	Search handle
USHORT SearchCount;	Maximum number of entries to return
USHORT InformationLevel;	Levels described in `TRANS2_FIND_FIRST2` request
ULONG ResumeKey;	Value returned by previous find2 call
USHORT Flags;	Additional information: bit set- 0 - close search after this request 1 - close search if end of search reached 2 - return resume keys for each entry found 3 - resume/continue from previous ending place 4 - find with backup intent
STRING FileName;	Resume file name

SID is the value returned by a previous successful `TRANS2_FIND_FIRST2` call. If *BIT3* of *FLAGS* is set, then *FileName* may be the NULL string, since the search is continued from the previous `TRANS2_FIND` request. Otherwise, *FileName* must not be more than 256 characters long.

Response Field	Description
USHORT SearchCount;	Number of entries returned
USHORT EndOfSearch;	Was last entry returned?
USHORT EaErrorOffset;	Offset into EA list if EA error
USHORT LastNameOffset;	Offset into data to file name of last entry, if server needs it to resume search; else 0
UCHAR Data[TotalDataCount]	Level dependent info about the matches found in the search

4.3.6 FIND_CLOSE2: Close Directory Search

This SMB closes a search started by the `TRANS2_FIND_FIRST2` transaction request.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT Sid;	Find handle
USHORT ByteCount;	Count of data bytes = 0

Server Response	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

4.3.7 NT_TRANSACT_NOTIFY_CHANGE: Request Change Notification

Client Setup Words	Description
ULONG CompletionFilter;	Specifies operation to monitor (NT format)

USHORT Fid;	Fid of directory to monitor
BOOLEAN WatchTree;	TRUE = watch all subdirectories too
UCHAR Reserved;	MBZ

This command notifies the client when the directory specified by *FID* is modified. It also returns the name(s) of the file(s) that changed. The command completes once the directory has been modified based on the supplied *CompletionFilter*. The command is a "single shot" and therefore needs to be reissued to watch for more directory changes.

A directory file must be opened before this command may be used. Once the directory is open, this command may be used to begin watching files and subdirectories in the specified directory for changes. The first time the command is issued, the *MaxParameterCount* field in the transact header determines the size of the buffer that will be used at the server to buffer directory change information between issuances of the notify change commands.

When a change that is in the *CompletionFilter* is made to the directory, the command completes. The names of the files that have changed since the last time the command was issued are returned to the client. The *ParameterCount* field of the response indicates the number of bytes that are being returned. If too many files have changed since the last time the command was issued, then zero bytes are returned and an alternate status code is returned in the *Status* field of the response.

The *CompletionFilter* is a mask created as the sum of any of the following flags:

```
FILE_NOTIFY_CHANGE_FILE_NAME                0x00000001

FILE_NOTIFY_CHANGE_DIR_NAME                 0x00000002

FILE_NOTIFY_CHANGE_NAME                     0x00000003

FILE_NOTIFY_CHANGE_ATTRIBUTES               0x00000004

FILE_NOTIFY_CHANGE_SIZE                     0x00000008

FILE_NOTIFY_CHANGE_LAST_WRITE               0x00000010

FILE_NOTIFY_CHANGE_LAST_ACCESS              0x00000020

FILE_NOTIFY_CHANGE_CREATION                 0x00000040

FILE_NOTIFY_CHANGE_EA                       0x00000080

FILE_NOTIFY_CHANGE_SECURITY                 0x00000100

FILE_NOTIFY_CHANGE_STREAM_NAME              0x00000200

FILE_NOTIFY_CHANGE_STREAM_SIZE              0x00000400

FILE_NOTIFY_CHANGE_STREAM_WRITE             0x00000800
```

Server Response	Description

================================	================================
ParameterCount Parameters[ParameterCount]	# of bytes of change data FILE_NOTIFY_INFORMATION structures

The response contains FILE_NOTIFY_INFORMATION structures, as defined below. The NextEntryOffset field of the structure specifies the offset, in bytes, from the start of the current entry to the next entry in the list. If this is the last entry in the list, this field is zero. Each entry in the list must be longword aligned, so NextEntryOffset must be a multiple of four.

```
typedef struct {
    ULONG NextEntryOffset;
    ULONG Action;
    ULONG FileNameLength;
    WCHAR FileName[1];
} FILE_NOTIFY_INFORMATION;
```

Where *Action* describes what happened to the file named *FileName*:

```
FILE_ACTION_ADDED                0x00000001

FILE_ACTION_REMOVED              0x00000002

FILE_ACTION_MODIFIED             0x00000003

FILE_ACTION_RENAMED_OLD_NAME     0x00000004

FILE_ACTION_RENAMED_NEW_NAME     0x00000005

FILE_ACTION_ADDED_STREAM         0x00000006

FILE_ACTION_REMOVED_STREAM       0x00000007

FILE_ACTION_MODIFIED_STREAM      0x00000008
```

4.4 DFS Operations

4.4.1 TRANS2_GET_DFS_REFERRAL: Retrieve Distributed Filesystem Referral

The client sends this request to ask the server to convert *RequestFilename* into an alternate name for this file. This request can be sent to the server if the server response to the `NEGOTIATE SMB` included the `CAP_DFS` capability. The TID of the request must be IPC$. *BIT15* of *Flags2* in the SMB header must be set, indicating this is a UNICODE request.

Client Request ==========	Description ==========

WordCount TotalDataCount SetupCount Setup[0]	15 0 1 TRANS2_GET_DFS_REFERRAL
Parameter Block Encoding ==========	Description ==========
USHORT MaxReferralLevel WCHAR RequestFileName;	Latest referral version number understood DFS name of file for which referral is sought

Response Data Block ==========	Description ==========
USHORT PathConsumed; USHORT NumberOfReferrals; USHORT Flags; REFERRAL_LIST Referrals[]	Number of *REQUESTFILENAME* bytes client Number of referrals contained in this response bit0 - The servers in *REFERRALS* are capable of fielding TRANS2_GET_DFS_REFERRAL. bit1 - The servers in *REFERRALS* should hold the storage for the requested file. Set of referrals for this file
UNICODESTRINGE Strings	Used to hold the strings pointed to by Version 2 Referrals in *REFERRALS*.

The server response is a list of *Referrals* which inform the client where it should resubmit the request to obtain access to the file. *PathConsumed* in the response indicates to the client how many characters of *RequestFilename* have been consumed by the server. When the client chooses one of the referrals to use for file access, the client may need to strip the leading *PathConsumed* characters from the front of *RequestFileName* before submitting the name to the target server. Whether or not the pathname should be trimmed is indicated by the individual referral as detailed below.

FLAGS indicates how this referral should be treated. If *BIT0* is clear, any entity in the *REFERRALS* list holds the storage for *REQUESTFILENAME*. If *BIT0* is set, any entity in the *REFERRALS* list has further referral information for

REQUESTFILENAME – a `TRANS2_GET_DFS_REFERRAL` request should be sent to an entity in the *REFERRALS* list for further resolution.

The format of an individual referral contains version and length information allowing the client to skip referrals it does not understand. MaxReferralLevel indicates to the server the latest version of referral which the client can digest. Since each referral has a uniform element, *MAXREFERRALLEVEL* is advisory only. Each element in *REFERRALS* has this envelope:

REFERRAL_LIST element ==========	
USHORT VersionNumber	Version of this referral element
USHORT ReferralSize	Size of this referral element

The following referral element versions are defined:

Version 1 Referral Element Format ==========	
USHORT ServerType	Type of *NODE* handling referral: 0 - Don't know 1 - SMB Server 2 - Netware Server 3 - Domain
USHORT ReferralFlags	Flags which describe this referral: 01 - Strip off *PATHCONSUMED* characters before submitting *REQUESTFILENAME* to *NODE*
UNICODESTRING Node	Name of entity to visit next

Version 2 Referral Element Format ========== 	
USHORT ServerType	Type of *NODE* handling referral: 0 - Don't know 1 - SMB Server 2 - Netware Server 3 - Domain
USHORT ReferralFlags	Flags which describe this referral: 01 - Strip off *PATHCONSUMED* characters before submitting *REQUESTFILENAME* to *NODE*
ULONG Proximity	A hint describing the proximity of this server to the client. 0 indicates the closest, higher numbers indicate increasingly "distant" servers. The number is only relevant within the context of the servers listed in *THIS* particular SMB.
ULONG TimeToLive	Number of seconds for which the client can cache this referral.
USHORT DfsPathOffset	Offset, in bytes from the beginning of this referral, of the DFS Path that matched *PATHCONSUMED* bytes of the *REQUESTFILENAME.*
USHORT DfsAlternatePathOffset	Offset, in bytes from the beginning of this referral, of an alternate name (8.3 format) of the DFS Path that matched *PATHCONSUMED* bytes of the *REQUESTFILENAME.*
USHORT NetworkAddressOffset	Offset, in bytes from the beginning of this referral, of the entity to visit next.

The SMB protocol imposes no referral selection policy.

4.4.2 TRANS2_REPORT_DFS_INCONSISTENCY: Inform a server about DFS Error

As part of the Distributed Name Resolution algorithm, a DFS client may discover a knowledge inconsistency between the referral server (i.e., the server that handed out a referral), and the storage server (i.e., the server to which the client was redirected to by the referral server). When such an inconsistency is discovered, the DFS client optionally sends this SMB to the referral server, allowing the referral server to take corrective action.

Client Request ==========	Description ==========

WordCount MaxParameterCount SetupCount Setup[0]	15 0 1 TRANS2_REPORT_DFS_INCONSISTENCY
Parameter Block Encoding ==================	Description ==================
UNICODESTRING RequestFileName;	DFS Name of file for which referral was sought

The data part of this request contains the referral element (Version 1 format only) believed to be in error. These are encoded as described in the TRANS2_GET_DFS_REFERRAL response. If the server returns success, the client can resubmit the TRANS2_GET_DFS_REFERRAL request to this server to get a new referral. It is not mandatory for the DFS knowledge to be automatically repaired – the client must be prepared to receive further errant referrals and must not wind up looping between this request and the TRANS2_GET_DFS_REFERRAL request.

BIT15 of *FLAGS2* in the SMB header must be set, indicating this is a UNICODE request.

4.5 Print Spooling Operations

4.5.1 OPEN_PRINT_FILE: Create Print Spool file

This message is sent to create a new printer file which will be deleted once it has been closed and printed.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 2
USHORT SetupLength;	Length of printer setup data
USHORT Mode;	0 = Text mode (DOS expands TABs) 1 = Graphics mode
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING IdentifierString[];	Identifier string

TID in the SMB header must refer to a printer resource type.

SETUPLENGTH is the number of bytes in the first part of the resulting print spool file which contains printer-specific control strings.

MODE can have the following values:

0 Text mode. The server may optionally expand tabs to a series of spaces.

1 Graphics mode. No conversion of data should be done by the server.

IDENTIFIERSTRING can be used by the server to provide some sort of per-client identifying component to the print file.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 1
USHORT Fid;	File handle
USHORT ByteCount;	Count of data bytes = 0

FID is the returned handle which may be used by subsequent write and close operations. When the file is finally closed, it will be sent to the spooler and printed.

4.5.2 GET_PRINT_QUEUE: Get Printer Queue Entries

This message obtains a list of the elements currently in the print queue on the server.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 2
USHORT MaxCount;	Max number of entries to return
USHORT StartIndex;	First queue entry to return
USHORT ByteCount;	Count of data bytes = 0

STARTINDEX specifies the first entry in the queue to return.

MAXCOUNT 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.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 2
USHORT Count;	Number of entries returned
USHORT RestartIndex;	Index of entry after last returned
USHORT ByteCount;	Count of data bytes; min = 3
UCHAR BufferFormat;	0x01 -- Data block
USHORT DataLength;	Length of data
UCHAR Data[];	Queue elements

COUNT indicates how many entries were actually returned. *RESTARTINDEX* is the index of the entry following the last entry returned; it may be used as the *STARTINDEX* in a subsequent request to resume the queue listing.

The format of each returned queue element is:

Queue Element Member	Description
SMB_DATE FileDate;	Date file was queued
SMB_TIME FileTime;	Time file was queued
UCHAR Status;	Entry status. One of:

	01 = held or stopped 02 = printing 03 = awaiting print 04 = in intercept 05 = file had error 06 = printer error 07-FF = reserved
USHORT SpoolFileNumber;	Assigned by the spooler
ULONG SpoolFileSize;	Number of bytes in spool file
UCHAR Reserved;	
UCHAR SpoolFileName[16];	Client which created the spool file

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

Support for this SMB is server optional. In particular, no current Microsoft client software issues this request.

4.6 Miscellaneous Operations

4.6.1 NT_TRANSACT_IOCTL

This command allows device and file system control functions to be transferred transparently from client to server.

Setup Words Encoding ==========	Description ==========
ULONG FunctionCode;	NT device or file system control code
USHORT Fid;	Handle for io or fs control. Unless *BIT0* of *ISFLAGS* is set.
BOOLEAN IsFsctl;	Indicates whether the command is a device control (FALSE) or a file system control (TRUE).
UCHAR IsFlags;	*BIT0* - command is to be applied to share root handle. Share must be a DFS share.

Data Block Encoding ==========	Description ==========
Data[TotalDataCount]	Passed to the Fsctl or Ioctl

Server Response ==========	Description ==========
SetupCount	1
Setup[0]	Length of information returned by io or fs control
DataCount	Length of information returned by io or fs control
Data[DataCount]	The results of the io or fs control

4.6.2 NT_TRANSACT_QUERY_SECURITY_DESC

This command allows the client to retrieve the security descriptor on a file.

Client Parameter Block ==========	Description ==========
USHORT Fid;	FID of target
USHORT Reserved;	MBZ
ULONG SecurityInformation;	Fields of descriptor to set

`NtQuerySecurityObject()` is called, requesting *SECURITYINFORMATION*. The result of the call is returned to the client in the *DATA* part of the transaction response.

4.6.3 NT_TRANSACT_SET_SECURITY_DESC

This command allows the client to change the security descriptor on a file.

Client Parameter Block Encoding ==========	Description ==========
USHORT Fid;	FID of target

USHORT Reserved; ULONG SecurityInformation;	MBZ Fields of SD that to set
Data Block Encoding ==========	Description ==========
Data[TotalDataCount]	Security Descriptor information

DATA is passed directly to `NtSetSecurityObject()`, with *SECURITYINFORMATION* describing which information to set. The transaction response contains no parameters or data.

5. Obsolescent SMB Requests

This section lists the "obsolescent" SMB requests -- ones that are superceded by "best practice" requests, either in function or performance. Clients need not use them to get full function or performance, however, servers do need to support them in order to interoperate with existing clients.

*5.1 CLOSE_PRINT_FILE: Close and Spool Print Job**

This message invalidates the specified file handle and queues the file for printing.

Client Request ==========	Description ==========
UCHAR WordCount; USHORT Fid; USHORT ByteCount;	Count of parameter words = 1 File handle Count of data bytes = 0

FID refers to a file previously created with SMB_COM_OPEN_PRINT_FILE. On successful completion of this request, the file is queued for printing by the server.

Server Response ==========	Description ==========
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

Servers which negotiate dialects of LANMAN1.0 and newer allow all the other types of *FID* closing requests to invalidate the *FID* and begin spooling.

5.2 CREATE: Create File*

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.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 3
USHORT FileAttributes;	New file attributes
UTIME CreationTime;	Time file was created
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING FileName[];	File name

FileName is the fully qualified name of the file relative to *Tid*.

FileAttributes are encoded as described in the "File Attribute Encoding" section.

Server support of the *CreationTime* field is optional. Encoding of these fields is discussed in the "Time And Date Encoding" section.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT Fid;	File handle
USHORT ByteCount;	Count of data bytes = 0

Clients must 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.

5.3 CREATE_DIRECTORY: Create Directory

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.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING DirectoryName[];	Directory name

Servers require clients to have at least *CREATE* permission for the subtree containing the directory in order to create a new directory. The creator's access rights to the new directory are be determined by local policy on the server.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

5.4 CREATE_NEW: Create File*

This message is sent to create a new data file or truncate an existing data file to length zero, and open the file.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 3
USHORT FileAttributes;	New file attributes
UTIME CreationTime;	Creation time for created file
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING FileName[];	File name

FileAttributes specify the attributes of the newly created file, their encoding is described in the "Attribute Encoding" section of this document.

CreationTime is the creation timestamp the file should be given, server support for these is optional.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 1
USHORT Fid;	File handle
USHORT ByteCount;	Count of data bytes = 0

The returned *FID* can be used in subsequent *FID*-related messages.

The access permissions granted on a created file are 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.

*5.5 LOCK_AND_READ: Lock and Read Bytes**

This request is used to lock and "read ahead" the specified bytes of the file indicated by *FID* in the SMB header

Client Request ==============================	Description ==================================
UCHAR WordCount;	Count of parameter words = 5
USHORT Fid;	File handle
USHORT Count;	Count of bytes being requested
ULONG Offset;	Offset in file of first byte to read
USHORT Remaining;	Estimate of bytes to read if nonzero
USHORT ByteCount;	Count of data bytes = 0

FID must refer to a disk file. *COUNT* specifies the requested number of bytes. *OFFSET* specifies the offset in the file of the first byte to be locked then read. Note that this offset is limited to 32 bits, so this client request is inappropriate for files having 64 bit offsets.

REMAINING 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. *REMAINING* is not included in the byte range to be locked.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 5
USHORT Count;	Count of bytes actually returned
USHORT Reserved [4];	Reserved (must be 0)
USHORT ByteCount;	Count of data bytes
UCHAR BufferFormat;	0x01 -- Data block
USHORT DataLength;	Length of data

BYTECOUNT is the number of bytes actually being returned. *BYTECOUNT* 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 results 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.

As in the core SMB_LOCK_BYTE_RANGE request, if the lock can not be immediately granted an error should be returned to the client. If an error occurs on the lock, the bytes should not be read. If a Read requests more data than can be placed in a message of the maximum-xmit-size for the *TID* specified, the server will abort the connection to the client.

5.6 LOCK_BYTE_RANGE: Lock Bytes*

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 prevent attempts to lock, read or write the locked portion of the file by other clients or *PID*s. Overlapping locks are not allowed. Offsets beyond the current end of file may be locked. Such locks will not cause allocation of file space.

Since *OFFSET* is a 32 bit quantity, this request is inappropriate for general locking within a very large file.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 5
USHORT Fid;	File handle
ULONG Count;	Count of bytes to lock
ULONG Offset;	Offset from start of file
USHORT ByteCount;	Count of data bytes = 0

Locks may only be unlocked by the *PID* that performed the lock.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

This client request does not wait for the lock to be granted. If the lock can not be immediately granted (within 200-300 ms), the server should return failure to the client

5.7 OPEN: Open File*

This message is sent to obtain a file handle for a data file. This returned *FID* is used in subsequent client requests such as read, write, close, etc.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 2
USHORT DesiredAccess;	Mode - read/write/share

USHORT SearchAttributes;	
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING FileName[];	File name

FILENAME is the fully qualified file name, relative to the root of the share specified in the *TID* field of the SMB header. If *TID* in the SMB header refers to a print share, this SMB creates a new file which will be spooled to the printer when closed. In this case, *FILENAME* is ignored.

SEARCHATTRIBUTES specifies the type of file desired. The encoding is described in the "File Attribute Encoding" section.

DESIREDACCESS controls the mode under which the file is opened, and the file will be opened only if the client has the appropriate permissions. The encoding of *DESIREDACCESS* is discussed in the section entitled "Access Mode Encoding".

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 7
USHORT Fid;	File handle
USHORT FileAttributes;	Attributes of opened file
UTIME LastWriteTime;	Time file was last written
ULONG DataSize;	File size
USHORT GrantedAccess;	Access allowed
USHORT ByteCount;	Count of data bytes = 0

FID is the handle value which should be used for subsequent file operations.

FILEATTRIBUTES specifies the type of file obtained. The encoding is described in the "File Attribute Encoding" section.

GRANTEDACCESS indicates the access permissions actually allowed, and may have one of the following values:

0 read-only

1 write-only

2 read/write

File Handles (*FID*s) are scoped per client. A *PID* may reference any *FID* established by itself or any other *PID* on the client (so far as the server is concerned). The actual accesses allowed through the *FID* depends on the open and deny modes specified when the file was opened (see below).

The MS-DOS compatibility mode of file open provides exclusion at the client 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* on the same client. If the first client has the file open for writing, then the file may not be opened in any way by any other client. If the first client has the file open only for reading, then other clients may open the file, in compatibility mode, for reading.. The above notwithstanding, if the filename has an extension of .EXE, .DLL, .SYM, or .COM other clients are permitted to open the file regardless of read/write open modes of other compatibility mode opens. However, once multiple clients have the file open for reading, no client is permitted to open the file for writing and no other client may open the file in any mode other than compatibility mode.

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 different client, a *PID* from the same client, 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.

Although *FID*s are available to all *PID*s on a client, *PID*s 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 clients may only read the file and cannot write; if a file is opened for Read in Deny read mode, then the other clients can neither read nor write the file.

5.8 OPEN_ANDX: Open File*

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 15
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Flags;	Additional information: bit set- 0 - return additional info 1 - exclusive oplock requested 2 - batch oplock requested
USHORT DesiredAccess;	File open mode
USHORT SearchAttributes;	
USHORT FileAttributes;	
UTIME CreationTime;	Creation timestamp for file if it gets created
USHORT OpenFunction;	Action to take if file exists
ULONG AllocationSize;	Bytes to reserve on create or truncate
ULONG Reserved[2];	Must be 0
USHORT ByteCount;	Count of data bytes; min = 1
UCHAR BufferFormat	0x04
STRING FileName;	

Server Response	Description
UCHAR WordCount;	Count of parameter words = 15
UCHAR AndXCommand;	Secondary (X) command; 0xFF = none
UCHAR AndXReserved;	Reserved (must be 0)
USHORT AndXOffset;	Offset to next command WordCount
USHORT Fid;	File handle
USHORT FileAttributes;	
UTIME LastWriteTime;	
ULONG DataSize;	Current file size
USHORT GrantedAccess;	Access permissions actually allowed
USHORT FileType;	Type of file opened
USHORT DeviceState;	State of the named pipe
USHORT Action;	Action taken
ULONG ServerFid;	Server unique file id
USHORT Reserved;	Reserved (must be 0)
USHORT ByteCount;	Count of data bytes = 0

DesiredAccess describes the access the client desires for the file; the encoding of this field is described in the "Access Mode Encoding" section elsewhere in this document.

OpenFunction specifies the action to be taken depending on whether or not the file exists (see section 3.7). *Action* in the response specifies the action as a result of the Open request (see section 3.8).

SearchAttributes indicates the attributes that the file must have to be found while searching to see if it exists. The encoding of this field is described in the "File Attribute Encoding" section elsewhere in this document. If *SEARCHATTRIBUTES* 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.

FILETYPE returns the kind of resource actually opened:

Name	Value	Description

FileTypeDisk	0	Disk file or directory as defined in the attribute field
FileTypeByteModePipe	1	Named pipe in byte mode
FileTypeMessageModePipe	2	Named pipe in message mode
FileTypePrinter	3	Spooled printer
FileTypeUnknown	0xFFFF	Unrecognized resource type

If bit0 of *FLAGS* is clear, the *FileAttributes*, *LastWriteTime*, *DataSize*, *FileType*, and *DeviceState* have indeterminate values in the response.

This SMB can request an oplock on the opened file. Oplocks are fully described in the "Oplocks" section elsewhere in this document, and there is also discussion of oplocks in the SMB_COM_LOCKING_ANDX SMB description. *BIT1* and *BIT2* of the *FLAGS* field are used to request oplocks during open.

The following SMBs may follow SMB_COM_OPEN_ANDX:

SMB_COM_READ SMB_COM_READ_ANDX

SMB_COM_IOCTL

5.9 PROCESS_EXIT: Process Exit*

This command informs the server that a client process has terminated. The server must close all files opened by *PID* in the SMB header. This must automatically release all locks the process holds.

Client Request ==================	Description ==================
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

Server Response ==================	Description ==================
UCHAR WordCount; USHORT ByteCount;	Count of parameter words = 0 Count of data bytes = 0

This SMB should not generate any errors from the server, unless the server is a *USER MODE* server and *UID* in the SMB header is invalid.

Clients are not required to send this SMB, they can do all cleanup necessary by sending close SMBs to the server to release resources. In fact, clients who have negotiated `LANMAN 1.0` and later probably do not send this message at all.

5.10 QUERY_INFORMATION: Get File Attributes

This request is sent to obtain information about a file.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING FileName[];	File name

FileName is the fully qualified name of the file relative to the *Tid* in the header.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 10
USHORT FileAttributes;	
UTIME LastWriteTime;	Time of last write
ULONG FileSize;	File size
USHORT Reserved [5];	Reserved - client should ignore
USHORT ByteCount;	Count of data bytes = 0

FileAttributes are as described in the "Attributes Encoding" section of this document.

Note that *FileSize* is limited to 32 bits, this request is inappropriate for files whose size is too large.

5.11 QUERY_INFORMATION2: Get File Information

This SMB is gets information about the file represented by *FID*.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 2
USHORT Fid;	File handle
USHORT ByteCount;	Count of data bytes = 0

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 11
SMB_DATE CreationDate;	
SMB_TIME CreationTime;	
SMB_DATE LastAccessDate;	
SMB_TIME LastAccessTime;	
SMB_DATE LastWriteDate;	
SMB_TIME LastWriteTime;	
ULONG FileDataSize;	File end of data
ULONG FileAllocationSize;	File allocation size
USHORT FileAttributes;	
USHORT ByteCount;	Count of data bytes; min = 0

The file being interrogated is specified by *FID*, which must possess at least read permission.

FileAttributes are described in the "File Attribute Encoding" section elsewhere in this document.

5.12 READ: Read File*

The read message is sent to read bytes of a resource indicated by *FID* in the SMB header.

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 5
USHORT Fid;	File handle
USHORT Count;	Count of bytes being requested
ULONG Offset;	Offset in file of first byte to read
USHORT Remaining;	Estimate of bytes to read if nonzero
USHORT ByteCount;	Count of data bytes = 0

COUNT is used to specify the requested number of bytes.

OFFSET specifies the offset in the file of the first byte to be read. Note that this offset is limited to 32 bits, so this client request is inappropriate for files having 64 bit offsets.

REMAINING 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.

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 5
USHORT Count;	Count of bytes actually returned
USHORT Reserved [4];	Reserved (must be 0)
USHORT ByteCount;	Count of data bytes
UCHAR BufferFormat;	0x01 -- Data block
USHORT DataLength;	Length of data

BYTECOUNT is the number of bytes actually being returned. If *FID* refers to a disk file, *BYTECOUNT* 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 results 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 maximum-xmit-size for the *TID* specified, the server will abort the connection to the client.

5.13 READ_MPX: Read Block Multiplex*

The Read Block Multiplexed protocol is used to maximize the performance of reading a large block of data from the server to the client while still allowing other operations to take place between the client and server in the meantime. The NT server supports SMB_COM_READ_MPX only over connectionless transports.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 8
USHORT Fid;	File handle
ULONG Offset;	Offset in file to begin read
USHORT MaxCount;	Max bytes to return (maximum 65535)
USHORT MinCount;	Min bytes to return (normally 0)
ULONG Reserved1;	
USHORT Reserved2;	
USHORT ByteCount;	Count of data bytes = 0

FID identifies the resource being read, and may refer to a disk file or a spooled printer.

TIMEOUT is the number of milliseconds to wait for completion *FID* refers to a named pipe.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 8
ULONG Offset;	Offset in file where data read
USHORT Count;	Total bytes being returned
USHORT Reserved;	
USHORT DataCompactionMode;	
USHORT Reserved;	
USHORT DataLength;	Number of data bytes this buffer
USHORT DataOffset;	Offset (from header start) to data
USHORT ByteCount;	Count of data bytes

UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Data[];	Data (size = DataLength)

Other requests may be active between the client and server. The server responds with the one or more response messages as defined above until the requested data amount has been returned. Each response contains the *PID* and *MID* of the original request and the *OFFSET* and *COUNT* of describing the placement of the data within the file.

The client knows the maximum amount of data bytes which the server may return (from *MAXCOUNT* of the request). Thus the client initializes its bytes expected variable to this value. The server then informs the client of the actual amount being returned via each part of the response in *COUNT*. The server may reduce the expected bytes by lowering the total number of bytes expected in *COUNT* in any response.

When the amount of data bytes received (sum of the *DATALENGTH* fields) equals the total amount of data bytes expected (smallest *COUNT* received), then the client has received all the data bytes. This allows the protocol to work even if the responses are received out of sequence.

Note that *DATALENGTH* being returned here can not be larger than the smaller of the client's buffer size (as specified in *MAXBUFFERSIZE* on the COM_SESSION_SETUP_AND_X client request SMB) or the server's buffer size (as specified in *MAXBUFFERSIZE* of the COM_NEGOTIATE server response SMB).

As is true in SMB_COM_READ, the total number of bytes returned may be less than the number requested only if a read specifies bytes beyond the current file size and *FID* refers to a disk file. 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 *COUNT*. 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).

Once started, the Read Block Multiplexed operation is expected to go to completion. The client 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 SMB_COM_READ_MPX protocol is:

```
client ---------------> Read MPX. request >----------------------> server
client <---------------< Read MPX response 1 with data <---------- server
client <---------------< Read MPX response 2 with data <---------- server
...
client <---------------< Read MPX response n with data <---------- server
```

5.14 SEARCH: Search Directory using Wildcards*

This command is used to search directories.

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 2
USHORT MaxCount;	Number of dir. entries to return
USHORT SearchAttributes;	
USHORT ByteCount;	Count of data bytes; min = 5
UCHAR BufferFormat1;	0x04 -- ASCII
UCHAR FileName[];	File name, may be null
UCHAR BufferFormat2;	0x05 -- Variable block
USHORT ResumeKeyLength;	Length of resume key, may be 0
UCHAR ResumeKey[];	Resume key

FILENAME specifies the file to be sought. *SEARCHATTRIBUTES* indicates the attributes that the file must have, and is described in the "File Attribute Encoding" section of this document. If *SEARCHATTRIBUTES* 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.

MAXCOUNT specifies the number of directory entries to be returned.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT Count;	Number of entries returned
USHORT ByteCount;	Count of data bytes; min = 3
UCHAR BufferFormat;	0x05 -- Variable block
USHORT DataLength;	Length of data
UCHAR DirectoryInformationData[];	Data

The response will contain one or more directory entries as determined by the *COUNT* field. No more than *MAXCOUNT* entries will be returned. Only entries that match the sought *FILENAME* and *SEARCHATTRIBUTES* combination will be returned.

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

Resume Key Field	Description
UCHAR Reserved;	bit 7 - consumer use bits 5,6 - system use (must preserve) bits 0-4 - server use (must preserve)
UCHAR FileName[11];	Name of the returned file
UCHAR ReservedForServer[5];	Client must not modify
UCHAR ReservedForConsumer[4];	Server must not modify

FILENAME is 8.3 format, with the three character extension left justified into *FILENAME*[9-11]. If the client is prior to the LANMAN1.0 dialect, the returned *FILENAME* should be uppercased.

SMB_COM_SEARCH terminates 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 SMB_COM_SEARCH supports wildcards in the last component of *FILENAME* of the initial request.

Returned directory entries in the *DIRECTORYINFORMATIONDATA* field of the response each have the following format:

Directory Information Field	Description
SMB_RESUME_KEY ResumeKey;	Described above
UCHAR FileAttributes;	Attributes of the found file
SMB_TIME LastWriteTime;	Time file was last written
SMB_DATE LastWriteDate;	Date file was last written
ULONG FileSize;	Size of the file
UCHAR FileName[13];	ASCII, space-filled null terminated

FILENAME must conform to 8.3 rules, and is padded after the extension with 0x20 characters if necessary. If the client has negotiated a dialect prior to the LANMAN1.0 dialect, or if *BIT0* of the *FLAGS2* SMB header field of the request is clear, the returned *FILENAME* should be uppercased.

As can be seen from the above structure, SMB_COM_SEARCH can not return long filenames, and can not return UNICODE filenames. Files which have a size greater than 2^32 bytes should have the least significant 32 bits of their size returned in *FILESIZE*.

5.15 SET_INFORMATION: Set File Attributes

This message is sent to change the information about a file.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 8
USHORT FileAttributes;	Attributes of the file
UTIME LastWriteTime;	Time of last write
USHORT Reserved [5];	Reserved (must be 0)
USHORT ByteCount;	Count of data bytes; min = 2
UCHAR BufferFormat;	0x04
STRING FileName[];	File name

FileName is the fully qualified name of the file relative to the *TID*.

Support of all parameters is optional. A server which does not implement one of the parameters will ignore that field. If the *LastWriteTime* field contain zero then the file's time is not changed.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

5.16 SET_INFORMATION2: Set File Information

Client Request	Description
UCHAR WordCount;	Count of parameter words = 7
USHORT Fid;	File handle
SMB_DATE CreationDate;	
SMB_TIME CreationTime;	
SMB_DATE LastAccessDate;	
SMB_TIME LastAccessTime;	
SMB_DATE LastWriteDate;	
SMB_TIME LastWriteTime;	
USHORT ByteCount;	Count of data bytes = 0

SMB_COM_SET_INFORMATION2 sets information about the file represented by *Fid*. The target file is updated from the values specified. A date or time value or zero indicates to leave that specific date and time unchanged.

Server Response	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

FID must be open with (at least) write permission.

5.17 QUERY_INFORMATION_DISK: Get Disk Attributes

The SMB_COM_QUERY_INFORMATION_DISK command is used to determine the capacity and remaining free space on the drive hosting the directory structure indicated by *Tid* in the SMB header.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

Server Response	Description
UCHAR WordCount;	Count of parameter words = 5
USHORT TotalUnits;	Total allocation units per server
USHORT BlocksPerUnit;	Blocks per allocation unit
USHORT BlockSize;	Block size (in bytes)
USHORT FreeUnits;	Number of free units
USHORT Reserved;	Reserved (client should ignore)
USHORT ByteCount;	Count of data bytes = 0

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.

This SMB only returns 16 bits of information for each field, which may not be large enough for some disk systems. In particular *TotalUnits* is commonly > 64K. Fortunately, it turns out the all the client cares about is the total disk size, in bytes, and the free space, in bytes. So, it is reasonable for a server to adjust the relative values of *BlocksPerUnit* and *BlockSize* to accommodate. If after all adjustment, the numbers are still too high, the largest possible values for TotalUnit or FreeUnits (i.e. 0xFFFF) should be returned.

5.18 TRANS2_OPEN2: Create or Open File with Extended Attributes

This transaction is used to open or create a file having extended attributes.

Client Request ==========	Value ==========
WordCount	15
TotalDataCount	Total size of extended attribute list
DataOffset	Offset to extended attribute list in this request
SetupCount	1
Setup[0]	TRANS2_OPEN2
Parameter Block Encoding ==========	Description ==========
USHORT Flags;	Additional information: bit set- 0 - return additional info 1 - exclusive oplock requested 2 - batch oplock requested 3 - return total length of EAs
USHORT DesiredAccess;	Requested file access
USHORT Reserved1;	Ought to be zero. Ignored by the server.
USHORT FileAttributes;	Attributes for file if create
SMB_TIME CreationTime;	Creation time to apply to file if create
SMB_DATE CreationDate;	Creation date to apply to file if create
USHORT OpenFunction;	Open function
ULONG AllocationSize;	Bytes to reserve on create or truncate
USHORT Reserved [5];	Must be zero
STRING FileName;	Name of file to open or create
UCHAR Data[TotalDataCount]	FEAList structure for file to be created

If secondary requests are required, they must contain 0 parameter bytes, and the *FID* in the secondary request is 0xFFFF.

DESIREDACCESS is encoded as described in the "Access Mode Encoding" section elsewhere in this document.

FILEATTRIBUTES are encoded as described in the "File Attribute Encoding" section elsewhere in this document.

OPENFUNCTION specifies the action to be taken depending on whether or not the file exists (see section 3.7) .

ACTION in the response specifies the action as a result of this request (see section 3.8).

Response Parameter Block ==========	Description ==========
USHORT Fid;	File handle
USHORT FileAttributes;	Attributes of file
SMB_TIME CreationTime;	Last modification time
SMB_DATE CreationDate;	Last modification date
ULONG DataSize;	Current file size
USHORT GrantedAccess;	Access permissions actually allowed
USHORT FileType;	Type of file
USHORT DeviceState;	State of IPC device (e.g. pipe)
USHORT Action;	Action taken
ULONG Reserved;	
USHORT EaErrorOffset;	Offset into EA list if EA error
ULONG EaLength;	Total EA length for opened file

FILETYPE returns the kind of resource actually opened:

Name	Value	Description
FileTypeDisk	0	Disk file or directory as defined in the attribute field
FileTypeByteModePipe	1	Named pipe in byte mode
FileTypeMessageModePipe	2	Named pipe in message mode
FileTypePrinter	3	Spooled printer
FileTypeUnknown	0xFFFF	Unrecognized resource type

DeviceState is applicable only if the *FileType* is FileTypeByteModePipe or FileTypeMessageModePipe and is encoded as in section 3.9.

If an error was detected in the incoming EA list, the offset of the error is returned in *EaErrorOffset*.

If *BIT0* of *FLAGS* in the request is clear, the *FileAttributes*, *CREATIONTIME*, *CREATIONDATE*, *DATASIZE*, *GRANTEDACCESS*, *FILETYPE*, and *DEVICESTATE* have indeterminate values in the response. Similarly, if *BIT3* of the request is clear, *EALENGTH* in the response has an indeterminate value in the response.

This SMB can request an oplock on the opened file. Oplocks are fully described in the "Oplocks" section elsewhere in this document, and there is also discussion of oplocks in the SMB_COM_LOCKING_ANDX SMB description. *BIT1* and *BIT2* of the *FLAGS* field are used to request oplocks during open.

5.19 TREE_CONNECT: Tree Connect

When a client connects to a server resource, an SMB_COM_TREE_CONNECT message is generated to the server.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes; min = 4
UCHAR BufferFormat1;	0x04
STRING Path[];	Server name and share name
UCHAR BufferFormat2;	0x04
STRING Password[];	Password
UCHAR BufferFormat3;	0x04

STRING Service[];	Service name

The serving machine verifies the combination and returns an error code or an identifier. The full name is included in this request message and the identifier identifying the connection is returned in the *TID* field of the SMB header. The *TID* field in the client request is ignored. The meaning of this identifier (*TID*) is server specific; the client must not associate any specific meaning to it.

If the negotiated dialect is prior to LANMAN1.0 and the client has not sent a successful SMB_COM_SESSION_SETUP_ANDX request when the tree connect arrives, a user level server must nevertheless validate the client's credentials as discussed earlier in this document. If the negotiated dialect is LANMAN1.0 and later, then it is a protocol violation for the client to send this message prior to a successful SMB_COM_SESSION_SETUP_ANDX. Having received an SMB_COM_SESSION_SETUP_AND_X, the server ignores *PASSWORD*.

PATH follows UNC style syntax, that is to say it is encoded as \\server\share and it indicates the name of the resource the client wishes to connect to.

If the server is paused, administrative privilege is required to connect to any share; if the server is not paused, administrative privilege is required only for administrative shares (C$, etc.). Of course, the server can enforce whatever policy it desires to govern share access. Such policies may include valid times of day, software usage license limits, number of simultaneous server users or share users, etc.

The Service component indicates the type of resource the client intends to access. Valid values are:

Service ========	Description ==========	Earliest Dialect Allowed ==========
A:	disk share	PC NETWORK PROGRAM 1.0
LPT1:	printer	PC NETWORK PROGRAM 1.0
IPC	named pipe	MICROSOFT NETWORKS 3.0
COMM	communications device	MICROSOFT NETWORKS 3.0
?????	any type of device	MICROSOFT NETWORKS 3.0

The SMB server responds with:

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 2
USHORT MaxBufferSize;	Max size message the server handles
USHORT Tid;	Tree ID
USHORT ByteCount;	Count of data bytes = 0

If the negotiated dialect is `MICROSOFT NETWORKS 1.03` or earlier, MaxBufferSize in the response message indicates the maximum size message that the server can handle. The client should not generate messages, nor expect to receive responses, larger than this. This must be constant for a given server. For newer dialects, this field is ignored.

TID should be included in any future SMBs referencing this tree connection.

5.20 UNLOCK_BYTE_RANGE: Unlock Bytes*

This message is sent to unlock the given byte range. *OFFSET*, *COUNT*, and *PID* must be identical to that specified in a prior successful lock. If an unlock references an address range that is not locked, no error is generated.

Since *OFFSET* is a 32 bit quantity, this request is inappropriate for general locking within a very large file.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 5
USHORT Fid;	File handle
ULONG Count;	Count of bytes to unlock
ULONG Offset;	Offset from start of file
USHORT ByteCount;	Count of data bytes = 0

Server Response	Description
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

5.21 WRITE: Write Bytes*

The write message is sent to write bytes into the resource indicated by *FID* in the SMB header.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 5
USHORT Fid;	File handle
USHORT Count;	Number of bytes to be written
ULONG Offset;	Offset in file to begin write
USHORT Remaining;	Bytes remaining to satisfy request
USHORT ByteCount;	Count of data bytes
UCHAR BufferFormat;	0x01 -- Data block

USHORT DataLength;	Length of data
UCHAR Data[Count];	The data to write

COUNT specifies the number of bytes to be written. *OFFSET* is the offset in the file of the first byte to be written. Since offset is 32 bits, this request is inappropriate for general use in a very large file. *REMAINING* 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 cache behavior.

When *FID* represents a disk file and the request 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 are initialized to 0. When a write specifies a length of zero, the file is truncated (or extended) to the length specified by the offset.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 1
USHORT Count;	Count of bytes actually written
USHORT ByteCount;	Count of data bytes = 0

COUNT in the response 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 resources available with which to satisfy the complete write.

If a Write sends a message of length greater than the *MAXBUFFERSIZE* for the TID specified, the server may abort the connection to the client.

*5.22 WRITE_AND_UNLOCK: Write Bytes and Unlock Range**

This 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 client 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. Whether or not this SMB is supported (along with `SMB_COM_READ_AND_LOCK`) is returned in *BIT0* of the *FLAGS* field of the negotiate response.

Client Request ================================	Description ==================================
UCHAR WordCount;	Count of parameter words = 5
USHORT Fid;	File handle
USHORT Count;	Number of bytes to be written
ULONG Offset;	Offset in file to begin write
USHORT Remaining;	Bytes remaining to satisfy request
USHORT ByteCount;	Count of data bytes
UCHAR BufferFormat;	0x01 -- Data block
USHORT DataLength;	Length of data

COUNT specifies the number of bytes to be written. *OFFSET* is the offset in the file of the first byte to be written. Since offset is 16 bits, this request is inappropriate for general use in a very large file. *REMAINING* 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 cache behavior. A value of 0 for *COUNT* is an error.

If the request 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 are initialized to 0.

Server Response ================================	Description ==================================
UCHAR WordCount;	Count of parameter words = 1
USHORT Count;	Count of bytes actually written
USHORT ByteCount;	Count of data bytes = 0

COUNT in the response 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 resources available with which to satisfy the complete write.

If a Write sends a message of length greater than the *MAXBUFFERSIZE* for the TID specified, the server may abort the connection to the client. If an error occurs on the write, the bytes remain locked.

*5.23 WRITE_AND_CLOSE: Write Bytes and Close File**

This request is used to first write the specified bytes and then close the file.

Client Request	Description
UCHAR WordCount;	Count of parameter words = 6
USHORT Fid;	File handle
USHORT Count;	Number of bytes to write
ULONG Offset;	Offset in file of first byte to write
UTIME LastWriteTime;	Time of last write
USHORT ByteCount;	1 (for pad) + value of Count
UCHAR Pad;	To force to doubleword boundary
UCHAR Buffer[Count];	Data to write

Client Request	Description
UCHAR WordCount;	Count of parameter words = 12
USHORT Fid;	File handle
USHORT Count;	Number of bytes to write
ULONG Offset;	Offset in file of first byte to write
UTIME LastWriteTime;	Time of last write
ULONG Reserved[3];	Reserved, must be 0
USHORT ByteCount;	1 (for pad) + value of Count
UCHAR Pad;	To force to doubleword boundary
UCHAR Buffer[Count];	Data to write

Server Response	Description
UCHAR WordCount;	Count of parameter words = 1
USHORT Count;	Count of bytes actually written

USHORT ByteCount;	Count of data bytes = 0

Since clients can formulate the request in either of two ways, *WORDCOUNT* must be used in order to correctly locate the data to be written.

COUNT specifies the number of bytes to be written. *OFFSET* is the offset in the file of the first byte to be written. Since *OFFSET* is 32 bits, this request is inappropriate for general use in a very large file.

If *LASTWRITETIME* and *LASTWRITEDATE* are 0, the server should allow its local operating system to set the file's times. Otherwise, the server should set the time to the values requested. Failure to set the times, even if requested by the client in this message, should not result in an error response from the server.

If *COUNT* is 0, the file is truncated (or extended) to *OFFSET*.

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

5.24 WRITE_MPX: Write Block Multiplex*

Client Request ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 12
USHORT Fid;	File handle
USHORT Count;	Total bytes, including this buffer
USHORT Reserved;	
ULONG Offset;	Offset in file to begin write
ULONG Timeout;	milliseconds to wait for completion
USHORT WriteMode;	Write mode: bit 0 - complete write to disk and send final result response bit 1 - return Remaining bit 7 - Connectionless mode
ULONG RequestMask;	Connectionless mode mask
USHORT DataLength;	Number of data bytes this buffer
USHORT DataOffset;	Offset (from header start) to data

USHORT ByteCount;	Count of data bytes
UCHAR Pad[];	Pad to SHORT or LONG
UCHAR Data[];	Data (# = DataLength)

Server Response ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 1
ULONG ResponseMask;	OR of all masks received
USHORT ByteCount;	Count of data bytes = 0

SMB_COM_WRITE_MPX is used to maximize the performance of writing a large block of data from the client to the server. The NT server supports SMB_COM_WRITE_MPX only over connectionless transports, consequently *BIT7* of *WRITEMODE* in the request must be set.

FID in the request must refer to either a file or a spooled printer.

MASK contains a bit mask indicating where in the transfer that the SMB belongs. The response which contains the logical OR of all of the *MASK* values received and is always generated. All in this exchange use the same SMB header *MID* value but only final message is a connectionless sequenced request (*SEQUENCENUMBER* is non-zero).

The server keeps a *RESPONSEMASK* which is the logical or-ing of the *REQUESTMASK* value contained in each SMB_COM_WRITE_MPX received since the last sequenced SMB_COM_WRITE_MPX. The server only responds to the final (sequenced) command, and this response contains the accumulated *RESPONSEMASK*. The client uses the *RESPONSEMASK* received to determine which packets, if any, must be retransmitted. The server imposes no restrictions on the values in the mask nor upon the order or contiguity of the data being sent. The client uses this behavior to only send the missing parts in the next write sequence when retransmitting. The next SMB_COM_WRITE_MPX sequence sent must use a new *SEQUENCENUMBER* value or the server will incorrectly respond with the mask from the previous SMB_COM_WRITE_MPX command.

The flow is:

Client ==========	Sequence Number =========	<-> ====	Server ==========
SMB_COM_WRITE_MPX	0	->	
SMB_COM_WRITE_MPX	0	->	
...			
SMB_COM_WRITE_MPX	S	->	
	S	<-	SMB_COM_WRITE_MPX OK
SMB_COM_WRITE_MPX	0	->	
SMB_COM_WRITE_MPX	0	->	
....			
SMB_COM_WRITE_MPX	S+1	->	
	S+1	<-	SMB_COM_WRITE_MPX OK

Other SMB requests can intervene during this protocol exchange.

A server response will be generated only after the sequenced SMB_COM_WRITE_MPX has been received unless this SMB is received over a connection oriented transport (in which case the error response is immediately sent).

At the time of the request, the client knows the number of data bytes expected to be sent and passes this information to the server in *COUNT*. The server can use this information to reserve buffer space, if possible.

If *BIT0* of *WRITEMODE* is clear, the request assumed to be a form of write behind on the part of the client. 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 *BIT0* of *WRITEMODE* is set, the server will collect all the data, write it to disk and then send a final response indicating the result of the write . The total number of bytes written is also returned in this response.

5.25 WRITE_PRINT_FILE: Write to Print File*

This message is sent to write bytes into a print spool file.

Client Request ==========	Description ==========
UCHAR WordCount;	Count of parameter words = 1
USHORT Fid;	File handle
USHORT ByteCount;	Count of data bytes; min = 4

UCHAR BufferFormat;	0x01 -- Data block
USHORT DataLength;	Length of data
UCHAR Data[];	Data

FID indicates the print spool file to be written, it must refer to a print spool file.

BYTECOUNT specifies the number of bytes to be written, and must be less than *MAXBUFFERSIZE* for the Tid specified.

DATA contains the bytes to append to the print spool file. The first *SETUPLENGTH* bytes in the resulting print spool file contain printer setup data. *SETUPLENGTH* is specified in the SMB_COM_OPEN_PRINT_FILE SMB request.

Server Response ================================	Description ================================
UCHAR WordCount;	Count of parameter words = 0
USHORT ByteCount;	Count of data bytes = 0

Servers which negotiate a protocol dialect of `LANMAN1.0` or later also support the application of normal write requests to print spool files.

6. SMB Symbolic Constants

6.1 SMB Command Codes

The following values have been assigned for the SMB Commands.

SMB_COM_CREATE_DIRECTORY	0x00
SMB_COM_DELETE_DIRECTORY	0x01
SMB_COM_OPEN	0x02
SMB_COM_CREATE	0x03
SMB_COM_CLOSE	0x04
SMB_COM_FLUSH	0x05
SMB_COM_DELETE	0x06
SMB_COM_RENAME	0x07
SMB_COM_QUERY_INFORMATION	0x08
SMB_COM_SET_INFORMATION	0x09
SMB_COM_READ	0x0A
SMB_COM_WRITE	0x0B
SMB_COM_LOCK_BYTE_RANGE	0x0C
SMB_COM_UNLOCK_BYTE_RANGE	0x0D
SMB_COM_CREATE_TEMPORARY	0x0E
SMB_COM_CREATE_NEW	0x0F
SMB_COM_CHECK_DIRECTORY	0x10
SMB_COM_PROCESS_EXIT	0x11
SMB_COM_SEEK	0x12
SMB_COM_LOCK_AND_READ	0x13
SMB_COM_WRITE_AND_UNLOCK	0x14
SMB_COM_READ_RAW	0x1A
SMB_COM_READ_MPX	0x1B
SMB_COM_READ_MPX_SECONDARY	0x1C

SMB_COM_WRITE_RAW	0x1D
SMB_COM_WRITE_MPX	0x1E
SMB_COM_WRITE_COMPLETE	0x20
SMB_COM_SET_INFORMATION2	0x22
SMB_COM_QUERY_INFORMATION2	0x23
SMB_COM_LOCKING_ANDX	0x24
SMB_COM_TRANSACTION	0x25
SMB_COM_TRANSACTION_SECONDARY	0x26
SMB_COM_IOCTL	0x27
SMB_COM_IOCTL_SECONDARY	0x28
SMB_COM_COPY	0x29
SMB_COM_MOVE	0x2A
SMB_COM_ECHO	0x2B
SMB_COM_WRITE_AND_CLOSE	0x2C
SMB_COM_OPEN_ANDX	0x2D
SMB_COM_READ_ANDX	0x2E
SMB_COM_WRITE_ANDX	0x2F
SMB_COM_CLOSE_AND_TREE_DISC	0x31
SMB_COM_TRANSACTION2	0x32
SMB_COM_TRANSACTION2_SECONDARY	0x33
SMB_COM_FIND_CLOSE2	0x34
SMB_COM_FIND_NOTIFY_CLOSE	0x35
SMB_COM_TREE_CONNECT	0x70
SMB_COM_TREE_DISCONNECT	0x71
SMB_COM_NEGOTIATE	0x72
SMB_COM_SESSION_SETUP_ANDX	0x73
SMB_COM_LOGOFF_ANDX	0x74
SMB_COM_TREE_CONNECT_ANDX	0x75

SMB_COM_QUERY_INFORMATION_DISK	0x80
SMB_COM_SEARCH	0x81
SMB_COM_FIND	0x82
SMB_COM_FIND_UNIQUE	0x83
SMB_COM_NT_TRANSACT	0xA0
SMB_COM_NT_TRANSACT_SECONDARY	0xA1
SMB_COM_NT_CREATE_ANDX	0xA2
SMB_COM_NT_CANCEL	0xA4
SMB_COM_OPEN_PRINT_FILE	0xC0
SMB_COM_WRITE_PRINT_FILE	0xC1
SMB_COM_CLOSE_PRINT_FILE	0xC2
SMB_COM_GET_PRINT_QUEUE	0xC3

6.2 Named Pipe Transaction Protocol Subcommand Codes

The subcommand codes, placed in *SETUP[0]*, for named pipe operations are:

SubCommand Code ==================	Value =====	Description ====================================
CallNamedPipe	0x54	open/write/read/close pipe
WaitNamedPipe	0x53	wait for pipe to be nonbusy
PeekNmPipe	0x23	read but don't remove data
QNmPHandState	0x21	query pipe handle modes
SetNmPHandState	0x01	set pipe handle modes
QNmPipeInfo	0x22	query pipe attributes
TransactNmPipe	0x26	write/read operation on pipe
RawReadNmPipe	0x11	read pipe in "raw" (non message mode)
RawWriteNmPipe	0x31	write pipe "raw" (non message mode) */

6.3 SMB_COM_TRANSACTION2 Subcommand codes

The subcommand code for `SMB_COM_TRANSACTION2` request is placed in Setup[0]. The parameters associated with any particular request are placed in the *PARAMETERS* vector of the request. The defined subcommand codes are:

Setup[0] Transaction2 Subcommand Code ==========	Value =====	Description ==========
TRANS2_OPEN2	0x00	Create file with extended attributes
TRANS2_FIND_FIRST2	0x01	Begin search for files
TRANS2_FIND_NEXT2	0x02	Resume search for files
TRANS2_QUERY_FS_INFORMATION	0x03	Get file system information
	0x04	Reserved
TRANS2_QUERY_PATH_INFORMATION	0x05	Get information about a named file or directory
TRANS2_SET_PATH_INFORMATION	0x06	Set information about a named file or directory
TRANS2_QUERY_FILE_INFORMATION	0x07	Get information about a handle
TRANS2_SET_FILE_INFORMATION	0x08	Set information by handle
TRANS2_FSCTL	0x09	Not implemented by NT server
TRANS2_IOCTL2	0x0A	Not implemented by NT server
TRANS2_FIND_NOTIFY_FIRST	0x0B	Not implemented by NT server
TRANS2_FIND_NOTIFY_NEXT	0x0C	Not implemented by NT server
TRANS2_CREATE_DIRECTORY	0x0D	Create directory with extended attributes
TRANS2_SESSION_SETUP	0x0E	Session setup with extended security information
TRANS2_GET_DFS_REFERRAL	0x10	Get a DFS referral
TRANS2_REPORT_DFS_INCONSISTENCY	0x11	Report a DFS knowledge inconsistency

6.4 SMB_COM_NT_TRANSACTION Subcommand Codes

For these transactions, *FUNCTION* in the primary client request indicates the operation to be performed. It may assume one of the following values:

SubCommand Code ==========	Value =====	Description ==========
NT_TRANSACT_CREATE	1	File open/create
NT_TRANSACT_IOCTL	2	Device IOCTL

NT_TRANSACT_SET_SECURITY_DESC	3	Set security descriptor
NT_TRANSACT_NOTIFY_CHANGE	4	Start directory watch
NT_TRANSACT_RENAME	5	Reserved (Handle-based rename)
NT_TRANSACT_QUERY_SECURITY_DESC	6	Retrieve security descriptor info

6.5 SMB Protocol Dialect Constants

This is the list of SMB protocol dialects, ordered from least functional (earliest) version to most functional (most recent) version:

Dialect Name ==========	Comment ==========
PC NETWORK PROGRAM 1.0	The original MSNET SMB protocol (otherwise known as the "core protocol")
PCLAN1.0	Some versions of the original MSNET defined this as an alternate to the core protocol name
MICROSOFT NETWORKS 1.03	This is used for the MS-NET 1.03 product. It defines Lock&Read,Write&Unlock, and a special version of raw read and raw write.
MICROSOFT NETWORKS 3.0	This is the DOS LANMAN 1.0 specific protocol. It is equivalent to the LANMAN 1.0 protocol, except the server is required to map errors from the OS/2 error to an appropriate DOS error.
LANMAN1.0	This is the first version of the full LANMAN 1.0 protocol
LM1.2X002	This is the first version of the full LANMAN 2.0 protocol
DOS LM1.2X002	This is the dos equivalent of the LM1.2X002 protocol. It is identical to the LM1.2X002 protocol, but the server will perform error mapping to appropriate DOS errors.
DOS LANMAN2.1	DOS LANMAN2.1
LANMAN2.1	OS/2 LANMAN2.1

`Windows for Workgroups 3.1a`	Windows for Workgroups Version 1.0
`NT LM 0.12`	The SMB protocol designed for NT networking. This has special SMBs which duplicate the NT semantics.

SMB servers select the most recent version of the protocol known to both client and server. Any SMB server which supports dialects newer than the original core dialect must support all the messages and semantics of the dialects between the core dialect and the newer one. This is to say that a server which supports the `NT LM 0.12` dialect must also support all of the messages of the previous 10 dialects. It is the client's responsibility to ensure it only sends SMBs which are appropriate to the dialect negotiated. Clients must be prepared to receive an SMB response from an earlier protocol dialect -- even if the client used the most recent form of the request.

7. Error Codes and Classes

This section lists all of the valid values for *STATUS.DOSERROR.ERRORCLASS*, and most of the error codes for *STATUS.DOSERROR.ERROR*.

The following error classes may be returned by the server to the client.

Class =======	Code ====	Comment ==========
SUCCESS	0	The request was successful.
ERRDOS	0x01	Error is from the core DOS operating system set.
ERRSRV	0x02	Error is generated by the server network file manager.
ERRHRD	0x03	Error is an hardware error.
ERRCMD	0xFF	Command was not in the "SMB" format.

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

Class =======	Code ====	Comment ==========
SUCCESS	0	The request was successful.

The following error codes may be generated with the ERRDOS error class. When an SMB dialect greater than equal to `LANMAN 1.0` has been negotiated, all of the error codes below may be generated plus any of the error codes defined for OS/2 (see OS/2 operating system documentation for complete list of OS/2 error codes). When an earlier dialect has been negotiated, the server must map additional OS/2 (or OS/2 like) errors to the errors listed below.

Error ==========	Code =====	Description ==========
ERRbadfunc	1	Invalid function. The server 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.
ERRbadfile	2	File not found. The last component of a file's pathname could not be found.
ERRbadpath	3	Directory invalid. A directory component in a pathname could not be found.
ERRnofids	4	Too many open files. The server has no file handles available.
ERRnoaccess	5	Access denied, the client's context does not permit the requested function. This includes the following conditions: o invalid rename command o write to fid open for read only o read on fid open for write only o attempt to delete a non-empty directory
ERRbadfid	6	Invalid file handle. The file handle specified was not recognized by the server.
ERRbadmcb	7	Memory control blocks destroyed.
ERRnomem	8	Insufficient server memory to perform the requested function.
ERRbadmem	9	Invalid memory block address.
ERRbadenv	10	Invalid environment.
ERRbadformat	11	Invalid format.
ERRbadaccess	12	Invalid open mode.
ERRbaddata	13	Invalid data (generated only by IOCTL calls within the server).
ERRbaddrive	15	Invalid drive specified.
ERRremcd	16	A Delete Directory request attempted to remove the server's current directory.
ERRdiffdevice	17	Not same device (e.g., a cross volume rename was attempted)
ERRnofiles	18	A File Search command can find no more files matching the specified criteria.
ERRbadshare	32	The sharing mode specified for an Open conflicts with existing FIDs on the file.
ERRlock	33	A Lock request conflicted with an existing lock or specified an invalid mode, or

		an Unlock requested attempted to remove a lock held by another process.
ERRfilexists	80	The file named in a Create Directory, Make New File or Link request already exists. The error may also be generated in the Create and Rename transaction.
ERRbadpipe	230	Pipe invalid.
ERRpipebusy	231	All instances of the requested pipe are busy.
ERRpipeclosing	232	Pipe close in progress.
ERRnotconnected	233	No process on other end of pipe.
ERRmoredata	234	There is more data to be returned.

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

Error	Code	Description
ERRerror	1	Non-specific error code. It is returned under the following conditions: o resource other than disk space exhausted (e.g. TIDs) o first SMB command was not negotiate o multiple negotiates attempted o internal server error
ERRbadpw	2	Bad password - name/password pair in a Tree Connect or Session Setup are invalid.
ERRaccess	4	The client does not have the necessary access rights within the specified context for the requested function.
ERRinvnid	5	The Tid specified in a command was invalid.
ERRinvnetname	6	Invalid network name in tree connect.
ERRinvdevice	7	Invalid device - printer request made to non-printer connection or non-printer request made to printer connection.
ERRqfull	49	Print queue full (files) -- returned by open print file.
ERRqtoobig	50	Print queue full -- no space.
ERRqeof	51	EOF on print queue dump.
ERRinvpfid	52	Invalid print file FID.
ERRsmbcmd	64	The server did not recognize the command received.
ERRsrverror	65	The server encountered an internal error, e.g., system file unavailable.
ERRfilespecs	67	The Fid and pathname parameters contained an invalid combination of values.
ERRbadpermits	69	The access permissions specified for a file or directory are not a valid combination. The server cannot set the requested attribute.
ERRsetattrmode	71	The attribute mode in the Set File Attribute request is invalid.
ERRpaused	81	Server is paused. (reserved for messaging)
ERRmsgoff	82	Not receiving messages. (reserved for messaging).

ERRnoroom	83	No room to buffer message. (reserved for messaging).
ERRrmuns	87	Too many remote user names. (reserved for messaging).
ERRtimeout	88	Operation timed out.
ERRnoresource	89	No resources currently available for request.
ERRtoomanyuids	90	Too many Uids active on this session.
ERRbaduid	91	The Uid is not known as a valid user identifier on this session.
ERRusempx	250	Temporarily unable to support Raw, use MPX mode.
ERRusestd	251	Temporarily unable to support Raw, use standard read/write.
ERRcontmpx	252	Continue in MPX mode.
ERRnosupport	65535	Function not supported.

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

Error	Code	Description
ERRnowrite	19	Attempt to write on write-protected media
ERRbadunit	20	Unknown unit.
ERRnotready	21	Drive not ready.
ERRbadcmd	22	Unknown command.
ERRdata	23	Data error (CRC).
ERRbadreq	24	Bad request structure length.
ERRseek	25	Seek error.
ERRbadmedia	26	Unknown media type.
ERRbadsector	27	Sector not found.
ERRnopaper	28	Printer out of paper.
ERRwrite	29	Write fault.
ERRread	30	Read fault.
ERRgeneral	31	General failure.
ERRbadshare	32	A open conflicts with an existing open.
ERRlock	33	A Lock request conflicted with an existing lock or specified an invalid mode, or an Unlock requested attempted to remove a lock held by another process.
ERRwrongdisk	34	The wrong disk was found in a drive.
ERRFCBUnavail	35	No FCBs are available to process request.
ERRsharebufexc	36	A sharing buffer has been exceeded.

8. Legal Notice

Microsoft does not know of any third-party rights that are violated by this contribution. Microsoft makes no other representations regarding this contribution.

9. References

[1] P. Mockapetris, "Domain Names - Concepts And Facilities", RFC 1034, November 1987

[2] P. Mockapetris, "Domain Names - Implementation And Specification", RFC 1035, November 1987

[3] Karl Auerbach, "Protocol Standard For A Netbios Service On A Tcp/Udp Transport: Concepts And Methods", RFC 1001, March 1987

[4] Karl Auerbach, "Protocol Standard For A Netbios Service On A Tcp/Udp Transport: Detailed Specifications", RFC 1002, March 1987

[5] US National Bureau of Standards, "Data Encryption Standard", Federal Information Processing Standard (FIPS) Publication 46-1, January 1988

[6] Rivest, R. - MIT and RSA Data Security, Inc., "The MD4 Message Digest Algorithm", RFC 1320, April 1992

[7] X/Open Company Ltd., "X/Open CAE Specification - Protocols for X/Open PC Interworking: SMB, Version 2", X/Open Document Number: CAE 209, September 1992.

10. Security Considerations

There are four authentication mechanisms, each with their own strengths and weaknesses, as well as attacks that are independent of the authentication protocol.

10.1 Share level protection

In share level protection, there are no per-user passwords; knowledge of the read or write password is what gives read or write access. Since the passwords must be disclosed to be used, and hence known by many people, the scheme is quite weak.

In addition, the passwords are sent in the clear over the network, so they have all the weaknesses of clear text passwords in user level security.

10.2 Plaintext Password Authentication

This authentication protocol sends the client's password in the clear. It should be used only when needed for backwards compatibility, and only where the chances of eavesdropping is deemed acceptable, such as relatively isolated networks. Passwords sent to such servers should never be the same as passwords used for more secure servers.

10.3 LANMAN 2.1 (and earlier) Challenge/Response

This authentication protocol is subject to the following vulnerabilities:

- Known plaintext attack
- Small key space
- Chosen plaintext attack
- Dictionary attack
- Badly chosen passwords

These attacks, if successful, compromise the client's password, and allow the attacker access to the client's files even after the client's session has ended. Because of these attacks, it should be used only when needed for backwards compatibility, or where the chances of eavesdropping are deemed acceptable, such as relatively isolated networks. It is more secure than plaintext password authentication, because passwords are never seen in the clear on the network. Whenever possible, the use of the NT LM 0.12 authentication is to be preferred.

10.3.1 Known Plaintext Attacks

Because the challenge is plaintext, an eavesdropper can acquire known plaintext/ciphertext pairs. It can then test a guess at a password by using it to generate a key, encrypting the plaintext, and comparing it to the corresponding ciphertext.

10.3.2 Small Key Space

The combination of the use of only uppercase characters, the usual user practice of choosing passwords that have alpha and perhaps numeric characters, plus the fact that the protocol treats the upper and lower halves of the 14 bytes key almost identically means that the key space is rather small. Enumerating 7 uppercase characters and digits leads to a key space of 36**7, or 78.3 billion combinations. When this mechanism was introduced nearly a decade ago, this was probably an adequately large key space, but with today's much more powerful systems, it is now small enough to make a brute force search feasible upon a plaintext/ciphertext pair obtained via a known plaintext attack.

10.3.3 Chosen Plaintext Attacks

A "main-in-the-middle" or a counterfeit server can choose the challenge "C8" which the client will then encrypt using a key derived from the client's password. The ability to choose the plaintext to be encrypted is known to make breaking many ciphers much easier.

10.3.4 Dictionary Attacks

The attacker can precompute the response for many common passwords to a challenge of its choice, and build a dictionary of (response, password) pairs. It can then use the chosen plaintext attack to acquire a response corresponding to that challenge, and just look up the password in the dictionary.

10.3.5 Badly Chosen Passwords

Passwords that are not long enough, or that are words in the language of the clients, make the above attacks even easier by reducing the key space even more.

10.4 NT LM 0.12 Challenge/Response

Because it uses MD4 to generate the keying material from the password, and because it preserves the password's case, the key space of this protocol is essentially the full 56 bits that single DES allows; this is probably an acceptable length for most purposes (although future dialects may use triple-DES for more assurance). However, it is still subject to the same chosen plaintext and dictionary attacks as LANMAN 2.1 challenge/response if passwords are badly chosen. The only cure is to make sure that passwords are well-chosen, and long enough to have at least 56 bits of randomness.

Other considerations:

- Transforming the password into Unicode leaves a pattern of alternating zeros and characters in the input to MD4. This may allow MD4 to be reversed much more easily, although there is currently no known way to exploit this.

- MD4 is known to be weak with respect to collisions. There may be a way to exploit this to attack its one-wayness, or to exploit the collision properties to limit key search time, although there is currently no known way to do so.

10.5 Other attacks

10.5.1 Hijack connections

Any attacker that can inject packets into the network that appear to the server to be coming from a particular client, can hijack that client's connection. Once a connection is set up and the client has authenticated, subsequent packets are not authenticated, so the attacker can inject requests to read, write, or delete files to which the client has access. Doing so require that the injected packets have the right transport level sequence numbers, which can be tricky, especially if the client is sending packets at the same time. It is significantly more difficult to hijack a connection than to eavesdrop, and doing so only permits the attacker to access files as the client for the duration of the session.

10.5.2 Downgrade attack

A "man-in-the-middle" can remove the bit in the SMB_COM_NEGPROT response that says the server supports challenge/response, thus fooling a client into thinking that it should supply a plaintext password. This attack can be mitigated if clients are able to be configured to require challenge/response, either in general or for particular servers.

10.5.3 Spoofing by Counterfeit Servers

A counterfeit server is one that spoofs the DNS name resolution process so that the client gets the counterfeit's IP address instead of the genuine server's IP address, thus fooling the client into connecting to the counterfeit while believing it is connecting to the genuine server. Counterfeit servers are not detectable by the challenge/response authentication mechanism, which only authenticates clients. A counterfeit server can use the downgrade attack above to obtain a client's password; it can also execute a denial of service attack by denying the client's requests or returning bogus results. Counterfeit server attacks can be mitigated by deployment of DNSSEC.

10.5.4 Storing Passwords Safely

The passwords used in any of the authentication mechanisms used by this protocol have to be protected from access from over the network and from physical access. If the server does not support access control at the individual file level, but only at the file tree

level, then password files can not be placed in a file tree that is accessible from the network, as all files in such a tree have to be at least equally readable.

11. Author's Addresses

Isaac Heizer
Paul Leach
Dan Perry
Microsoft
1 Microsoft Way
Redmond, WA 98052
isaache@microsoft.com
paulle@microsoft.com
danp@microsoft.com

12.