Microsoft Networks

# SMB FILE SHARING PROTOCOL

Document Version 6.0p

January 1, 1996

*Microsoft Corporation*

## Introduction

This document describes the Lan Manager Server Message Block (SMB) file sharing protocol. Client systems use this protocol to request file, print, and communications service from server systems over a network.

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. For example, two NT systems would agree to use the NT-specific protocol dialect, while a Windows For Workgroups client communicating with an NT server might negotiate a Windows For Workgroups dialect. Different dialects can include both new messages as well as changes to the fields and semantics of existing messages in other dialects.

## Resource Sharing Connections

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. Windows for Workgroups and Windows 95 servers, for instance, implement the share level security model.
- 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. NT servers and LM/U servers implement this security model and do not support the *share level* model. User level servers are preferred over share level servers for any new server implementation, since corporations generally find *user level* servers easier to administer as employees come and go.

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 resouce 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.

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.

## 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:

```
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
        struct {
            USHORT PidHigh;             // High part of PID (NT Create And X)
                struct {
                    ULONG  HdrReserved;     // 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 vairous client capabilities.
- *PidHigh* is used in the *NtCreateAndX* request SMB
- *Connectionless. Sid*, and *Connectionless.SequenceNumber* are used when the client to server connection is on a datagram oriented protocol such as IPX or UDP.
- *StreamProtocol.SMBLength* is used to frame this SMB when the client to server connection is on a byte stream protocol such as TCP. It is the entire length of the SMB from the initial 0xFF to the final byte.
- *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.

## *Sample Messge Flow*

The following illustrates a typical message exchange for a client connecting to a user level server, opening a file, reading its data, closing the file, and disconnecting from the server.

| 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 |

## *SMB Protocol Dialects*

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.

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 equivilant 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 equivilant 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. 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.

## *Message Transport*

Clients and servers exchange messages over either a reliable NetBIOS transport or a connectionless transport such as IPX.

### Reliable NetBIOS Transports

The client and server can use NETBIOS to establish and maintain communications. The server 'posts' a name on the network and the client connects to that name. For compatibility with pre-Windows 95 and pre-Windows NT clients, a server's NetBIOS name should comply with the standard DOS 8.3 format and be blank padded to the right. All NETBIOS-based SMB servers have a name whose 16-th character is 20 hex.

When using such a reliable message-oriented transport, the SMB protocol makes no higher level attempts to ensure reliable sequenced delivery of messages between the client and server. The transport should 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 from a client terminates, all work in progress by that client is terminated and all resources open by that client are closed.

The rules for reliable transport establishment and 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 resouces 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.

## Connectionless IPX Transport

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.

Five IPX sockets are used as follows:

| Socket Name | Value | Purpose |
|---|---|---|
| SMB_SERVER_SOCKET | 0x0550 | SMB requests from clients |
| SMB_NAME_SOCKET | 0x0551 | name claims and name query messages |
| REDIR_SOCKET | 0x0552 | is used by the redirector for sending SMB requests and receiving SMB replies. |
| MAILSLOT_SOCKET | 0x0553 | is used by the redirector and browser for mailslot datagrams. |
| MESSENGER_SOCKET | 0x0554 | is used by the redirector to send messages from client to client (NetMessageBufferSend). |

The SMB header includes two fields specifically designed for use on IPX. *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 Negotiate Protocol 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 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 maximum packet size for some IPX routers is 576 bytes including the IPX header. Because of this, the client must limit the size of the negotiated buffer size to 546 bytes when the server's network ID is not the same as the client's network ID. 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.

Sequenced commands are used for operations which cause a state change on the server that cannot be repeated. 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. For example, reading or writing to a disk file. The server maintains a small save area for each client to keep the response information from the previous sequenced command. Because the server has a limited amount of space available for this save area, the client must send all commands with a large response size as unsequenced. Such commands include file read and file search. 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 in 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. 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.

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, enumerate servers, 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 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. The NT server has a default timeout value of 15 minutes.

## Naming On Ipx

The name claim/query packet and mailslot datagram packets use the structure:

```
struct ipxnm {
        uchar   inm_route[32];          /* routing info (used by IPX routers) */
        uchar   inm_op;                 /* operation being requested */
        uchar   inm_type;               /* type of name */
        ushort  inm_msgid;              /* message ID for sender */
        uchar   inm_name[16];           /* name being sought or claimed */
        uchar   inm_srcname[16];        /* name of requesting machine */
};
```

Below are the values for inm_op:

```
INAME_CLAIM             0xf1    // server name claim message
INAME_DELETE    0xf2    // relinquish server name
INAME_QUERY     0xf3    // locate server name
INAME_FOUND     0xf4    // response to INAME_QUERY
IMSG_HANGUP             0xf5    // Messenger Hangup (contained in SMB)
IMSLOT_SEND             0xfc    // packet contains mslot write, no resp needed
IMSLOT_FIND     0xfd    // find name for mslot write, no data included
IMSLOT_NAME     0xfe    // find name response
```

The following are the values for inm_type:

```
INTYPE_MACHINE          1
INTYPE_WKGROUP          2
INTYPE_BROWSER          3
```

When the server starts, it sends broadcasts a name claim (inm_type == INAME_CLAIM) packet five (5) times at 500 millisecond intervals. The server's name is put into both the inm_name and the inm_srcname fields. The IPX packet type is 32 (0x20) which IPX routers will forward through up to eight (8) hops. If no other machine responds within 500 milliseconds of the transmission of the last broadcast, the server claims the name as its own.

When a client wishes to locate the address of a server, it broadcasts a name query (inm_type == INAME_QUERY) packet with the server's name in inm_name and its own name in inm_srcname. The first broadcast is an IPX type 4 packet which is not forwarded by routers. After 500 milliseconds, the client will perform an all nets broadcast (IPX type 32) four (4) times at 500 milliseconds intervals. The client extracts the server's address from the IPX header of the response packet.

Once a client has the address of a server, it may open a circuit to the server by sending a negotiate SMB to the SMB_SERVER_SOCKET. In the negotiate request, smb_sid must be zero (0), smb_seq must be one (1) and two 16 bytes names are appended to the end of the SMB in NetBIOS name format. The size of the names is NOT included in the smb_bcc value. The first 16 bytes contains the client computer name space padded with a zero (0) in the 16th position. The second 16 bytes contains the remote server's name space padded to 16 bytes. The server retains the client's name for informational purposes and verifies that the server name matches its own name. If the server name does not match, the server will respond to the negotiate with an ERRSRV class error ERRnotme. The server returns a session identifier in smb_sid which the client must place in smb_sid in all subsequent requests sent to the server.

## *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:

1. An *exclusive* oplock allows a client to open a file for exclusive access and allows the client to perform arbitrary buffering
2. 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.
3. 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.

## 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.

## 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:

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

### 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*.

## *NAMED PIPES*

Named pipes provide a facility which allows interprocess communications pipes to be named and act like full duplex virtual circuits between a pair of endpoints. Support of named pipes is server optional, and the earliest valid dialect supporting named pipes is LANMAN1.0.

### Named Pipe Features

- Pipes are named and accessed across a network.

- Once created, named pipes can be opened and read/written like standard files, i.e., using Open, Read, Write, and Close protocols.

- Named pipes support message as well as byte stream modes.

- Byte stream mode lets processes read and write byte streams, exactly like byte conventional pipes, except the pipe is full-duplex, emulating a virtual circuit.

- Message mode lets processes read and write streams of messages (as opposed to bytes). Message mode is optimized for peer-to-peer communication between remote as well as local processes.

- Named pipes can be serially re-used by different clients (closed and reopened by another process).

- A serving process can create multiple identically named pipes so that multiple clients opening to that name will get distinct pipes to the serving process.

Named pipes are generally used to support some API requests to the server. In early incarnations of Lan Manager networking, use of named pipes for generalized client to server communications was encouraged. Microsoft subsequently provided tools and runtime support for generalized DCE compliant RPC exchanges between clients and servers. The RPC runtime can use named pipes, datagrams, or direct use of transport facilities to communicate between clients and servers, and the RPC MIDL complier allows high level expression of the messages to be exchanged between clients and servers. Developers are strongly encouraged to use the RPC tools to implement client and server protocols rather than coding directly to any named pipe interfaces.

## SMB Messages And Formats

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.

### *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.

## 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 acknowledgement 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 consumer is requesting that the file be "opportunisticly" 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 coresponding response SMB to indicate to the consumer 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 consumer to the server as in the matching response from the server to the consumer. This bit unambiguously distinguishes the command request from the command response. | PC NETWORK PROGRAM 1.0 |

## 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 | Earilest 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 consumer is aware of extended attributes | |
| 2 | If set, SMB_FLAGS2_IS_LONG_NAME | |
| 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 |

## 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.

## 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.

## 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.

## 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 incomming 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.

## 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.

## 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 labelled as STRING. If the encoding is ASCII, even if the negotiatiated string is Unicode, the quantity is labelled 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.

## Time And Date Encoding

When SMB requests or responses encode time values, the following describes the encoding into 16 bits.

```
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.

## 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

## *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 |

## *"ANDX" SMB 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:

1. The embedded command does not repeat the SMB header information. Rather the next SMB starts at the *WordCount* field.
2. 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.

3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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 consumer 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.

## *SMB MESSAGES*

### Valid SMB Messages 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
```

## 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. Implementors 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; | Count of parameter words = 1 |
| USHORT DialectIndex; | Index of selected dialect |
| USHORT ByteCount; | 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:<br>bit 0: 0 = share, 1 = user<br>bit 1: 1 = encrypt passwords |
| 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:<br>bit 0: 1 = Read Raw supported<br>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 encrypt passwords, the client should use the *EncryptionKey* to encrypt transmitted passwords. Current servers specify an *EncryptionKeyLength* of 8.

Clients should submit no more than *MaxMpxCount* distinct unanswered SMBs to the server.

MICROSOFT NETWORKS 1.03 clients 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 LANAMN2.1 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:<br>bit 0: 0 = share, 1 = user<br>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 | |
| | | |
| | | |

## SESSION_SETUP_ANDX: Session Setup And X

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; | Consumer'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 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:<br>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 |

Because *AccountPassword* may be encrypted, it is a variable length field with the length specified by *PasswordLength* (if password encryption is not being used, *AccountPassword* should be a null terminated ASCII string with *PasswordLength* set to the string size including the null). The password is case insensitive.

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

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 consumer. Here *MaxBufferSize* is the maximum message size which the consumer can receive. Thus although the server may support 16k buffers (as returned in the SMB_COM_NEGOTIATE response), if the consumer 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 max SMB message size which the consumer 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 consumer. Thus if the server's buffer sizewere 4k and the consumer's buffer size were only 2K, the consumer 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.

The *VcNumber* field specifies whether the consumer 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; | Consumer'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 max 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_OPEN_PRINT_FILE
SMB_COM_GET_PRINT_QUEUE             SMB_COM_TRANSACTION
SMB_COM_NO_ANDX_COMMAND
```

## LOGOFF_ANDX: User Logoff And X

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.

## 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 succesful 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, admin 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.

## TREE_CONNECT_ANDX: Tree Connect And X

| 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<br>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 | |

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_OPEN_PRINT_FILE | SMB_COM_GET_PRINT_QUEUE |
| SMB_COM_TRANSACTION | SMB_COM_NO_ANDX_COMMAND |

## 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; | Count of parameter words = 0 |
| USHORT ByteCount; | 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;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

## 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;<br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING DirectoryName[]; | Count of parameter words = 0<br>Count of data bytes; min = 2<br>0x04<br>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;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

## 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;<br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING DirectoryName[]; | Count of parameter words = 0<br>Count of data bytes; min = 2<br>0x04<br>Directory name |

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

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

## 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;<br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING DirectoryPath[]; | Count of parameter words = 0<br>Count of data bytes; min = 2<br>0x04<br>Directory path |

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

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

## 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;<br>USHORT DesiredAccess;<br>USHORT SearchAttributes;<br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING FileName[]; | Count of parameter words = 2<br>Mode - read/write/share<br><br>Count of data bytes; min = 2<br>0x04<br>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 referrs 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;<br>USHORT Fid;<br>USHORT FileAttributes;<br>SMB_DATE LastWriteTime;<br>SMB_TIME LastWriteDate;<br>ULONG DataSize;<br>USHORT GrantedAccess;<br>USHORT ByteCount; | Count of parameter words = 7<br>File handle<br>Attributes of opened file<br>Time file was last written<br>Date file was last written<br>File size<br>Access allowed<br>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 clientss 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.

## 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 |
| SMB_TIME CreationTime; | Time file was created |
| SMB_DATE CreationDate; | Date 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* and *CreationDate* fields 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.

## 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; | Count of parameter words = 3 |
| USHORT Fid; | File handle |
| SMB_TIME LastWriteTime | Time of last write |
| SMB_DATE LastWriteDate; | Date of last write |
| USHORT ByteCount; | 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; | Count of parameter words = 0 |
| USHORT ByteCount; | Count of data bytes = 0 |

## 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; | Count of parameter words = 1 |
| USHORT Fid; | File handle |
| USHORT ByteCount; | 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; | Count of parameter words = 0 |
| USHORT ByteCount; | Count of data bytes = 0 |

## 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 "wild cards" in the last component of *FileName*. "?" is the wild card for single characters, "*" or "null" matches any number of filename characters within a single part of the filename component. The filename is divided into two parts -an eight character name and a three character extension. The name and extension are divided by a ".".

If a filename part commences with one or more "?"s then exactly that number of characters will be matched by the wildcards, 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

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

*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.j

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

## RENAME: Rename File

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

| Client Request | Description |
|---|---|
| UCHAR WordCount;<br>USHORT SearchAttributes;<br>USHORT ByteCount;<br>UCHAR BufferFormat1;<br>STRING OldFileName[];<br>UCHAR BufferFormat2;<br>STRING NewFileName[]; | Count of parameter words = 1<br>Target file attributes<br>Count of data bytes; min = 4<br>0x04<br>Old file name<br>0x04<br>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 "wild cards" in the file name (last component of the pathname). The wild card matching algorithm is described in the SMB_COM_DELETE description.

*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; | Count of parameter words = 0 |

| | |
|---|---|
| USHORT ByteCount; | Count of data bytes = 0 |

## QUERY_INFORMATION: Get File Attributes

This request is sent to obtain information about a file.

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

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

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT FileAttributes;<br>SMB_TIME LastWriteTime;<br>SMB_DATE LastWriteDate;<br>ULONG FileSize;<br>USHORT Reserved [5];<br>USHORT ByteCount; | Count of parameter words = 10<br><br>Time of last write<br>Date of last write<br>File size<br>Reserved - client should ignore<br>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.

## SET_INFORMATION: Set File Attributes

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

| Client Request | Description |
|---|---|
| UCHAR WordCount;<br>USHORT FileAttributes;<br>SMB_TIME LastWriteTime;<br>SMB_DATE LastWriteDate;<br>USHORT Reserved [5];<br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING FileName[]; | Count of parameter words = 8<br>Attributes of the file<br>Time of last write<br>Date of last write<br>Reserved (must be 0)<br>Count of data bytes; min = 2<br>0x04<br>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* and *LastWriteDate* fields contain zero then the file's time is not changed.

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

## 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* referrs 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 max-xmit-size for the *Tid* specified, the server will abort the connection to the consumer.

## 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.

## 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 prevent attempts to lock, read or write the locked portion of the file by other clients or *Pid*s. Overlapping locks are not allowed. *Offset*s 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

## 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 |

## 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 |
| SMB_TIME CreationTime; | New file's time stamp |
| SMB_DATE CreationDate; | New file's date 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* and *CreationDate* by the server is optional.

## 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;<br>USHORT FileAttributes;<br>SMB_TIME CreationTime;<br>SMB_DATE CreationDate;<br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING FileName[]; | Count of parameter words = 3<br>New file attributes<br>Time of created file<br>Date for created file<br>Count of data bytes; min = 2<br>0x04<br>File name |

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

*CreationTime* and *CreationDate* are the timestamp the file should be given, server support for these is optional.

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT Fid;<br>USHORT ByteCount; | Count of parameter words = 1<br>File handle<br>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.

## PROCESS_EXIT: Process Exit

This command informs the server that a consumer 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;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | Count of parameter words = 0<br>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.

## SEEK: Seek in File

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

| Client Request | Description |
|---|---|
| UCHAR WordCount;<br>USHORT Fid;<br>USHORT Mode;<br><br><br><br>LONG Offset;<br>USHORT ByteCount; | Count of parameter words = 4<br>File handle<br>Seek mode:<br>0 = from start of file<br>1 = from current position<br>2 = from end of file<br>Relative offset<br>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;<br>ULONG Offset;<br>USHORT ByteCount; | Count of parameter words = 2<br>Offset from start of file<br>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 .

## SMB_QUERY_INFORMATION_DISK: Get Disk Attributes

This 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;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT TotalUnits;<br>USHORT BlocksPerUnit;<br>USHORT BlockSize;<br>USHORT FreeUnits;<br>USHORT Reserved;<br>USHORT ByteCount; | Count of parameter words = 5<br>Total allocation units per server<br>Blocks per allocation unit<br>Block size (in bytes)<br>Number of free units<br>Reserved (client should ignore)<br>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 accomodate. If after all adjustment, the numbers are still too high, the largest possible values for TotalUnit or FreeUnits (i.e. 0xFFFF) should be returned.

## SEARCH: Search Directory

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 - comsumer use<br>bits 5,6 - system use (must preserve)<br>bits 0-4 - server use (must preserve) |
| UCHAR FileName[11]; | Name of the returned file |

| | |
|---|---|
| UCHAR ReservedForServer[5];<br>UCHAR ReservedForConsumer[4]; | Client must not modify<br>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 "wild cards" in the last component of *FileName* of the initial request. The wild card matching algorithm is described in the SMB_COM_DELETE description.

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

| Directory Information Field | Description |
|---|---|
| SMB_RESUME_KEY ResumeKey;<br>UCHAR FileAttributes;<br>SMB_TIME LastWriteTime;<br>SMB_DATE LastWriteDate;<br>ULONG FileSize;<br>UCHAR FileName[13]; | Described above<br>Attributes of the found file<br>Time file was last written<br>Date file was last written<br>Size of the file<br>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*.

## 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;<br>USHORT SetupLength;<br>USHORT Mode;<br><br>USHORT ByteCount;<br>UCHAR BufferFormat;<br>STRING IdentifierString[]; | Count of parameter words = 2<br>Length of printer setup data<br>0 = Text mode (DOS expands TABs)<br>1 = Graphics mode<br>Count of data bytes; min = 2<br>0x04<br>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.

## 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.

## 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; | Count of parameter words = 1 |
| USHORT Fid; | File handle |
| USHORT ByteCount; | Count of data bytes = 0 |

*Fid* referrs 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; | Count of parameter words = 0 |

| | |
|---|---|
| USHORT ByteCount; | 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.

## 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:<br>01 = held or stopped<br>02 = printing<br>03 = awaiting print<br>04 = in intercept<br>05 = file had error<br>06 = printer error<br>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.

## 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 max-xmit-size for the *Tid* specified, the server will abort the connection to the consumer.

## 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 consumer can buffer the locked bytes locally while they are being updated, then when the unlock request is received submit this protocol to both write and then unlock bytes. 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.

## 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 consumer. 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 (max 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* referrs 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 consumer should then retry using a dofferent 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 accomodates very large files having 64 bit offsets.

| Client Request | Server Response |
|---|---|
| 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 (max 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 differented 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.

## 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 (max 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* referrs 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 consumer 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 consumer'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 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:

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

## 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 consumer 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:<br>bit 0 - complete write to disk and send final result response<br>bit 1 - return Remaining (pipe/dev)<br>(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 accomodates 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:<br>bit 0 - complete write to disk and send final result response<br>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.

## 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:<br>bit 0 - complete write to disk and send final result response<br>bit 1 - return Remaining<br>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 consumer 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.

## 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.

## 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.

## LOCKING_ANDX: Lock or UnLock Bytes

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 | Readonly 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; | Ofset 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; | Ofset to bytes to [un]lock (high) |
| ULONG OffsetLow; | Ofset 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 endof-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 consumer 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 max 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 | |

## 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:<br>0 - target must be a file<br>1 - target must be a directory<br>2 - reserved (must be 0)<br>3 - reserved (must be 0)<br>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.

## 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:<br>bit 0 - target must be a file<br>bit 1 - target must ba a dir.<br>bit 2 - copy target mode:<br>0 = binary, 1 = ASCII<br>bit 3 - copy source mode:<br>0 = binary, 1 = ASCII<br>bit 4 - verify all writes<br>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.

*OpenFunction* controls what should happen if the destination file exists, and has the following bit mapping:

bits:

```
1111 11
5432 1098 7654 3210
rrrr rrrr rrrC rrOO
```

where:

O - Open (action to be taken if destination file exists).
- 0 - Fail.
- 1 - Append file.

2 - Truncate file.

r - reserved (must be zero).

C - Create (action to be taken if destination file does not exist).
0 -- Fail.
1 -- Create 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* . 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.

## 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;<br>USHORT EchoCount;<br>USHORT ByteCount;<br>UCHAR Buffer[1]; | Count of parameter words = 1<br>Number of times to echo data back<br>Count of data bytes; min = 1<br>Data to echo |

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

Each response echos 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.

## 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 |
| SMB_TIME 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 |
| SMB_TIME LastWriteTime; | Time of last write |
| SMB_DATE LastWriteDate; | Date 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.

## OPEN_ANDX: Open File And X

| 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-<br>0 - return additional info<br>1 - exclusive oplock requested<br>2 - batch oplock requested |
| USHORT DesiredAccess; | File open mode |
| USHORT SearchAttributes; | |
| USHORT FileAttributes; | |
| SMB_TIME CreationTime; | |
| SMB_DATE CreationDate; | |
| 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; | |
| SMB_TIME LastWriteTime; | |
| SMB_DATE LastWriteDate; | |
| 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. 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.

*Action* in the response 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.

*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 |

*DeviceState* is applicable only if the *FileType* is FileTypeByteModePipe or FileTypeMessageModePipe and is encoded as follows:

```
5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
B E * * T T R R
```

where:

B - Blocking - 0 => reads/writes block if no data available
1 => reads/writes return immediately if no data
E - Endpoint - 0 => consumer end of pipe
1 => server end of pipe
TT - Type of pipe - 00 => pipe is a byte stream pipe
01 => pipe is a message pipe
RR - Read Mode - 00 => Read pipe as a byte stream
01 => Read messages from pipe

If bit0 of *Flags* is clear, the *FileAttributes*, *LastWriteTime*, *LastWriteDate*, *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

## NT_CREATE_ANDX: Create File

This command is used to create or open a file or a directory. Many of the parameters are passed directly to the NT open functions.

| 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:<br>0x02 - Request an oplock<br>0x04 - Request a batch oplock<br>0x08 - Target of open must be directory |
| ULONG RootDirectoryFid; | If non-zero, open is relative to this directory |
| ACCESS_MASK DesiredAccess; | NT 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 QOS information<br>1 - Dynamic Tracking<br>2 - Effective only |
| USHORT ByteCount; | Length of byte parameters |
| STRING Name[]; | File to open or create |

| 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

## 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.

*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.

**DataCompactionMode.**

SMB_COM_CLOSE is the only valid command for *AndXCommand*.

## 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:<br>0 - write through<br>1 - return Remaining<br>2 - use WriteRawNamedPipe (n. pipes)<br>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:<br>0 - write through<br>1 - return Remaining<br>2 - use WriteRawNamedPipe (n. pipes)<br>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 | |

## TRANSACTIONS

`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).

### *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:<br>bit 0 - also disconnect TID in *Tid*<br>bit 1 - one-way transacion (no resp) |
| ULONG Timeout; | |
| USHORT Reserved2; | |
| USHORT ParameterCount; | Parameter bytes sent this buffer |
| USHORT ParameterOffset; | Offset (from header start) to params |
| 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 params |
| USHORT ParameterDisplacement; | Displacement of these param 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 params |
| USHORT ParameterDisplacement; | Displacement of these param 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) |

### *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 params |
| 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 params |
| 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 |

### *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 reasemble 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 reasemble 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 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 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.

## *SMB_COM_TRANSACTION Operations*

### Mail Slot Transaction Protocol

The only transaction allowed to a mailslot is a mailslot write. The following table shows the interpretation of parameters for a mailslot transaction:

| Name | Value | Description |
|---|---|---|
| Command | `SMB_COM_TRANSACTION` | |
| Name | \MAILSLOT\<name> | STRING Name of mail slot to write |
| SetupCount | 3 | |
| Setup[0] | 1 | Command code == write mailslot |
| Setup[1] | | Ignored |
| Setup[2] | | Ignored |
| TotalDataCount | n | Size of data to write to the mailslot |
| Data[ n ] | | The data to write to the mailslot |

### Named Pipe Transaction Protocol

A named pipe `SMB_COM_TRANSACTION` is used to wait for the specified named pipe to become available (WaitNmPipe) or perform a logical "open → write → read → close" of the pipe (CallNmPipe), along with other functions defined below.

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

| Name | Value | Description |
|---|---|---|
| Command | `SMB_COM_TRANSACTION` | |
| Name | \PIPE\<name> | Name of pipe for operation |
| SetupCount | 2 | |
| Setup[0] | See Below | Subcommand code |
| Setup[1] | *Fid* of pipe | If required |
| TotalDataCount | n | Size of data |
| Data[ n ] | | If required |

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) */ |

### CallNamedPipe

This command is used to implement the Win32 CallNamedPipe() API remotely. The CallNamedPipe function connects to a message-type pipe (and waits if an instance of the pipe is not available), writes to and reads from the pipe, and then closes the pipe.

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

The number of bytes being written is defined by *TotalDataCount* and the max number of bytes to return is defined by *MaxDataCount*.

On the response *TotalParameterCount* is 0 (no param bytes to return), *TotalDataCount* indicates the amount of databytes being returned in total and *DataCount* identifies the amount of data being retuned in each buffer.

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

### WaitNamedPipe

The command is used to implement the Win32 WaitNamedPipe() API remotely. The WaitNamedPipe function waits until either a time-out interval elapses or an instance of the specified named pipe is available to be connected to (that is, the pipe's server process has a pending ConnectNamedPipe operation on the pipe).

The server will wait up to *Timeout* milliseconds for a pipe of the name given to become available. Note that although the timeout is specified in milliseconds, by the time that the timeout occurs and the client receives the timed out response much more time than specified may have occurred.

This form of the transaction protocol sends no data or parameter bytes. The response also contains no data or parameters. If the transaction response indicates success, the pipe may now be available. However, this request does not reserve the pipe, thus all waiting programs may race to get the pipe now available. The losers will get an error on the pipe open attempt.

### PeekNamedPipe

This form of the pipe Transaction protocol is used to implement the Win32 PeekNamePipe() API remotely. The PeekNamedPipe function copies data from a named or anonymous pipe into a buffer without removing it from the pipe. It also returns information about data in the pipe.

*TotalParameterCount* and *TotalDataCount* should be 0 for this request. The *Fid* of the pipe to which this request should be applied is in Setup[1]. *MaxParameterCount* should be set to 6, requesting 3 words of information about the pipe, and *MaxDataCount* should be set to the number of bytes to "peek".

The response contains the following *Parameter words*:

| Name | Description |
|---|---|
| Parameters[0, 1] | Total number of bytes available to be read from the pipe |
| Parameters[2,3] | Total number of bytes remaining in the message at the "head" of the pipe |
| Parameters[4,5] | Pipe status.<br>1 Disconnected by server<br>2 Listening<br>3 Connection to server is OK<br>4 Server end of pipe is closed |

The *Data* portion of the response is the data peeked from the named pipe.

## GetNamedPipeHandleState

This form of the pipe transaction protocol is used to implement the Win32 GetNamedPipeHandleState() API. The GetNamedPipeHandleState function retrieves information about a specified named pipe. The information returned can vary during the lifetime of an instance of the named pipe.

This request sends no parameters and no data. The *Fid* of the pipe to which this request should be applied is in Setup[1]. *MaxParameterCount* should be set to 2 (requesting the 1 word of information about the pipe) and *MaxDataCount* should be 0 (not reading the pipe).

The response returns one parameter of pipe state information interpreted as:

Pipe Handle State Bits

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

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

where:

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

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

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

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

Icount - 8-bit count to control pipe instancing

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

## SetNamedPipeHandleState

This form of the pipe transaction protocol is used to implement the Win32 SetNamedPipeHandleState() API. The SetNamedPipeHandleState function sets the read mode and the blocking mode of the specified named pipe.

This request sends 1 parameter word (*TotalParameterCount* = 2) which is the pipe state to be set. The *Fid* of the pipe to which this request should be applied is in *Setup[1]*.

The response contains no data or parameters.

The interpretation of the input parameter word is:

Pipe Handle State Bits

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

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

where:

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

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

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

### GetNamedPipeInfo

This form of the pipe transaction protocol is used to implement the Win32 GetNamedPipeInfo() API. The GetNamedPipeInfo function retrieves information about the specified named pipe.

The request sends 1 parameter word (*TotalParameterCount* = 2) which is the information level requested and must be set to 1. The *Fid* of the pipe to which this request should be applied is in *Setup[1]*. *MaxDataCount* should be set to the size of the buffer specified by the user in which to receive the pipe information.

Pipe information is returned in the data area of the response, up to the number of bytes specified. The information is returned in the following format:

| Name | Size | Description |
|---|---|---|
| OutputBufferSize | USHORT | actual size of buffer for outgoing (server) I/O |
| InputBufferSize | USHORT | actual size of buffer for incoming (client) I/O |
| MaximumInstances | UCHAR | Maximum allowed number of instances |
| CurrentInstances | UCHAR | Current number of instances |
| PipeNameLength | UCHAR | Length of pipe name (including the null) |
| PipeName | STRING | Name of pipe (NOT including \\NodeName - \\NodeName is prepended to this string by the client before passing back to the user) |

### TransactNamedPipe

This form of the pipe transaction protocol is used to implement the Win32 TransactNamedPipe() API. The TransactNamedPipe function combines into a single network operation the functions that write a message to and read a message from the specified named pipe.

It provides an optimum way to implement transaction-oriented dialogs. TransactNamedPipe will fail if the pipe currently contains any unread data or is not in message read mode. Otherwise the call will write the entire request data bytes to the pipe and then read a response from the pipe and return it in the data bytes area of the response protocol. In the transaction request, *Setup[1]* must contain the *Fid* of the pipe.

If *Name* is \PIPE\LANMAN, this is a server API request. The request encoding is:

| Request Field | Description |
|---|---|
| Parameters[0→1] | API # |
| Parameters[2→N] | ASCIIZ RAP description of input structure |
| Parameters[N→X] | The input structure |

The response is formatted as:

| Response Field | Description |
|---|---|
| Parameters[0→1] | Result Status |
| Parameters[2→3] | Offset to result structure |

The state of blocking/nonblocking has no effect on this protocol (TransactNamedPipe does not return until a message has been read into the response protocol). If *MaxDataCount* is too small to contain the response message, an error is returned.

### RawReadNamedPipe

RawReadNamedPipe reads bytes directly from a pipe, regardless of whether it is a message or byte pipe. For a byte pipe, this is exactly like `SMB_COM_READ`. For a message pipe, this is exactly like reading the pipe in byte read mode, except message headers will also be returned in the buffer (note that message headers will always be returned in toto--never split at a byte boundary).

This request sends no parameters or data to the server, and *Setup[1]* must contain the *Fid* of the pipe to read. *MaxDataCount* should contain the number of bytes to read raw.

The response will return 0 parameters, and *DataCount* will be set to the number of bytes read.

### RawWriteNamedPipe

RawWriteNamedPipe puts bytes directly into a pipe, regardless of whether it is a message or byte pipe. The data will include message headers if it is a message pipe. This call ignores the blocking/nonblocking state and always acts in a blocking manner. It returns only after all bytes have been written.

The request sends no parameters. *Setup[1]* must contain the *Fid* of the pipe to write. *TotalDataCount* is the total amount of data to write to the pipe. Writing zero bytes to a pipe is an error unless the pipe is in message mode.

The response contains no data and one parameter word. If no error is returned, the one parameter word indicates the number of the requested bytes that have been "written raw" to the specified pipe.

## *SMB_COM_TRANSACTION2 Operations*

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_OPEN2

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-<br>0 - return additional info<br>1 - exclusive oplock requested<br>2 - batch oplock requested<br>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. 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.

*Action* in the response specifies the action as a result of this 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.

| 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 fo 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 follows:

5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
B E * * T T R R

where:

B - Blocking - 0 => reads/writes block if no data available
1 => reads/writes return immediately if no data
E - Endpoint - 0 => consumer end of pipe
1 => server end of pipe
TT - Type of pipe - 00 => pipe is a byte stream pipe
01 => pipe is a message pipe
RR - Read Mode - 00 => Read pipe as a byte stream
01 => Read messages from pipe

If an error was detected in the incomming 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.

## TRANS2_FIND_FIRST2

| 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:<br>Bit 0 - close search after this request<br>Bit 1 - close search if end of search reached<br>Bit 2 - return resume keys for each entry found<br>Bit 3 - continue search from previous ending place<br>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;<br>USHORT SearchCount;<br>USHORT EndOfSearch;<br>USHORT EaErrorOffset;<br>USHORT LastNameOffset;<br><br>UCHAR Data[ TotalDataCount ] | Search handle<br>Number of entries returned<br>Was last entry returned?<br>Offset into EA list if EA error<br>Offset into data to file name of last entry, if server needs it to resume search; else 0<br>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.

### *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 |

### *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 |

### *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.

*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 |

*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 |

*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 |

*SMB_FIND_FILE_NAMES_INFO*

| Response Field | Description |
|---|---|
| ULONG NextEntryOffset;<br>ULONG FileIndex;<br>ULONG FileNameLength;<br>STRING FileName; | Offset from this structure to beginning of next one<br><br>Length of FileName in bytes<br>Files full length name |

### *TRANS2_FIND_NEXT2*

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

| Client Request | Value |
|---|---|
| WordCount<br>SetupCount<br>Setup[0] | 15<br>1<br>TRANS2_FIND_NEXT2 |
| **Parameter Block Encoding** | **Description** |
| USHORT Sid;<br>USHORT SearchCount;<br>USHORT InformationLevel;<br>ULONG ResumeKey;<br>USHORT Flags;<br><br><br><br><br><br>STRING FileName; | Search handle<br>Maximum number of entries to return<br>Levels described in `TRANS2_FIND_FIRST2` request<br>Value returned by previous find2 call<br>Additional information: bit set-<br>0 - close search after this request<br>1 - close search if end of search reached<br>2 - return resume keys for each entry found<br>3 - resume/continue from previous ending place<br>4 - find with backup intent<br>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 Parameter Block | Description |
|---|---|
| USHORT SearchCount;<br>USHORT EndOfSearch;<br>USHORT EaErrorOffset;<br>USHORT LastNameOffset;<br><br>UCHAR Data[TotalDataCount] | Number of entries returned<br>Was last entry returned?<br>Offset into EA list if EA error<br>Offset into data to file name of last entry, if server needs it to resume search; else 0<br>Level dependent info about the matches found in the search |

### *TRANS2_QUERY_FS_INFORMATION*

This transaction requests information about a filesystem on the server.

| Client Request | Value |
|---|---|
| WordCount;<br>TotalParameterCount;<br>MaxSetupCount;<br>SetupCount;<br>Setup[0]; | 15<br>2 or 4<br>0<br>1 or 2<br>TRANS2_QUERY_FS_INFORMATION |
| **Parameter Block Encoding** | **Description** |
| USHORT Information Level; | Level of information requested |

If the transaction request is TRANS2_QUERY_FS_INFORMATION, 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 `NtQueryVolumeInformatinFile` 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.

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 |

SMB_INFO_VOLUME

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

### *TRANS2_QUERY_PATH_INFORMATION*

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

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

The following InformationLevels may be requested:

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

SMB_INFO_STANDARD & SMB_INFO_QUERY_EA_SIZE

| Data Block Encoding | Description |
|---|---|

| | |
|---|---|
| SMB_DATE CreationDate;<br>SMB_TIME CreationTime;<br>SMB_DATE LastAccessDate;<br>SMB_TIME LastAccessTime;<br>SMB_DATE LastWriteDate;<br>SMB_TIME LastWriteTime;<br>ULONG DataSize;<br>ULONG AllocationSize;<br>USHORT Attributes;<br>ULONG EaSize; | Date when file was created<br>Time when file was created<br>Date of last file access<br>Time of last file access<br>Date of last write to the file<br>Time of last write to the file<br>File Size<br>Size of filesystem allocation unit<br>File Attributes<br>Size of file's EA information (SMB_INFO_QUERY_EA_SIZE) |

#### 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;<br>UCHAR EaList[] | Length of the remaining data<br>The extended attributes list |

#### 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.

### *TRANS2_SET_PATH_INFORMATION*

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

| Client Request | Value |
|---|---|
| WordCount<br>MaxSetupCount<br>SetupCount<br>Setup[0] | 15<br>0<br>1<br>TRANS2_SET_PATH_INFORMATION |
| Parameter Block Encoding | Description |
| USHORT InformationLevel;<br>ULONG Reserved;<br>STRING FileName; | Level of information to set<br>Must be zero<br>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:

SMB_INFO_STANDARD & SMB_INFO_QUERY_EA_SIZE

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

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;<br>UCHAR EaList[] | Length of the remaining data<br>The extended attributes list |

### *TRANS2_QUERY_FILE_INFORMATION*

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

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

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

### *TRANS2_SET_FILE_INFORMATION*

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<br>MaxSetupCount<br>SetupCount<br>Setup[0] | 15<br>0<br>1<br>TRANS2_SET_FILE_INFORMATION |
| Parameter Block Encoding | Description |
| USHORT Fid; | Handle of file for request |

| | |
|---|---|
| USHORT InformationLevel;<br>USHORT Reserved; | Level of information requested<br>Ignored by the server |

The following *InformationLevels* may be set:

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

### *TRANS2_CREATE_DIRECTORY*

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 | 15 |
| MaxSetupCount | 0 |
| SetupCount | 1 |
| Setup[0] | TRANS2_CREATE_DIRECTORY |
| **Parameter Block Encoding** | **Description** |
| ULONG Reserved;<br>STRING Name[];<br>UCHAR Data[]; | Reserved--must be zero<br>Directory name to create<br>Optional FEAList for the new directory |

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

## *SMB_COM_NT_TRANSACTION Operations*

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 |

The following sections describe these requests.

## NT_TRANSACT_CREATE

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<br>0 - No oplock granted<br>1 - Exclusive oplock granted<br>2 - Batch oplock granted<br>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.

## 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;<br>USHORT Fid;<br><br>BOOLEAN IsFsctl;<br><br>UCHAR IsFlags; | NT device or file system control code<br>Handle for io or fs control. Unless *bit0* of *IsFlags* is set.<br>Indicates whether the command is a device control (FALSE) or a file system control (TRUE).<br>*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<br>Setup[0]<br>DataCount<br>Data[ DataCount ] | 1<br>Length of information returned by io or fs control<br>Length of information returned by io or fs control<br>The results of the io or fs control |

## 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;<br>USHORT Reserved;<br>ULONG SecurityInformation; | FID of target<br>MBZ<br>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.

## NT_TRANSACT_NOTIFY_CHANGE

| Client Setup Words | Description |
|---|---|
| ULONG CompletionFilter;<br>USHORT Fid;<br>BOOLEAN WatchTree;<br>UCHAR Reserved; | Specifies operation to monitor (NT format)<br>Fid of directory to monitor<br>TRUE = watch all subdirectories too<br>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.

| Server Response | Description |
|---|---|
| ParameterCount<br>Parameters[ ParameterCount ] | # of bytes of change data<br>FILE_NOTIFY_INFORMATION structures |

The response contains FILE_NOTIFY_INFORMATION structures, as defined in ntioapi.h. 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.

### NT_TRANSACT_QUERY_SECURITY_DESC

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

| Client Parameter Block | Description |
|---|---|
| USHORT Fid;<br>USHORT Reserved;<br>ULONG SecurityInformation; | FID of target<br>MBZ<br>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.

## NT_CANCEL: Cancel request

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

| Client Request | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | No words are sent (== 0)<br>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.

## FIND_CLOSE2: Close Search

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

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

| Server Response | Description |
|---|---|
| UCHAR WordCount;<br>USHORT ByteCount; | Count of parameter words = 0<br>Count of data bytes = 0 |

## 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
```

## 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:<br>• invalid rename command<br>• write to fid open for read only<br>• read on fid open for write only<br>• 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:<br>• resource other than disk space exhausted (e.g. TIDs)<br>• first SMB command was not negotiate<br>• multiple negotiates attempted<br>• 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. |