Microsoft Networks

SMB FILE SHARING PROTOCOL EXTENSIONS

SMB File Sharing Protocol Extensions Version 2.0

Document Version 3.3

November 7, 1988

Microsoft Corporation

1. INTRODUCTION

This document defines the extensions to the OpenNET/Microsoft Networks File Sharing Protocol (Intel PN 136329-001) (sometimes referred to as the "core" protocol) that are required to support Operating Systems richer in function than MS-DOS 3.x. The primary goal of these extensions is to allow fully transparent access to remote files for OS/2 systems using the Microsoft OS/2 LAN Manager. However, they are not intended to be specific to OS/2. It is anticipated that other Operating Systems will have many similar requirements and that they will use the same services and protocols to meet them.

This extension, when combined with the core protocol, allows all file oriented OS/2 functions to be performed on remote files using LANMAN 1.0.

The extended protocol defined in this document is selected by the dialect string "LANMAN1.0" in the core protocol negotiate request.

Acronyms used include:

VC - Virtual Circuit. A transport level connection (sometimes called a session) between two networked machines (nodes).

TID - Tree Identifier. A token representing an instance of authenticated use of a network resource (often a shared subdirectory tree structure).

UID - User Identifier. A token representing an authenticated user of a network resource.

PID - Process Identifier. A number which uniquely identifies a process on a node.

MID - Multiplex Identifier. A number which uniquely identifies a protocol request and response within a process.

FID - File Identifier. A number which identifies an instance of an open file (sometimes called file handle).

T.B.D.- To Be Defined. Further detail will be provided at a later time.

MBZ - Must Be Zero. All reserved fields must be set to zero by the consumer.

2. MESSAGE FORMAT

All messages sent while using the extended protocol (both the core messages used and the additional messages defined in this document) will have the following format.

```
 BYTE    smb_idf[4];    /* contains 0xFF,'SMB' */
 BYTE    smb_com;       /* command code */
 BYTE    smb_rcls;      /* error class */
 BYTE    smb_reh;       /* reserved for future */
WORD     smb_err;       /* error code */
 BYTE    smb_flg;       /* flags */
WORD     smb_res[7];    /* reserved for future */
WORD     smb_tid;       /* authenticated resource identifier */
WORD     smb_pid;       /* caller's process id */
WORD     smb_uid;       /* unauthenticated user id */
WORD     smb_mid;       /* multiplex id */
 BYTE    smb_wct;       /* count of 16-bit words that follow */
WORD     smb_vwv[];     /* variable number of 16-bit words */
WORD     smb_bcc;       /* count of bytes that follow */
 BYTE    smb_buf[];     /* variable number of bytes */
```

The structure defined from smb_idf through smb_wct is the fixed portion of the SMB structure sometimes referred to as the SMB header. Following the header there is a variable number of words (defined

by smb_wct) and following that is smb_bcc which defines an additional variable number of bytes.

> A BYTE is 8 bits.
> A WORD is two BYTEs.
> The BYTEs within a WORD are ordered such that the low BYTE precedes the high BYTE.
> A DWORD is two WORDs.
> The WORDs within a DWORD are ordered such that the low WORD precedes the high WORD.

smb_com: - command code.

smb_rcls: - error class (see below).

smb_ret: - error returned (see below).

smb_tid: - Used by the server to identify a resource (e.g., a disk sub-tree). (see below)

smb_pid: - caller's process id. Generated by the consumer (redirector) to uniquely identify a process within the consumer's system. A response message will always contain the same value in smb_pid (and smb_mid) as in the corresponding request message.

smb_mid: - this field is used for multiplexing multiple messages on a single Virtual Circuit (VC) normally when multiple requests are from the same process. The PID (in smb_pid) and the MID (in smb_mid) uniquely identify a request and are used by the consumer to correlate incoming responses to previously sent requests.

3. NOTES:

1. smb_flg can have the following values:

bit0 - When set (returned) from the server in the Negotiate response protocol, this bit indicates that the server supports the "sub dialect" consisting of the LockandRead and WriteandUnlock protocols defined later in this document.

bit1 - When on (on a protocol request being sent to the server), the consumer guarantees that there is a receive buffer posted such that a "Send.No.Ack" can be used by the server to respond to the consumer's request. The LANMAN 1.0 Redirector for OS/2 will not set this bit.

bit2 - Reserved (must be zero).

bit3 - When on, all pathnames in the protocol must be treated as caseless. When off, the pathnames are case sensitive. This allows forwarding of the protocol message on various extended VCs where caseless may not be the norm. The LANMAN 1.0 Redirector for OS/2 will always have this bit on to indicate caseless pathnames.

bit4 - When on (on the Session Setup and X protocol defined later in this document), all paths sent to the server by the consumer are already in the canonicalized format used by OS/2. This means that file/directory names are in upper case, are valid characters and backslashes are used as seperators.

bit5 - When on (on core protocols Open, Create and Make 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 protocol to indicate to the consumer that the oplock request was granted. See the discussion of "oplock" in the sections defining the "Open and X" and "Locking and X" protocols later in this document (this bit has the same function as bit 1 of smb_flags of the "Open and X" protocol).

bit6 - When on (on core protocols Open, Create and Make New), this indicates that the server should notify the consumer on any action which can modify the file (delete, setattrib, rename, etc.). If not set, the server need only notify the consumer on another open request. See the discussion of "oplock" in the sections defining the "Open and X" and "Locking and X" protocols later in this document (this bit has the same function as bit 2 of smb_flags of the "Open and X" protocol). Bit6 only has meaning if bit5 is set.

bit7 - When on, this protocol is being sent from the server in response to a consumer request. The smb_com (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. On a multiplexed VC on a node where both server and consumer are active, this bit can be used by the node's SMB delivery system to help identify whether this protocol should be routed to a waiting consumer process or to the server.

2. smb_uid is the user identifier. It is used by the LANMAN 1.0 extended protocol when the server is executing in "user level security mode" to validate access on protocols which reference symbolicly named resources (such as file open). Thus differing users accessing the same TID may be granted differing access to the resources defined by the TID based on smb_uid. The UID requested is validated by the server via the Session Set Up protocol.

   Note that -2 is reserved as an invalid UID.

4. In the LANMAN 1.0 extended protocol environment the TID represents an instance of an authenticated use. This is the result of a successful NET USE to a server using a valid netname and password (if any).

   If the server is executing in a "share level security mode", the tid is the only thing used to allow access to the shared resource. Thus if the user is able to perform a successful NET USE 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 Session Set UP protocol) and the TID is NOT associated with access control but rather merely defines the resource (such as the shared directory tree).

   In most SMB protocols, smb_tid must contain a valid TID. Exceptions include prior to getting a TID established including NEGOTIATE, TREE CONNECT, SESS_SETUPandX and TREE_CONNandX protocols. Other exceptions include QUERY_SRV_INFO some forms of the TRANSACTION protocol and ECHO. A NULL TID is defined as 0xFFFF. The server is responsible for enforcing use of a valid TID where appropriate.

5. As in the core, smb_pid uniquely identifies a consumer process. Consumers inform servers of the creation of a new process by simply introducing a new smb_pid value into the dialogue (for new processes).

   In the core protocol however, the "Process Exit" protocol was used to indicate the catastrophic termination of a process (or session). 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 Process Exit protocol was used for this occurrence to allow the server to close all files opened by that process.

In the LANMAN 1.0 extended protocol, no "Process Exit" protocol will be sent. The operating system will ensure that the "close Protocol" 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 VC and TID. There is no "birth announcement" (no "fork" protocol) sent to the server. It is up to the consumer to ensure only valid processes gain access to FIDs (and TIDs). On TREE DISCONNECT (or when the VC environment is terminated) the server may invalidate any files opened by any process within the VC environment using that TID.

6. Systems using the LANMAN 1.0 extended protocol will typically be multi-tasked and will allow multiple asynchronous input/output requests per task. Therefore a multiplex ID (smb_mid) is used (along with smb_pid) to allow multiplexing the single consumer/server VC among the consumer's multiple processes, threads and requests per thread.

   The consumer is responsible for ensuring that every request includes a value in the smb_mid field which will allow the response to be associated with the correct request (at least the smb_pid and smb_mid must uniquely identify the request/response relationship system wide).

   The server is responsible for ensuring that every response contains the same smb_mid value (and smb_pid value) as its request. The consumer may then use the smb_mid value (along with smb_pid value) for associating requests and responses and may have up to the negotiated number of requests outstanding at any time on a multiplexed file server VC.

7. The LANMAN 1.0 extended protocol enhances the semantics of the pathname.

   Two special pathname component values —— "." and ".." —— must be recognized. There may be multiple of these components in a path name. They have the standard meanings —— "." points to its own directory, ".." points to its directory's parent.

   Note that it is the server's responsibility to ensure that the ".." can not be used to gain access to files/directories above the "virtual root" as defined by the Tree Connect (TID).

8. The new LANMAN 1.0 extended protocol requests and responses are variable length (as was true in "core"). Thus additional words may be added in the smb_vwv[] area in the future as well as additional bytes added within the smb_buf[] area. Servers must be implemented such that additional fields in either of these areas will not cause the command to fail. If additional fields are encountered which are not recognized by the server's level of SMB implementation, they should be ignored. This allows for future upgrade of the protocol and eliminates the need for "reserved fields".

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

10. When LANMAN 1.0 extended protocol has been negotiated, the ERRDOS error class has been expanded to include all errors which may be generated by the OS/2 operating system. As such, the error code values defined for error class ERRDOS in this document are a subset of the possible error values. See the OS/2 operating system documentation for the complete set of possible OS/2 (ERRDOS) error codes.

These semantic changes apply to all "core" requests used by the extended protocol. Where there are additional changes, they are documented with the new requests. The server having negotiated

LANMAN 1.0 is expected to still support all core protocol requests.

The following are the core protocol requests which must still be supported in the LANMAN 1.0 extended protocol without change. See "File Sharing Protocol" Intel Part number 136329-001 for detailed explanation of each protocol request/response.

TREE CONNECT
TREE DISCONNECT
OPEN FILE
CREATE FILE
CLOSE FILE
FLUSH FILE
READ
WRITE
SEEK
CREATE DIRECTORY
DELETE DIRECTORY
DELETE FILE
RENAME FILE
GET FILE ATTRIBUTES
SET FILE ATTRIBUTES
LOCK RECORD
UNLOCK RECORD
CREATE TEMPORARY FILE (no longer used by LANMAN 1.0 Redirector)
PROCESS EXIT (no longer used by LANMAN 1.0 Redirector)
MAKE NEW FILE
CHECK PATH
GET SERVER ATTRIBUTES
NEGOTIATE PROTOCOL (additional fields in response if LANMAN 1.0 negotiated)
FILE SEARCH
CREATE PRINT FILE
CLOSE PRINT FILE
WRITE PRINT FILE
(core Message Commands are also supported)

Support of all core requests within the LANMAN 1.0 extended protocol is mandatory. However, the following core requests will no longer be generated by the OS/2 implementation of the redirector when LANMAN 1.0 extended protocol has been negotiated.

PROCESS EXIT
CREATE TEMPORARY FILE
CREATE PRINT FILE
CLOSE PRINT FILE
WRITE PRINT FILE

The only protocol format change to a core protocol service is that the response to the negotiate protocol (NEGOTIATE PROTOCOL) will contain additional fields if the LANMAN1.0 string has been selected by the server thus effectively placing the session into LANMAN 1.0 extended protocol. The additional fields returned will be documented in detail later in this document.

All other protocol requests within the LANMAN 1.0 extended protocol have a new command value from that of a similar function in core protocol. Thus the server need not constantly test the protocol version negotiated. The consumer is expected to only submit appropriate requests within the dialect negotiated.

The following are the new LANMAN 1.0 extended protocol requests, each will be defined in detail later in this document.

SESS_SETUPandX	(X is another valid protocol request e.g. TREE_CONNandX)
TREE_CONNandX	(X is another valid protocol request e.g. OPEN)
OPENandX	(X is another valid protocol request e.g. READ)
READandX	(X is another valid protocol request e.g. CLOSE)
WRITEandX	(X is another valid protocol request e.g. READ)
FIND	(matches OS/2 form of FILE SEARCH)
FIND_UNIQUE	(matches OS/2 form of FILE SEARCH)
FIND_CLOSE	(matches OS/2 form of FILE SEARCH)
READ_BLOCK_RAW	(read larger than negotiated buffer size request raw)
READ_BLOCK_MPX	(read larger than negotiated buffer size request multiplexed)
WRITE_BLOCK_RAW	(write larger than negotiated buffer size request raw)
WRITE_BLOCK_MPX	(write larger than negotiated buffer size request multiplexed)
GET_E_FILE_ATTR	(accommodate new OS/2 system call)
SET_E_FILE_ATTR	(accommodate new OS/2 system call)
LOCKINGandX	(accommodate new OS/2 system call)
COPY_FILE	(used when both source and target are remote)
MOVE_FILE	(used when both source and target are remote)
IOCTL	(pass IOCTL request on to server and retrieve results)
TRANSACTION	(allows bytes in/out associated with name)
ECHO	(echo sent data back)
WRITEandCLOSE	(write final bytes then close file)
LOCKandREAD	(Lock bytes then Read locked bytes)
WRITEandUnlock	(Write bytes then Unlock bytes)

On every new command ending in "and_X" above, the following rules apply:

- The embedded command does not repeat the SMB header information. Rather it starts at the smb_wct field.

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

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

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

- The server will implicitly use the result of the first command in the "X" command. For example the TID obtained via TCONandX would be used in the embedded OPENandX and the FID obtained in the OPENandX would be used in the embedded READ.

- If multiple (chained) requests reference a FID, the smb_fid field must contain the same FID value in each request. In other words, each 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. This simplifies the handling by the worker

process on the server node.

- The first function (command) to encounter an error will stop all further processing of embedded commands. The server will NOT back out commands that succeeded. Thus if and OPENandRead was being performed and the server was able to open the file successfully but the read encountered an error, the file would remain open. This is exactly the same as if the requests had been sent separately.

- If an error occurs while processing chained requests, the last response (of the chained responses in the buffer) will be the one which encountered the error. Other unprocessed chained requests will have been ignored when the server encountered the error and will not be represented in the chained response. Actually the last valid smb_com2 (if any) will represent the protocol on which the error occurred. If no valid smb_com2 is present, then the error occurred on the first request/response and smb_com contains the command which failed. In all cases the error class and code is returned in the smb_rcls and smb_err fields of the SMB header at the start of the response buffer.

- Each chained request and response contains the offset (from the start of the SMB header) to the next chained request/response (in the field smb_off2 in the various "and X" protocols defined later e.g. Open and X). This allows building the requests unpacked. There may be space between the end of the previous request (as defined by smb_wct and smb_bcc) 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. Transaction, Read Block MPX), the data in each response protocol 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.

4. ARCHITECTURAL MODEL

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

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

Each server makes available to the network a self-contained file structure (or other resource). The resource being shared may be a directory tree, spooled device, I/O device, named pipe, etc.. There are no storage or service dependencies on any other servers. A file (or other resource) must be entirely contained by a single server.

The LANMAN 1.0 extended (like the core) file sharing protocol requires server authentication of users before file accesses are allowed. Each server processor authenticates its own users.

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

The LANMAN 1.0 extended protocol however defines two methods which can be selected by the server for security.

A "share level security mode" server makes some directory on a disk device (or other resource) sharable

(accessible from any consumer on the network). An optional password may be required to gain access. Thus any user on the network who knows the name of the server, the name ("netname") of the resource and the password (if any) has full access to all files and directories under the shared "tree" ("full" access as defined by the access level specified for the share).

A "user level security mode" server also makes some directory on a disk device (or other resource) sharable but in addition requires the user to provide an account (user) name (and optional account (user) password) in order to gain access. The consumer also supplies the UID value it wishes to represent this user (see Session Set Up protocol definition).

Thus the server is now able to allow differing access rights depending on the validated UID (in smb_uid) on each resource. One account may have full access, another read only and perhaps another no access to differing files and directories within the shared "tree". Access implementations are server dependent and outside the scope of this document, however user level security mode allows validating access based on account-name (and password) access control lists associated with each resource.

The server is expected to be in either "user level security mode" or "share level security mode" (not in some combination). The security mode of the server is returned in the response on the negotiate command if LANMAN 1.0 extended dialect is selected by the server. This allows the consumer to easily select the appropriate protocols to be used.

The following environments exist in the LANMAN 1.0 extended file sharing protocol environment.

a) Virtual Circuit Environment. This consists of one or more VC(s) established between a consumer system and server system. Thus it is the logical session between the server node and consumer node which is implemented with the use of one or more Virtual Circuits. Each of these VC(s) may be a multiplex VC in that any number of tasks and any number of requests may be active on the VC at the same time.

   Additional VC(s) are used primarily to support optional LANMAN 1.0 extended protocols which allow rapid data movement if only one process is active on the VC. Consumers using additional VC(s) would normally have only a single request active on the VC at any time, in order to receive data directly into user space for example.

   Additional VC(s) when used are to be considered a logical extension of the first VC. A smb_tid field in a protocol received on an additional VC should be authenticated to the one established on the first VC, etc..

   All of the VCs are the same as far as the server is concerned. Support for more than one VC between the consumer and server is optional. If more than one VC is in use, the server must ensure that all responses are sent on the same VC which received the request.

   A VC is formed using transport services.

b) Resource Environment. As in the core protocol, this is represented by a Tree ID (TID). A TID uniquely identifies a resource sharing connection between a consumer and server. The resource being shared may be a directory tree, spooled device, I/O device, named pipe, etc..

   In a server executing in "share level security mode" (and also in servers which do not support these extended protocols), the TID also identifies the scope and type of accesses allowed across the connection.

   In most SMB protocols, smb_tid must contain a valid TID. Exceptions include prior to getting a

TID established including NEGOTIATE, TREE CONNECT, SESS_SETUPandX and TREE_CONNandX protocols. Other exceptions include QUERY_SRV_INFO some forms of the TRANSACTION protocol and ECHO. A NULL TID is defined as 0xFFFF. The server is responsible for enforcing use of a valid TID where appropriate.

There may be any number of resource sharing connections (TIDs) per VC set.

c) User Environment. This is represented by a User ID (UID). A UID in (smb_uid) uniquely identifies a user within a given VC environment. A server (executing in "user level security mode") uses this to identify the scope and type of access allowed this user.

c) Process Environment. This is represented by a process ID (PID). A PID uniquely identifies a consumer process (thread) within a given VC environment. From the server's point of view, a new process (new value in smb_pid) may be introduced at any time. There is no "fork" protocol and files opened by another process may be manipulated by the new process (provided the appropriate smb_tid is supplied).

d) File Environment. This is represented by a File Handle (FID). A FID identifies an open file and is unique within a given VC environment (same File Handle (FID) may be used in additional VC(s)). Note that the File Handle (FID) is logon environment wide in scope. A file may be opened and its Handle passed to another process for use without being opened by that process. The smb_tid field must contain the same value as that used when the file was opened.

If a VC environment (all VC(s)) is terminated all PIDs, TIDs and FIDs within it will be invalidated. Note - additional VC(s) may be terminated without terminating the "VC environment".

If a Resource Environment is terminated (TID invalidated via Tree Disconnect protocol) all PIDS and FIDs within it will be invalidated.

4.1. Process Management

How and when servers create and destroy processes is, of course, an implementation issue and there is no requirement that this be tied in any way to the consumer's process management.

Because a file handle may be obtained by one consumer process and passed to another (e.g. child) process for use, the server can not release resources such as locks or FIDs on process exit (Note that process exit is no longer sent).

Rather the server must wait until the lock is removed or the file closed by the consumer. Consumer implementations should close handles and free locks as soon as possible to prevent server resource problems.

When the server receives a tree disconnect protocol, all processes, locks, FIDs, etc. created on behalf of that TID (logon environment) may be freed.

If the VC aborts (VC environment is terminated) then all resources for the consuming node may be freed.

All messages, except Negotiate, include a process ID (PID) to indicate which user process initiated a request. Consumers inform servers of the creation of a new process by simply introducing a new PID into the dialogue. There is no "fork" protocol to indicate to the server any parent child process relationship. This is not needed because any process may use handles created by another process (there is no special privilege or relationship to the parent process). Thus most server implementations will not need the PID, however, the PID value in smb_pid must be returned in the response (along with smb_mid) for

the use of the consumer's "SMB delivery system".

In the LANMAN 1.0 extended protocol, no "Process Exit" protocol will be sent. The operating system will ensure that the "close Protocol" will be sent when the last process referencing a file closes it. Note that a close implicitly frees locks which may be present on the file (those locks placed on the file using the file handle being closed).

5. File Sharing Connections

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

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

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

5.1. Share Level Security Mode Server Nodes

The following examples apply to access to serving systems which do NOT have a user based permission mechanism.

a) NET SHARE

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

Examples:

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

assign password "bonzo" to all files within directory "dir1\src" and its subdirectories with the "short name" src being the name used to connect to this offer.

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

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

offer read-only access to everything (all files are within the root directory or its subdirectories) Offer read-write access to all files within the \work directory and its subdirectories.

b) NET USE

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

Examples:

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


This gains full access to the files and directories on Server1 matching the offer defined by the net-name "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).

The consumer system must remember the drive identifier supplied with the NET USE request and associate it with the TID value returned by the server. 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 consumer looks through its list of drives for that node and includes the TID associated with this drive in the smb_tid field of each request.

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

5.2. User Level Security Mode Server Nodes

Servers with user based file security (in "user level security mode") will require that the consumer present an account name and account passwd (if any) along with the requested UID value (via the Session Set Up protocol) prior to accessing resources.

When the Session Set Up request is received, the account-name is validated and the UID representing that authenticated instance of the user is validated. This UID must be included in all further requests made on behalf of the user.

The Tree Connect protocol is still used to define the directory (tree) or other resource available to the user.

The server in user level security mode uses the UID to allow differing types of access to the same resources under a given TID.

Note that a single consumer user may issue multiple Tree Connect commands in order to gain access to multiple shared resources. Multiple Session Set Up commands may also be issued in order to validate additional users. NOTE - The first release of LANMAN 1.0 will allow only one valid user at a time. A user established by a Session Set Up command may be logged off via the User Logoff and X command (after all files and other resources in use are closed) and the Session Set Up and X command used to validate another userid.

The permission-based (user level security mode) systems may execute a NET SHARE command which shares the entire system and set up name/password (or whatever) information in its user definition files in order to allow user/group based access to the shared files.

The server will return whether it is executing in user level security mode or share level security mode in the extended Negotiate response protocol (when the LANMAN 1.0 dialect has been selected). This allows the consumer to know whether the "User Logon" information is needed in the Session Set Up protocol.

A server in user level security mode (having negotiated core protocol with the consumer node) will accept the core format of the Tree Connect command and do the following:

> If the consumer's node name is defined as an account-name (and the Tree Connect passwd matches), the "user logon" will be performed using that value.
>
> If the above fails, the server may fail the request or assign a default account name (probably allowing limited access).
>
> The value in smb_uid will then be ignored and all access will be validated assuming the account name selected above.

The above allows servers in "user level security mode" to accommodate core protocol users.

5.3. Connection Protocols

The NET SHARE command generates no network messages. The server remembers the pathname's net-name and the password for later verification.

The NET USE command generates a tree connect message containing the netname and the associated password. If the server is in "user level security mode" the UID, account-name and account-password will also be supplied via the "Session Set Up" protocol. If the no Session Set Up protocol is received, the server will try the consumer node name as described above.

On receiving the Tree Connect protocol, the serving node verifies the name/password combination and returns an error code or an identifier (the TID).

The short-name is included in the Tree Connect request message and the identifier (TID) identifying the connection is returned in the smb_tid field. The meaning of this identifier (TID) is server specific; the requester must not associate any specific meaning to it.

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

The consumer must associate the identifier with the device name being redirected (specified in the NET USE) and include the appropriate identifier (TID) for all network file accesses made.

5.4. USER ADMINISTRATION

The LANMAN 1.0 extended protocol makes use of the "Server Based" method of user administration and allows for "Consumer Based" user administration.

The server based approach is similar to the consumer authentication scheme used by the core protocol. It is based on the principle that the consumer processors may or may not be trusted to authenticate users. It assumes that the LAN connects autonomous systems that are willing to make some subset of their local files available to remote users.

On some networks, there may be centralized logon servers or some other means to guarantee that a global UID is unique and valid. In these networks, the consumer will merely introduce a new UID into the SMB header and the server will know who it represents and that it is valid. This is known as a "consumer based" approach in that the consumer is responsible for validating the users.

The protocol supports both types of administration in that the server may respond with ERRbaduid which will require that the consumer send a Session Set Up command to "logon" the user or it may just accept a new UID from a "trusted" consumer node.

Each server may maintain a list of valid users (or invoke some other means of user validation). It may then verify every access by these users. It can therefore accept any and all transport connections offered.

A UID is selected by the consumer and validated by the server via the Session Set Up protocol.

This UID is used to both identify the user on all subsequent requests and prove to the server that this user has been authenticated.

The consumer must associate the UID with the user and include the appropriate identifier (UID) for all network file accesses made by that user.

From the servers point of view the user identifier (UID) is therefore NOT associated with a particular offer (shared resource identified by the TID) but the authenticated user. The UID may be used to access any shared resource which has been connected to via the Tree Connect protocol.

5.5. FILE SECURITY

The specific file security model enforced by a server in "user level security mode" is outside the scope of this document, however, the types of access a user may expect are discussed here.

As was true in core protocol, the user must be prepared to be denied access.

With LANMAN 1.0 extended protocol and the server in "user level security mode" the user should expect to be allowed differing access to different files within the same shared Tree structure. He/she may be allowed to read one file only and yet be able to write to others in the same directory.

Files may have specific access permissions for specified users or groups, and another set of permissions for all other users.

The protocol provides no way to set or modify the permissions of the files and directories on the server. It is expected that the "system administrator" of the server will have a mechanism to set and modify permissions.

When files are created on the server, the files by default take on the permissions defined by the parent directory.

5.6. FILE ATTRIBUTES AND TYPES

The LANMAN 1.0 Extended File Sharing Protocol may support additional attributes to those specified in the Core File Sharing Protocol.

The LANMAN 1.0 Extended File Sharing Protocol may support additional file types to those specified in the Core File Sharing Protocol. The following file types will be supported by the LANMAN 1.0 implementation:

named pipes
message mode named pipes
I/O devices
mail slots

NAMED PIPES

Named pipes provide a new facility which allows pipes to be named and act like full duplex virtual circuits between a pair of endpoints.

Named Pipe Features

- Allows LANMAN 1.0 pipes to be 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.
- Unmodified core consumers can access named pipes on a OS/2 server as if they were accessing a sequential file.

I/O DEVICES

The LANMAN 1.0 extended protocol allows a device to be opened for driver level I/O. This provides direct access to real time and interactive devices such as modems, scanners, etc.. Two such types of devices are defined, COM - communication devices like modems or terminals and LPT - printer devices which will be accessed directly (not spooled).

5.6.1. REPRESENTATION

On LANMAN 1.0 Extended Protocol servers the attribute field has the following format (bit0 is the least significant bit). This field matches that used by OS/2.

bit0 - read only file
bit1 - "hidden" file
bit2 - system file
bit3 - reserved
bit4 - directory
bit5 - archive file
bits6-15 - reserved (must be zero)

The LANMAN 1.0 Extended resource type field defines the additional resource types:

0 - Disk file or directory as defined in the attribute field.
1 - FIFO (named pipe)
2 - Named pipe (message mode)
3 - LPT (printer) Device
4 - COM (communication) Device

6. TIMEOUTS

The LANMAN 1.0 extended protocol provides for actions timing out on the server. Actions which may time out include:

- Opens (to I/O devices)
- Locks (on records within files)
- read/write (on I/O devices)

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.

7. QUEUEING

The LANMAN 1.0 extended protocol provides for Queuing for Opens of I/O devices and for bytes within a file to become free (unlocked). If a server implementation can not support queuing and the resource being queued for is busy, simply respond with a "busy" error just as if the queued request had timed out. The same thing could happen if queuing was supported but the queue remained full.

8. EXCEPTION HANDLING

Exception handling within the LANMAN 1.0 extended protocol environment is an extension to the core techniques. It is built upon the various environments supported by the file sharing protocol. When any environment is dissolved (in either an orderly or disorderly fashion) all contained environments are dissolved. The hierarchy of environments is summarized below:

Server Based User Authentication:

Virtual Circuit Environment
TID
PID
FID

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

The termination of a TID will cause the termination of all PIDs (or equivalent) created in support of that TID and close of all FIDs opened using the TID as the access key. The destruction of PIDs and FIDs has no affect on other environments.

The rules for VC establishment and dissolution are identical to those enforced by the core file sharing protocol with the following exceptions:

- If a server receives a VC establishment request from a consumer with which it is already conversing, it will be accepted as an "additional" VC to be used for rapid data movement protocols. The negotiate protocol will then be used to indicate if the consumer intends this VC to be the first VC or as an additional VC. If the consumer specified that this was to be the first VC, all other VCs to that consumer may be aborted.

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

- If a server gets a hard error on a VC (such as a send failure) all VCs to that consumer may be aborted.

- A server may drop the VC on last Tree Disconnect, however, we recommend that the drop be performed only after an automatic timeout time has passed or when the VC resource is needed. This will help performance in that the VC will not need to be reestablished if activity occurs again before the "timeout" time.

  Some server implementations will perform a "soft timeout" where if no tree connects (and thus no other resources) are in use on a VC, it may age until the server elects to drop it allowing other users access.

  Other implementations include a "hard timeout" in that if a VC has not been used for a (system administrator defined) amount of time, the VC will be aborted destroying all Tree Connections etc. which are still on the session.

  Others use an as needed basis where one or both of the above methods are used but only when Virtual Circuits are needed.

  A consumer is always expected to be able to reestablish in the case where the VC was dropped with no valid tree connects (and thus no other resources) on it.

  For Hard timeouts, the user may receive errors and have to reconnect (do another NET USE) in order to reestablish the session. This is similar to many "switch" timeouts on multi-user systems.

On write behind activity, a subsequent write or close of the file will return the fact that a previous write failed. Normally write behind failures are limited to hard disk errors and device out of space.

9. EXTENDED PROTOCOL

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

9.1. CORE SERVICE ENHANCEMENTS

The LANMAN 1.0 extended protocol includes functional enhancements to some core services, and these are defined in this section. Enhancements to services can only add to the service's existing semantics, i.e., additional values for parameters may be introduced and new parameters added, but parameters cannot be removed or used for a different purpose. All "core" requests must continue to function with the enhanced command. If these rules cannot be followed a new command must be defined.

9.1.1. NEGOTIATE

Request Format is unchanged in order to remain compatible with earlier versions and the core protocol.

Enhanced Response Format (returned only when LANMAN 1.0 dialect is selected):

```
    BYTE    smb_wct;        /* value = 13 */
   WORD     smb_index;      /* index identifying dialect selected */
 + WORD     smb_secmode;    /* security mode:
                            bit 0 - 1 = User level security, 0 =  Share level security.  */
                            bit 1 - 1 = encrypt passwords , 0 =  do not encrypt passwords */
 + WORD     smb_maxxmt;     /* max transmit buffer size server supports,  1k min */
 + WORD     smb_maxmux;     /* max pending multiplexed requests server supports */
 + WORD     smb_maxvcs;     /* max VCs per server/consumer session supported */
 + WORD     smb_blkmode;    /* block read/write mode support :
                            bit 0 - Read Block Raw supported (65,535 bytes max).
                            bit 1 - Write Block Raw supported (65,535 bytes max). */
+DWORD     smb_sesskey;    /* Session Key (unique token identifying session) */
 + WORD     smb_srv_time;   /* server's current time (hhhhh mmmmmm xxxxx)
                            where 'xxxxx' is in two second increments */
 + WORD     smb_srv_date;   /* server's current date (yyyyyyy mmmm ddddd) */
 + WORD     smb_srv_tzone;  /* server's current time zone */
+DWORD     smb_rsvd;       /* reserved */
   WORD     smb_bcc;        /* value = (size of smb_cryptkey) */
  + BYTE    smb_cryptkey[]; /* Key used for password encryption */
```

+ Additional parameters

In addition, if bit ZERO (low order bit) of smb_flg is set (returned) from the server in the Negotiate response protocol header, this indicates that the server supports the "sub dialect" consisting of the LockandRead and WriteandUnlock protocols.

Service Enhancement:

The dialect string for the LANMAN 1.0 extended protocol specified in this document is:

LANMAN1.0

Having selected LANMAN 1.0 as the dialect, the consumer needs to be informed of whether or not the server is executing in "share level security mode" or "user level security mode" in order to format appropriate Session Set Up protocols.

Whether or not Read Block Raw (where up to 65,535 bytes of data may be read directly into user space) is supported is returned.

Whether or not Write Block Raw (where up to 65,535 bytes of data may be written directly from user space) is supported is returned.

The smb_sesskey value is used to validate additional VCs added to a session (via the Session Set Up protocol).

The minimum server SMB buffer size (smb_maxxmt) is 1024 bytes (1k). This provides sufficient room for most protocols including the simple "request-response" mode of the IOCTL protocol.

Note that smb_maxxmt returned in the NEGOTIATE response is 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 smb_bufsize returned to the server from the consumer via the 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 size (as indicated in smb_maxxmt on NEGOTIATE) were 4k and the consumer's buffer size were only 2k (as indicated in smb_bufsize on SESSION SETUP and X), The consumer could send up to 4k (standard) write requests but must only request up to 2k for (standard) read requests. The max transaction response from the server would also be 2k.

LANMAN 1.0 will use a cryptkey of 8 bytes.

The date is in the following format:

bits:

```
1 1 1 1   1 1
5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
y y y y   y y y m   m m m d   d d d d
```

where:

y - bit of year 0-119 (1980-2099)
m - bit of month 1-12
d - bit of day 1-31

The time is in the following format:

bits:

```
1 1 1 1   1 1
5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
h h h h   h m m m   m m m x   x x x x
```

where:

h - bit of hour (0-23)
m - bit of minute (0-59)
x - bit of 2 second increment

Negotiate may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2. ADDITIONAL SERVICES

This LANMAN 1.0 Extension Set includes all functions considered necessary to provide transparency for all essential or frequently used file access functions. OS/2 requirements have been used as a guide for selecting the services included here; when other common operating systems are considered in more detail it may prove necessary to expand this set.

The protocols in this section are presented in alphabetical order.

9.2.1. COPY

Request Format:

```
  BYTE    smb_wct;          /* value = 3 */
 WORD     smb_tid2;         /* second (destination) path tid */
 WORD     smb_ofun;         /* what to do if destination file exists */
 WORD     smb_flags;        /* flags to control copy operations:
                            bit 0 - destination must be a file.
                            bit 1 - destination must be a directory.
                            bit 2 - copy destination mode: 0 = binary, 1 = ASCII.
                            bit 3 - copy source mode: 0 = binary, 1 = ASCII.
                            bit 4 - verify all writes. */
 WORD     smb_bcc;          /* minimum value = 2 */
  BYTE    smb_path[];       /* pathname of source file */
  BYTE    smb_new_path[];   /* pathname of destination file */
```

Response Format:

```
  BYTE    smb_wct;       /* value = 1 */
 WORD     smb_cct;       /* number of files copied */
 WORD     smb_bcc;       /* minimum value = 0 */
  BYTE    smb_errfile[]; /* pathname of file where error occured - ASCIIZ */
```

Service:

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

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

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

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

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

Wild Cards are not permitted in the destination path.

If the destination is a file and the source contains Wild Cards, the destination file will either be truncated or appended to at the start of the operation (depending on bits in smb_ofun) . Subsiquent files will then be appended to the file.

smb_ofun bit field 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.

Copy may generate the following errors.

Error Class ERRDOS

ERRbadfile
ERRbadpath
ERRfileexists
ERRnoaccess
ERRnofiles
ERRbadshare
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
ERRnosupport
ERRaccess
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.2. ECHO

Request Format:

```
  BYTE    smb_wct;      /* value = 01 */
 WORD     smb_reverb;   /* number of times to echo data back */
 WORD     smb_bcc;      /* minimum value = 4 */
  BYTE    smb_data[];   /* data to echo back */
```

Response Format (one for each echo requested):

```
  BYTE    smb_wct;      /* value = 1 */
 WORD     smb_seq;      /* sequence number of this echo */
 WORD     smb_bcc;      /* minimum value = 4 */
  BYTE    smb_data[];   /* echo data */
```

Service:

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

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

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

There is no need for a valid TID field in smb_tid for the success of this protocol (a null TID is defined as 0xFFFF). However, if a TID is present, then the server must check the TID for validity because the consumer may be sending this protocol to see if the TID is still valid. ERRSRV error class and ERRinvnid error code should be set in the protocol response if the TID is invalid.

The flow for the ECHO protocol is:

```
consumer ------------> ECHO request (data) > ------------> server
consumer <------------< ECHO response 1 (data) <------------ server
consumer <------------< ECHO response 2 (data) <------------ server
.                                  .                            .
consumer <------------< ECHO response n (data) <------------ server
```

ECHO may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror
ERRnosupport
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.3. FIND

Request Format: (same as core Search Protocol)

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

Response Format: (same as core Search Protocol)

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

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

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

The resume_key has the following format:

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

Service:

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

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

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

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

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

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

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

The resume_key field must be null (length = 0) on the initial ("Find First") find request. Subsequent find requests intended to continue a search must contain the resume_key field extracted from the last directory entry of the previous response. The resume_key field is self-contained, for on calls containing a resume_key neither the attribute or pathname fields will be valid in the request. A find request will terminate when either the requested maximum number of entries that match the named file are found, or the end of directory is reached without the maximum number of matches being found. A response containing no entries indicates that no matching entries were found between the starting point of the search and the end of directory.

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

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

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

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

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

If a Find requests more data than can be placed in a message of the max-xmit-size for the TID specified,

the server will return only the number of entries which will fit.

The number of entries returned will be the minimum of:

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

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

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

The date is in the following format:

bits:

```
1 1 1 1   1 1
5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
y y y y   y y y m   m m m d   d d d d
```

where:

y - bit of year 0-119 (1980-2099)
m - bit of month 1-12
d - bit of day 1-31

The time is in the following format:

bits:

```
1 1 1 1   1 1
5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
h h h h   h m m m   m m m x   x x x x
```

where:

h - bit of hour (0-23)
m - bit of minute (0-59)
x - bit of 2 second increment

Find may generate the following errors.

Error Class ERRDOS

ERRnofiles
ERRbadpath
ERRnoaccess
ERRbadaccess
ERRbadshare
<implementation specific>

Error Class ERRSRV

ERRerror
ERRaccess
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.4. FIND CLOSE

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

```
 BYTE    smb_wct;            /* value = 2 */
WORD     smb_count;          /* max number of entries to find */
WORD     smb_attr;           /* search attribute */
WORD     smb_bcc;            /* minimum value = 5 */
 BYTE    smb_ident1;         /* ASCII  (04) */
 BYTE    smb_pathname[];     /* null (may contain only null) */
 BYTE    smb_ident2;         /* Variable Block (05) */
WORD     smb_keylen;         /* resume (close) key length (may not be zero) */
 BYTE    smb_resumekey[*];   /* "Find Close" key (* = value of smb_keylen) */
```

Response Format: (same format as core Search Protocol)

```
 BYTE    smb_wct;        /* value = 1 */
WORD     smb_reserved;   /* reserved */
WORD     smb_bcc;        /* value = 3 */
 BYTE    smb_ident;      /* Variable Block (05) */
WORD     smb_datalen;    /* data length (value = 0) */
```

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

```
BYTE    sr_res;         /* reserved:
                             bit    7 - reserved for consumer use
                             bit  5,6 - reserved for system use (must be preserved)
                             bits 0-4 - rsvd for server (must be preserved by consumer) */
BYTE    sr_name[11];    /* pathname sought.  Format:
                             1-8 character file name, left justified
                             0-3 character extension, left justified (in last 3 chars) */
BYTE    sr_findid[1];   /* uniquely identifies find through find close */
BYTE    sr_server[4];   /* available for server use (must be non-zero) */
BYTE    sr_res[4];      /* reserved for consumer use */
```

Service:

The Find Close protocol closes the association between a Find id returned (in the resume_key) by the Find protocol and the directory search.

Whereas the First Find protocol logically opens the directory, subsequent find protocols presenting a resume_key further "read" the directory, the Find Close protocol "closes" the directory allowing the server to free any resources held in support of the directory search.

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

Although only the find id portion the resume key should be required to identify the search being terminated, the entire resume_key as returned in the previous Find (either a "Find First" or "Find Next") is sent to the server in this protocol.

Find Close may generate the following errors.

Error Class ERRDOS

ERRbadfid
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.5. FIND UNIQUE

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

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

Response Format: (same as core Search Protocol)

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

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

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

The resume_key has the following format:

```
BYTE     sr_res;          /* reserved:
                             bit    7 - reserved for consumer use
                             bit  5,6 - reserved for system use (must be preserved)
                             bits 0-4 - rsvd for server (must be preserved by consumer) */
BYTE     sr_name[11];     /* pathname sought.  Format:
                             1-8 character file name, left justified
                             0-3 character extension, left justified (in last 3 chars) */
BYTE     sr_findid[1];    /* uniquely identifies find through find close */
BYTE     sr_server[4];    /* available for server use (must be non-zero) */
BYTE     sr_res[4];       /* reserved for consumer use */
```

Service:

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

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

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

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

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

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

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

The resume_key field must be null (length = 0).

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

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

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

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

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

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

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

The number of entries returned will be the minimum of:

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

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

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

The date is in the following format:

bits:

```
1 1 1 1   1 1
5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
y y y y   y y y m   m m m d   d d d d
```

where:

y - bit of year 0-119 (1980-2099)
m - bit of month 1-12
d - bit of day 1-31

The time is in the following format:

bits:

```
1 1 1 1   1 1
5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
h h h h   h m m m   m m m x   x x x x
```

where:

h - bit of hour (0-23)
m - bit of minute (0-59)
x - bit of 2 second increment

Find Unique may generate the following errors.

Error Class ERRDOS

ERRnofiles
ERRbadpath
ERRnoaccess
ERRbadaccess
ERRbadshare
<implementation specific>

Error Class ERRSRV

ERRerror
ERRaccess
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.6. GET EXPANDED FILE ATTRIBUTES

Request Format:

```
  BYTE    smb_wct;    /* value = 1 */
 WORD     smb_fid;    /* file handle */
 WORD     smb_bcc;    /* value = 0 */
```

Response Format:

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

Service Enhancement:

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

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

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

The attribute field (smb_attr) has the following format (bit0 is the least significant bit). This field matches that used by OS/2.

bit0 - read only file
bit1 - "hidden" file
bit2 - system file
bit3 - reserved
bit4 - directory
bit5 - archive file
bits6-15 - reserved (must be zero)

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

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

The dates are in the following format:
bits:


```
          1 1 1 1   1 1
          5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
          y y y y   y y y m   m m m d   d d d d
```

where:

y - bit of year 0-119 (1980-2099)
m - bit of month 1-12
d - bit of day 1-31

The times are in the following format:

bits:

```
          1 1 1 1   1 1
          5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
          h h h h   h m m m   m m m x   x x x x
```

where:

h - bit of hour (0-23)
m - bit of minute (0-59)
x - bit of 2 second increment

Get Expanded file attributes may generate the following errors.

Error Class ERRDOS

ERRbadfile
ERRbadfid
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.7. IOCTL

Primary Request Format:

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

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

```
BYTE    smb_wct;    /* value = 0 */
WORD    smb_bcc;    /* value = 0 */
```

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

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


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

```
  BYTE    smb_wct;        /* value = 8 */
 WORD     smb_tprcnt;     /* total number of parameter bytes being returned */
 WORD     smb_tdrcnt;     /* total number of data bytes being returned */
 WORD     smb_prcnt;      /* number of param bytes being returned this buffer */
 WORD     smb_proff;      /* offset (from start of SMB hdr) to parameter bytes */
 WORD     smb_prdisp;     /* byte displacement for these parameter bytes */
 WORD     smb_drcnt;      /* number of data bytes being returned this buffer */
 WORD     smb_droff;      /* offset (from start of SMB hdr) to data bytes */
 WORD     smb_drdisp;     /* byte displacement for these data bytes */
 WORD     smb_bcc;        /* total bytes (including pad bytes) following */
  BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
  BYTE    smb_param[*];   /* param bytes (* = value of smb_prcnt) */
  BYTE    smb_pad1[];     /* (optional) to pad to word or dword boundary */
  BYTE    smb_data[*];    /* data bytes (* = value of smb_drcnt) */
```

Service:

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

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

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

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

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

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

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

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

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

The Consumer knows the maximum amount of data bytes and parameter bytes which the server may return (from smb_mprcnt and smb_mdrcnt of the request). Thus it initializes its bytes expected variables

to these values. The Server then informs the consumer of the actual amounts being returned via each "packet" (buffer) of the response (smb_tprcnt and smb_tdrcnt).

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

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

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

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

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

Thus the flow is:

1 The consumer sends the first (primary) request which identifies the total bytes (both parameters and data) which are expected to be sent and contains as many of those bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (both parameters and data) the server will need to return on IOCTL completion. If all the bytes fit in the single buffer, skip to step 4.

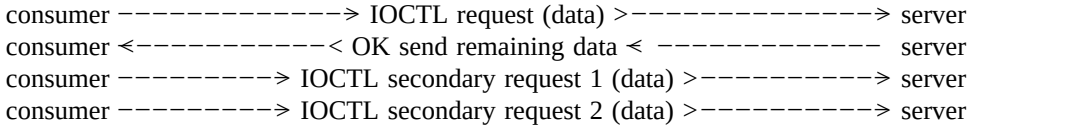
2 The server responds with a single interim response meaning "ok, send the remainder of the bytes".

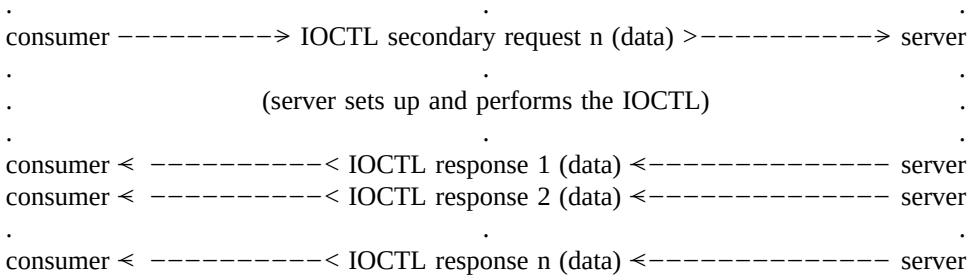
3 The consumer then sends another buffer full of bytes to the server. On each iteration of this secondary request, smb_tpscnt and/or smb_tdscnt could be reduced. This step is repeated until all bytes have been delivered to the server (total of all smb_pscnt equals smallest smb_tpscnt and total of all smb_dscnt equals smallest smb_tdscnt).

4 The Server sets up and performs the IOCTL with the information provided.

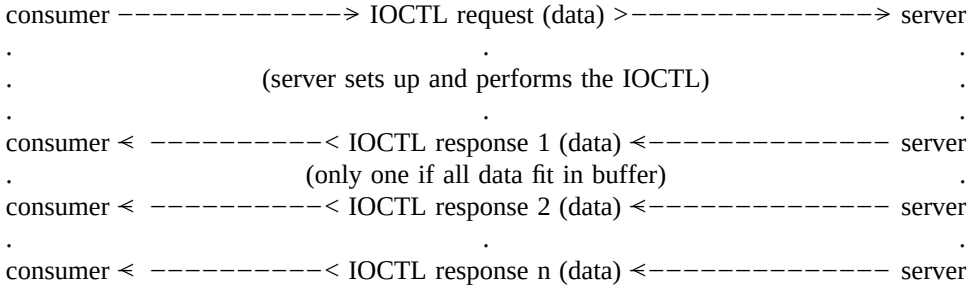
5 Upon completion of the IOCTL, the server sends back (up to) the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all result bytes have been returned. On each iteration of this response, smb_tprcnt and/or smb_tdrcnt could be reduced. This step is repeated until all bytes have been delivered to the consumer (total of all smb_prcnt equals smallest smb_tprcnt and total of all smb_drcnt equals smallest smb_tdrcnt).

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



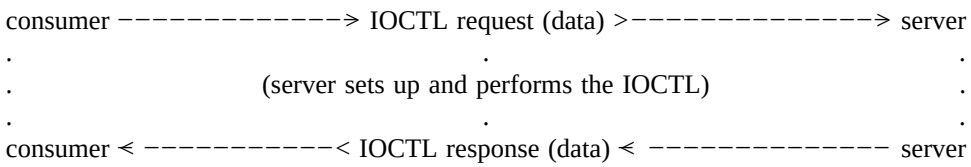


The flow for the IOCTL protocol when the request parameters and data does all fit in a single buffer is:



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

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



IOCTL may generate the following errors.

Error Class ERRDOS

ERRbadfile
ERRbadfid
ERRbaddata
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.8. LOCK and READ

Request Format (same as core READ):

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

Response Format (same as core READ):

```
  BYTE    smb_wct;      /* value = 5 */
  WORD    smb_count;    /* number of locked bytes read */
  WORD    smb_rsvd[4];  /* reserved (to match size of write request) */
  WORD    smb_bcc;      /* minimum value = 4 */
  BYTE    smb_ident1;   /* value = DATA_BLOCK */
  WORD    smb_size;     /* length of data returned */
  BYTE    smb_data[];   /* data */
```

Service:

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

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

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

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

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

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

LockandRead may generate the following errors:

Error Class ERRDOS:

ERRnoaccess
ERRbadfid
ERRlock
ERRbadaccess

Error Class ERRSRV:

ERRerror
ERRinvdevice
ERRinvnid
<implementation specific>

Error Class ERRHRD:

<implementation specific>

9.2.9. LOCKING and X

Request Format:

```
BYTE    smb_wct;          /* value = 8 */
BYTE    smb_com2;         /* secondary (X) command, 0xFF = none */
BYTE    smb_reh2;         /* reserved (must be zero) */
WORD    smb_off2;         /* offset (from SMB hdr start) to next cmd (@smb_wct) */
WORD    smb_fid;          /* file handle */
WORD    smb_locktype;     /* locking mode:
                             bit 0 - 0 = lock out all access, 1 = read ok while locked
                             bit 1 - 1 = single user total file unlock
DWORD   smb_timeout;      /* number of milliseconds to attempt each lock */
WORD    smb_unlocknum;    /* number of unlock range structures following */
WORD    smb_locknum;      /* number of lock range structures following */
WORD    smb_bcc;          /* total bytes following */
struct  smb_unlkrng[*];   /* unlock range structures (* = smb_unlocknum) */
struct  smb_lockrng[*];   /* lock range structures (* = smb_locknum) */
```

Unlock Range Structure (smb_unlkrng) Format:

```
WORD    smb_lpid;         /* pid of process "owning" the lock */
DWORD   smb_unlockoff;    /* file offset to bytes to be unlocked */
DWORD   smb_unlocklen;    /* number of bytes to be unlocked */
```

Lock Range Structure (smb_lockrng) Format:

```
WORD    smb_lpid;         /* pid of process "owning" the lock */
DWORD   smb_lockoff;      /* file offset to bytes to be locked */
DWORD   smb_locklen;      /* number of bytes to be locked */
```

Response Format:

```
BYTE    smb_wct;          /* value = 2 */
BYTE    smb_com2;         /* secondary (X) command, 0xFF = none */
BYTE    smb_res2;         /* reserved (pad to word) */
WORD    smb_off2;         /* offset (from SMB hdr start) to next cmd (@smb_wct) */
WORD    smb_bcc;          /* value = 0 */
```

Service Enhancement:

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

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

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

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

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

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

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

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

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

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

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

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

READ
READ and X

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

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

When another user attempts to Open (or otherwise modify if "opbatch" was requested) the file which a consumer has oplocked, the server will "hold off" the 2nd attempt and notify the consumer via a LockingX protocol (with bit one of smb_locktype set) that the "oplock" is being broken. The consumer is expected to then flush any dirty buffers, submit any file locks (LockingX protocol can be used for this) and respond to the server with either a LockingX protocol (with bit one of smb_locktype set) or with a

close protocol if the file is no longer in use. Note that because a close being sent to the server and break oplock notification from the server could cross on the wire, if the consumer gets an oplock notification on a file which it does not have open, that notification should be ignored. Once the "oplock" has been broken, the consumer must no longer do any form of data or lock caching. The "oplock" is never reenstated while the file is open. If the file is still open once the consumer has been notified, the 2nd opener does not get the file "oplocked" along with the open. If the file is closed by the consumer which had it open, the server is again free to grant the new opener the oplock.

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

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

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

Locking and X may generate the following errors.

Error Class ERRDOS

ERRbadfile
ERRbadfid
ERRlock
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvdevice
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.10. MOVE

Request Format:

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

Response Format:

```
  BYTE    smb_wct;          /* value = 1 */
 WORD     smb_cct;          /* number of files moved */
 WORD     smb_bcc;          /* minimum value = 0 */
  BYTE    smb_errfile[];    /* pathname of file where error occured - ASCIIZ */
```

Service:

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

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

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

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

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

Wild Cards are not permitted in the destination path.

smb_ofun bit field mapping:

bits:

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

where:

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

0 - Fail.
1 - reserved.
2 - Truncate file.

r - reserved (must be zero).

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

Move may generate the following errors.

Error Class ERRDOS

ERRbadfile
ERRbadpath
ERRfileexists
ERRnoaccess
ERRnofiles
ERRbadshare
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
ERRnosupport
ERRaccess
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.11. OPEN and X

Request Format:

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

Response Format:

```
   BYTE     smb_wct;          /* value = 15 */
   BYTE     smb_com2;         /* secondary (X) command, 0xFF = none */
   BYTE     smb_res2;         /* reserved (pad to word) */
   WORD     smb_off2;         /* offset (from SMB hdr start) to next cmd (@smb_wct) */
   WORD     smb_fid;          /* file handle */
 + WORD     smb_attribute;    /* attributes of file or device */
+DWORD     smb_time;         /* last modification time */
+DWORD     smb_size;         /* current file size */
 + WORD     smb_access;       /* access permissions actually allowed */
 + WORD     smb_type;         /* file type */
 + WORD     smb_state;        /* state of IPC device (e.g. pipe) */
   WORD     smb_action;       /* action taken */
  DWORD     smb_fileid;       /* server unique file id */
   WORD     smb_rsvd;         /* reserved */
   WORD     smb_bcc;          /* value = 0 */
```

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

Service Enhancement:

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

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

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

READ
READ and X
IOCTL

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

smb_mode bit field mapping:
bits:

```
1111 11
5432 1098 7654 3210
rWrr rrrr rSSS rAAA
```

where:

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

r - reserved (must be zero).

SSS - Sharing mode
0 -- Compatibility mode (as in core open protocol)
1 -- Deny read/write/execute (exclusive).
2 -- Deny write.
3 -- Deny read/execute.
4 -- Deny none.

AAA - 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 (as in core open protocol)

The "open function" field specifies the action to be taken depending on whether or not the file exists. This word has the following format:

smb_ofun bit field mapping:
bits:

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

where:

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

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

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

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

smb_action bit field mapping:
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.

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

bit0 - read only file
bit1 - "hidden" file
bit2 - system file
bit3 - reserved
bit4 - directory
bit5 - archive file
bits6-15 - reserved (must be zero)

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

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

0 - Disk file or directory as defined in the attribute field.
1 - FIFO (named pipe)
2 - Named pipe (message mode)
3 - LPT (printer) Device
4 - COM (communication) Device

IPC State Bits (smb_state)

```
          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 (N/A)

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

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

When another user attempts to Open (or otherwise modify if "opbatch" was requested) the file which a consumer has oplocked, the server will "hold off" the 2nd attempt and notify the consumer via a LockingX protocol (with bit one of smb_locktype set) that the "oplock" is being broken. The consumer is expected to then flush any dirty buffers, submit any file locks (LockingX protocol can be used for this) and respond to the server with either a LockingX protocol (with bit one of smb_locktype set) or with a close protocol if the file is no longer in use. Note that because a close being sent to the server and break oplock notification from the server could cross on the wire, if the consumer gets an oplock notification on a file which it does not have open, that notification should be ignored. Once the "oplock" has been broken, the consumer must no longer do any form of data or lock caching. The

"oplock" is never reenstated while the file is open. If the file is still open once the consumer has been notified, the 2nd opener does not get the file "oplocked" along with the open. If the file is closed by the consumer which had it open, the server is again free to grant the new opener the oplock.

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

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

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

The following errors may be generated by Open and X.

Error Class ERRDOS

ERRbadfile
ERRnofids
ERRnoaccess
ERRshare
ERRbadaccess
<implementation specific>

Error Class ERRSRV

ERRerror
ERRaccess
ERRinvnid
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.12. READ and X

Request Format:

```
   BYTE    smb_wct;        /* value = 10 */
   BYTE    smb_com2;       /* secondary (X) command, 0xFF = none */
   BYTE    smb_reh2;       /* reserved (must be zero) */
  WORD     smb_off2;       /* offset (from SMB hdr start) to next cmd (@smb_wct) */
  WORD     smb_fid;        /* file handle */
 DWORD     smb_offset;     /* offset in file to begin read */
  WORD     smb_maxcnt;     /* max number of bytes to return */
  WORD     smb_mincnt;     /* min number of bytes to return */
 DWORD     smb_timeout;    /* number of milliseconds to wait for completion */
  WORD     smb_countleft;  /* bytes remaining to satisfy user's request */
  WORD     smb_bcc;        /* value = 0 */
```

Response Format:

```
   BYTE    smb_wct;        /* value = 12 */
   BYTE    smb_com2;       /* secondary (X) command, 0xFF = none */
   BYTE    smb_res2;       /* reserved (pad to word) */
  WORD     smb_off2;       /* offset (from SMB hdr start) to next cmd (@smb_wct) */
  WORD     smb_remaining;  /* bytes remaining to be read (pipes/devices only) */
 DWORD     smb_rsvd;       /* reserved */
  WORD     smb_dsize;      /* number of data bytes (minimum value = 0) */
  WORD     smb_doff;       /* offset (from start of SMB hdr) to data bytes */
  WORD     smb_rsvd;       /* reserved (These last 5 words are reserved in */
 DWORD     smb_rsvd;       /* reserved  order to make the ReadandX response */
 DWORD     smb_rsvd;       /* reserved  the same size as the WriteandX request) */
  WORD     smb_bcc;        /* total bytes (including pad bytes) following */
   BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
   BYTE    smb_data[*];    /* data bytes (* = value of smb_dsize) */
```

Service:

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

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

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

CLOSE
CLOSE and DISCONNECT

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

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

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

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

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

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

Read and X may generate the following errors:

Error Class ERRDOS:

ERRnoaccess
ERRbadfid
ERRlock
ERRbadaccess

Error Class ERRSRV:

ERRerror
ERRinvnid
ERRtimeout
<implementation specific>

Error Class ERRHRD:

<implementation specific>

9.2.13. READ BLOCK MULTIPLEXED

Request Format:

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

Response Format (one or more of these are returned):

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

Service:

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

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

When this protocol is used, other requests may be active on the multiplexed VC. The server will respond with the one or more response protocol message as defined above until the requested data amount has been returned. Each response contains the smb_pid and smb_mid of the Read Block Multiplexed request, the file offset and data length defined in the Read response protocol (including the SMB header). This allows the consumer's message delivery (multiplexing) system to deliver the response to the appropriate thread.

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

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

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

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

size (as specified in smb_maxxmt of the NEGOTIATE response protocol).

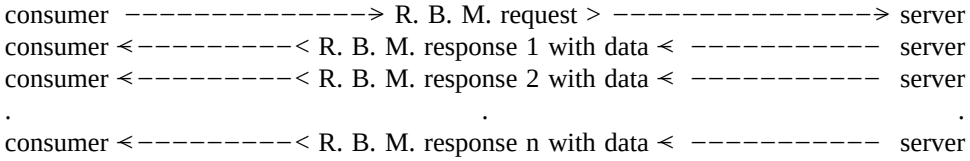
As is true in the core read protocol, (while reading a "standard blocked disk file"), the total number of bytes returned may be be less than the number requested only if a read specifies bytes beyond the current file size. In this case only the bytes that exist are returned. A read completely beyond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

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

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

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

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



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

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

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

Error Class ERRDOS

ERRnoaccess
ERRbadfid
ERRlock
ERRbadaccess
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
ERRnoresource
ERRusestd
ERRtimeout
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.14. READ BLOCK RAW

Request Format:

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

Response is the raw data (one send).

Service:

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

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

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

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

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

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

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

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

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

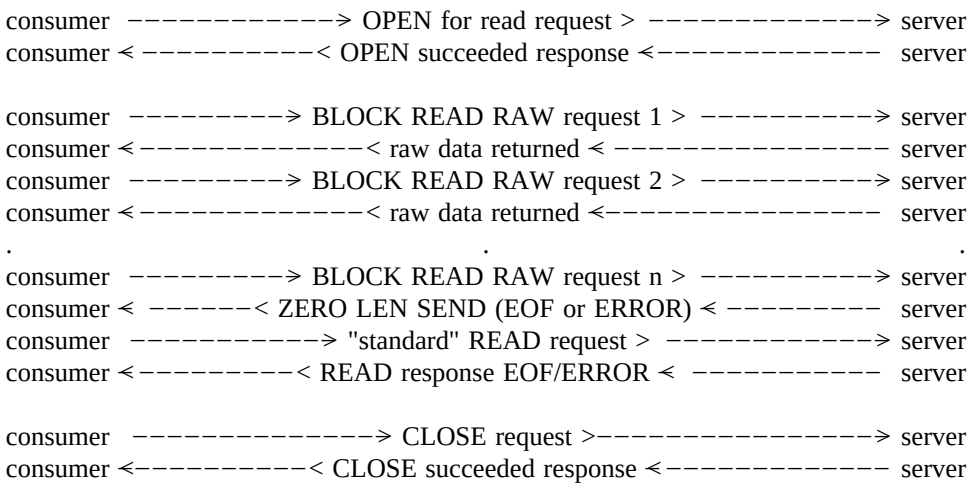
If an error should occur at the consumer end, all bytes must be received and thrown away. There is no

need to inform the server of the error.

Support of this protocol is optional.

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

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



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

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

9.2.15. SESSION SETUP and X

Request Format:

```
BYTE    smb_wct;          /* value = 10 */
BYTE    smb_com2;         /* secondary (X) command, 0xFF = none */
BYTE    smb_reh2;         /* reserved (must be zero) */
WORD    smb_off2;         /* offset (from SMB hdr start) to next cmd (@smb_wct) */
WORD    smb_bufsize;      /* the consumers max buffer size */
WORD    smb_mpxmax;       /* actual maximum multiplexed pending requests */
WORD    smb_vc_num;       /* 0 = first (only), non zero - additional VC number */
DWORD   smb_sesskey;      /* Session Key (valid only if smb_vc_num != 0) */
WORD    smb_apasslen;     /* size of account password (smb_apasswd) */
DWORD   smb_rsvd;         /* reserved */
WORD    smb_bcc;          /* minimum value = 0 */
BYTE    smb_apasswd[*];   /* account password (* = smb_apasslen value) */
BYTE    smb_aname[];      /* account name string */
```

Response Format:

```
BYTE    smb_wct;          /* value = 3 */
BYTE    smb_com2;         /* secondary (X) command, 0xFF = none */
BYTE    smb_res2;         /* reserved (pad to word) */
WORD    smb_off2;         /* offset (from SMB hdr start) to next cmd (@smb_wct) */
WORD    smb_action;       /* request mode:
                             bit0 = Logged in successfully - BUT as GUEST */
WORD    smb_bcc;          /* value = 0 */
```

Service definition:

This protocol 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". Here, the userid value (smb_uid of the SMB header) is set by the consumer to be the userid desired for the account (user) name supplied in smb_aname and validated by the account (user) password supplied in smb_apasswd (if a passwd is required).

Because the account password may be encrypted, it is a variable length field with the length specified by smb_apasslen (if password encryption is not being used, smb_apasswd should be a null terminated ASCII string with smb_apasslen set to the string size including the null).

The server validates the name and password supplied and if valid, it registers the UID (in smb_uid) on this session as representing the specified account (user) name. The smb_uid field will then be used to validate access on subsequent protocol requests. The protocol requests where permission checks are required are those which refer to a symbolically named resource such as OPEN, RENAME, DELETE, TRANSACT, etc..

In networks with untrusted consumers, the value of the UID (smb_uid) is relative to a session so it is possible to have the same UID value represent two different users on two different sessions at the server. The server must map the session id and the value in smb_uid) to a unique account.

In networks with trusted consumers, it is allowed to pass only the UID (no account name/password). On these systems, UIDs are unique and validated user IDs.

Multiple session setup commands may be sent to register additional users on this session. If the server receives an additional Session Setup protocol, only the smb_uid, smb_aname and smb_apasswd fields need contain valid values (the server will ignore the other fields). The error "ERRtoomanyuids" will be

returned if the server can not support the additional UID requested.

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

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 smb_bufsize is the maximum message size which the consumer can receive. Thus although the server may support 16k buffers (as returned in the negotiate response), if the consumer only has 4k buffers, the value of smb_bufsize here would be 4096.

The minimum SMB buffer size (smb_maxxmt) is 1024 bytes (1k). This provides sufficient room for most protocols including the simple "request-response" mode of the IOCTL protocol.

Note that smb_maxxmt returned in the NEGOTIATE response is 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 smb_bufsize returned to the server from the consumer via the 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 size (as indicated in smb_maxxmt on NEGOTIATE) were 4k and the consumer's buffer size were only 2k (as indicated in smb_bufsize on SESSION SETUP and X), The consumer could send up to 4k (standard) write requests but must only request up to 2k for (standard) read requests. The max transaction response from the server would also be 2k.

The field, smb_mpxmax informs the server of the maximum number of requests which the consumer will have outstanding on a given VC simultaneously.

The values for smb_bufsize, smb_mpxmax, and smb_vc_num must be less than or equal to the maximum values supported by the server as returned in the negotiate response.

The smb_vc_num field specifies whether the consumer wants this to be the first VC or an additional VC.

If the server gets a Session Set UP request with vc_number of 0 and other VCs are still connected (to that consumer), they will be aborted thus freeing any resources held. This condition could occur if the consumer was rebooted and reconnected to the server before the transport level had informed the server of the previous VC termination.

Because most transports do not make it easy to use differing size buffers on the same VC, the buffer size is negotiated at negotiate time rather than at Tree Connect time. The new expanded Tree Connect protocol (TREE_CONNECTandX) no longer negotiates buffer sizes. If buffer sizes are needed at Tree Connect time, the core version of TREE CONNECT may still be used.

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

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

TREE CONNECT and X
OPEN FILE
OPEN and X
CREATE FILE
MAKE NEW FILE
CREATE DIRECTORY
DELETE FILE
DELETE DIRECTORY
FILE SEARCH
FIND
FIND UNIQUE
COPY FILE
RENAME FILE
MOVE FILE
CHECK PATH
GET FILE ATTRIBUTES
SET FILE ATTRIBUTES
GET SERVER ATTRIBUTES
QUERY SERVER INFO
CREATE PRINT FILE
GET PRINT QUEUE
TRANSACTION

Session Setup and X may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror
ERRbadpw
ERRinvnetname
ERRtoomanyuids
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.16. SET EXPANDED FILE ATTRIBUTES

Request Format:

```
  BYTE    smb_wct;      /* value = 7 */
 WORD     smb_fid;      /* file handle */
 WORD     smb_cdate;    /* date of creation */
 WORD     smb_ctime;    /* time of creation */
 WORD     smb_adate;    /* date of last access */
 WORD     smb_atime;    /* time of last access */
 WORD     smb_mdate;    /* date of last modification */
 WORD     smb_mtime;    /* time of last modification */
 WORD     smb_bcc;      /* minimum value = 0 */
  BYTE    smb_rsvd[];   /* reserved */
```

Response Format:

```
  BYTE    smb_wct;    /* value = 0 */
 WORD     smb_bcc;    /* value = 0 */
```

Service Enhancement:

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

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

The dates are in the following format:

bits:

```
        1 1 1 1   1 1
        5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
        y y y y   y y y m   m m m d   d d d d
```

where:

y - bit of year 0-119 (1980-2099)
m - bit of month 1-12
d - bit of day 1-31

The times are in the following format:

bits:

```
        1 1 1 1   1 1
        5 4 3 2   1 0 9 8   7 6 5 4   3 2 1 0
        h h h h   h m m m   m m m x   x x x x
```

where:

h - bit of hour (0-23)
m - bit of minute (0-59)
x - bit of 2 second increment

Set Expanded File Attributes may generate the following errors.

Error Class ERRDOS

ERRbadfile
ERRbadfid
ERRnoaccess
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
ERRaccess
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.17. TRANSACTION

Primary Request Format:

```
BYTE    smb_wct;        /* value = (14 + value of smb_suwcnt) */
WORD    smb_tpscnt;     /* total number of parameter bytes being sent */
WORD    smb_tdscnt;     /* total number of data bytes being sent */
WORD    smb_mprcnt;     /* max number of parameter bytes to return */
WORD    smb_mdrcnt;     /* max number of data bytes to return */
BYTE    smb_msrcnt;     /* max number of setup words to return */
BYTE    smb_rsvd;       /* reserved (pad above to word) */
WORD    smb_flags;      /* additional information:
                           bit 0 - if set, also disconnect TID in smb_tid
                           bit 1 - if set, transaction is one way (no final response) */
DWORD   smb_timeout;    /* number of milliseconds to wait for completion */
WORD    smb_rsvd1;      /* reserved */
WORD    smb_pscnt;      /* number of parameter bytes being sent this buffer */
WORD    smb_psoff;      /* offset (from start of SMB hdr) to parameter bytes */
WORD    smb_dscnt;      /* number of data bytes being sent this buffer */
WORD    smb_dsoff;      /* offset (from start of SMB hdr) to data bytes */
BYTE    smb_suwcnt;     /* set up word count */
BYTE    smb_rsvd2;      /* reserved (pad above to word) */
WORD    smb_setup[*];   /* variable number of set up words (* = smb_suwcnt) */
WORD    smb_bcc;        /* total bytes (including pad bytes) following */
BYTE    smb_name[];     /* name of transaction */
BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
BYTE    smb_param[*];   /* param bytes (* = value of smb_pscnt) */
BYTE    smb_pad1[];     /* (optional) to pad to word or dword boundary */
BYTE    smb_data[*];    /* data bytes (* = value of smb_dscnt) */
```

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

```
BYTE    smb_wct;    /* value = 0 */
WORD    smb_bcc;    /* value = 0 */
```

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

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


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

```
  BYTE    smb_wct;        /* value = (10 + value of smb_suwcnt) */
 WORD     smb_tprcnt;     /* total number of parameter bytes being returned */
 WORD     smb_tdrcnt;     /* total number of data bytes being returned */
 WORD     smb_rsvd;       /* reserved */
 WORD     smb_prcnt;      /* number of parameter bytes being returned this buf */
 WORD     smb_proff;      /* offset (from start of SMB hdr) to parameter bytes */
 WORD     smb_prdisp;     /* byte displacement for these parameter bytes */
 WORD     smb_drcnt;      /* number of data bytes being returned this buffer */
 WORD     smb_droff;      /* offset (from start of SMB hdr) to data bytes */
 WORD     smb_drdisp;     /* byte displacement for these data bytes */
  BYTE    smb_suwcnt;     /* set up return word count */
  BYTE    smb_rsvd1;      /* reserved (pad above to word) */
 WORD     smb_setup[*];   /* variable # of set up return words (* = smb_suwcnt) */
 WORD     smb_bcc;        /* total bytes (including pad bytes) following */
  BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
  BYTE    smb_param[*];   /* param bytes (* = value of smb_prcnt) */
  BYTE    smb_pad1[];     /* (optional) to pad to word or dword boundary */
  BYTE    smb_data[*];    /* data bytes (* = value of smb_drcnt) */
```

Service:

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

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

The Transaction "set up information" and/or parameters define functions specific to a particular resource on a particular server. Therefore the functions supported are not defined by the protocol, but by consumer/server implementations. The protocol simply provides a means of delivering them and retrieving the results.

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

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

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

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

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

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

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

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

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

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

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

Thus the flow is:

1 The consumer sends the first (primary) request which identifies the total bytes (both parameters and data) which are expected to be sent and contains the set up words and as many of the parameter and data bytes bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters and data) the server is to return on TRANSACTION completion. If all the bytes fit in the single buffer, skip to step 4.

2 The server responds with a single interim response meaning "ok, send the remainder of the bytes" or (if error response) terminate the transaction.

3 The consumer then sends another buffer full of bytes to the server. On each iteration of this secondary request, smb_tpscnt and/or smb_tdscnt could be reduced. This step is repeated until all bytes have been delivered to the server (total of all smb_pscnt equals smallest smb_tpscnt and total of all smb_dscnt equals smallest smb_tdscnt).

4 The Server sets up and performs the TRANSACTION with the information provided.

5 Upon completion of the IOCTL, the server sends back (up to) the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all result bytes have been returned. On each iteration of this response, smb_tprcnt and/or smb_tdrcnt could be reduced. This step is repeated until all bytes have been delivered to the consumer (total of all smb_prcnt equals smallest smb_tprcnt and total of all smb_drcnt equals smallest smb_tdrcnt).

Thus the flow is:

1 The consumer sends the first (primary) request which identifies the total bytes (parameters and data) which are to be sent, contains the set up words and as many of the parameter and data bytes as will fit in a negotiated size buffer. This request also identifies the maximum number of bytes (setup, parameters and data) the server is to return on TRANSACTION completion. The parameter bytes are immediately followed by the data bytes (the length fields identify the break point). If all the bytes fit in the single buffer, skip to step 4.

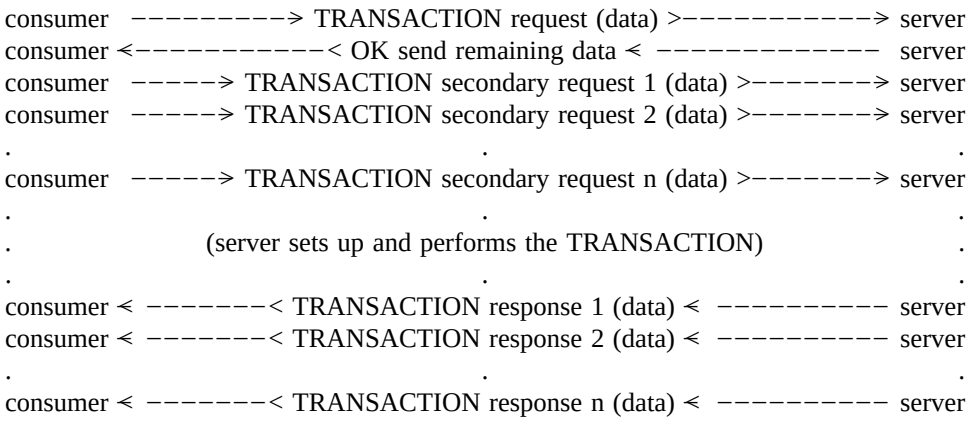
2 The server responds with a single interim response meaning "ok, send the remainder of the bytes" or (if error response) terminate the transaction.

3 The consumer then sends another buffer full of bytes to the server. This step is repeated until all bytes have been delivered to the server.

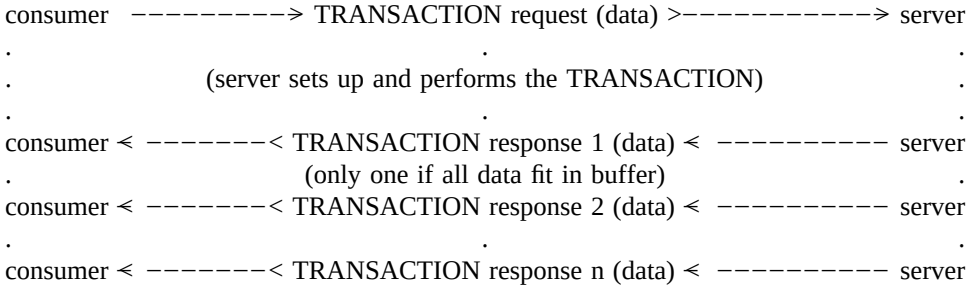
4 The Server sets up and performs the TRANSACTION with the information provided.

5 Upon completion of the TRANSACTION, the server sends back up to the the number of parameter and data bytes requested (or as many as will fit in the negotiated buffer size). This step is repeated until all bytes requested have been returned. On each iteration of this response, smb_rprcnt and smb_rdrcnt are reduced by the number of matching bytes returned in the previous response. The parameter count (smb_rprcnt) is expected to go to zero first because the parameters are sent before the data. The data count (smb_rdrcnt) may then continue to be counted down. Fewer than the requested number of bytes may be returned.

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



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



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

Transaction may generate the following errors:

Error Class ERRDOS:

ERRnoaccess
ERRbadaccess

Error Class ERRSRV:

ERRerror
ERRinvnid
ERRaccess
ERRmoredata
<implementation specific>

Error Class ERRHRD:

<implementation specific>

9.2.17.1. Defined Transaction Protocols

This section specifies some of the defined usages of the Transaction protocol. Each of the usages here utilize the basic (and flexible) transaction protocol format. This is NOT meant to be an exhaustive list.

Note that the simplest form of a Transaction performs a single send of the Transaction request and (optionally) gets back a single response. Thus if the entire Transaction message fits within the size limits for a Datagram (defined by NetBios to be 512 bytes max) and reliable delivery of the information is not required, the Transaction protocol may be sent/received as a datagram.

9.2.17.1.1. Mail Slot Transaction protocol

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

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

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

Primary Request Format:

```
BYTE    smb_wct;        /* value = 17 */
WORD    smb_tpscnt;     /* value = 0 total number of param bytes being sent */
WORD    smb_tdscnt;     /* total size of data to write to mail slot (if any) */
WORD    smb_mprcnt;     /* value = 2 one word return code expected */
WORD    smb_mdrcnt;     /* value = 0 size of data read from mail slot (N/A) */
BYTE    smb_msrcnt;     /* value = 0 max number of setup words to return (N/A)*/
BYTE    smb_rsvd;       /* reserved (pad above to word) */
WORD    smb_flags;      /* additional information:
                           bit 0 - if set, also disconnect TID in smb_tid
                           bit 1 - if set, no response is required */
DWORD   smb_timeout;    /* (user defined) number of milliseconds to wait */
WORD    smb_rsvd1;      /* reserved */
WORD    smb_pscnt;      /* value = 0 no param bytes being sent this buffer */
WORD    smb_psoff;      /* value = 0 no parameter bytes */
WORD    smb_dscnt;      /* number of data bytes being sent this buffer */
WORD    smb_dsoff;      /* offset (from start of SMB hdr) to data bytes */
BYTE    smb_suwcnt;     /* value = 3 */
BYTE    smb_rsvd2;      /* reserved (pad above to word) */
WORD    smb_setup1;     /* (op code) value = 1 - Write Mail slot */
WORD    smb_setup2;     /* (priority) priority of transaction */
WORD    smb_setup3;     /* (class) 1 = reliable, 2 = unreliable */
WORD    smb_bcc;        /* total bytes (including pad bytes) following */
BYTE    smb_name[];     /* "\MAILSLOT\<name>0"  */
BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
BYTE    smb_data[*];    /* data to be written to Mail Slot (if any)
                           (* = value of smb_dscnt) */
```


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

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


9.2.17.1.2. Announce (and request Announce) Mail Slot Transaction protocol

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

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

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

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

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

Announce Mail Slot Transaction format:

```
  BYTE    smb_wct;      /* value = 17 */
  WORD    smb_tpscnt;   /* value = 0 no param bytes being sent */
  WORD    smb_tdscnt;   /* size of announce or req_announce */
  WORD    smb_mprcnt;   /* value = 0 no param bytes to return (N/A)*/
  WORD    smb_mdrcnt;   /* value = 0 no data to read from mail slot (N/A)*/
  BYTE    smb_msrcnt;   /* value = 0 no setup words to return (N/A)*/
  BYTE    smb_rsvd1;    /* reserved (pad above to word) */
  WORD    smb_flags;    /* additional information:
                           bit 0 - 0 N/A
                           bit 1 - set, no response is required (value = 1) */
 DWORD    smb_timeout;  /* (user defined) number of milliseconds to wait */
  WORD    smb_rsvd;     /* reserved */
  WORD    smb_pscnt;    /* value = 0 no parameter bytes being sent this buf */
  WORD    smb_psoff;    /* value = 0 no parameter bytes */
  WORD    smb_dscnt;    /* size of announce or req_announce */
  WORD    smb_dsoff;    /* offset (from start of SMB hdr) to data bytes */
  BYTE    smb_suwcnt;   /* value = 3 */
  BYTE    smb_rsvd2;    /* reserved (pad above to word) */
  WORD    smb_setup1;   /* (op code) value = 1 - Write Mail slot */
  WORD    smb_setup2;   /* (priority) priority of transaction */
  WORD    smb_setup3;   /* (class) 2 = unreliable */
  WORD    smb_bcc;      /* total bytes (including pad bytes) following */
  BYTE    smb_name[];   /* "\MAILSLOT\LANMAN" (null terminated string) */
  BYTE    smb_pad[];    /* (optional) to pad to word or dword boundary */
  BYTE    smb_data[*];  /* (announce or req_announce structure)
                           (* = value of smb_dscnt) */
```

Announcement Structure Format:

```
  WORD    op_code;      /* value = 1 (announce) */
 DWORD    services;     /* may both be set
                           bit 0 - work station
                           bit 1 - server */
```



```
BYTE    vers_major;     /* major version number of node software */
BYTE    vers_minor;     /* minor version number of node software */
WORD    periodicity;    /* announcement cycle in seconds */
BYTE    node_name[];    /* computer name of this node */
BYTE    comment[];      /* descriptive remark */
```

Request Announce Structure Format:

```
WORD    op_code;        /* value = 2 (request announce) */
BYTE    node_name[];    /* computer name of this node */
```


9.2.17.1.3. Named pipe Transaction protocol

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

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

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

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

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

```
  BYTE    smb_wct;        /* value = 16 */
  WORD    smb_tpscnt;     /* total number of parameter bytes being sent */
  WORD    smb_tdscnt;     /* size of data to be written to pipe (if any) */
  WORD    smb_mprcnt;     /* max number of parameter bytes to return */
  WORD    smb_mdrcnt;     /* size of data to be read from pipe (if any) */
  BYTE    smb_msrcnt;     /* value = 0 max number of setup words to return */
  BYTE    smb_rsvd;       /* reserved (pad above to word) */
  WORD    smb_flags;      /* additional information:
                             bit 0 - if set, also disconnect TID in smb_tid
                             bit 1 - not set, response is required */
 DWORD    smb_timeout;    /* (user defined) number of milliseconds to wait */
  WORD    smb_rsvd1;      /* reserved */
  WORD    smb_pscnt;      /* number of parameter bytes being sent this buffer */
  WORD    smb_psoff;      /* offset (from start of SMB hdr) to parameter bytes */
  WORD    smb_dscnt;      /* number of data bytes being sent this buffer */
  WORD    smb_dsoff;      /* offset (from start of SMB hdr) to data bytes */
  BYTE    smb_suwcnt;     /* value = 2 */
  BYTE    smb_rsvd2;      /* reserved (pad above to word) */
  WORD    smb_setup1;     /* function (defined below)
                              0x54 - CallNmPipe - open/write/read/close pipe
                              0x53 - WaitNmPipe - wait for pipe to be nonbusy
                              0x23 - PeekNmPipe - read but don't remove data
                              0x21 - QNmPHandState - query pipe handle modes
                              0x01 - SetNmPHandState - set pipe handle modes
                              0x22 - QNmPipeInfo - query pipe attributes
                              0x26 - TransactNmPipe - write/read operation on pipe
                              0x11 - RawReadNmPipe - read pipe in "raw" (non message mode)
                              0x31 - RawWriteNmPipe - write pipe "raw" (non message mode) */
  WORD    smb_setup2;     /* FID (handle) of pipe (if needed), or priority */
  WORD    smb_bcc;        /* total bytes (including pad bytes) following */
  BYTE    smb_name[];     /* "\PIPE\<name>0"  */
  BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
  BYTE    smb_param[*];   /* param bytes (* = value of smb_prcnt) */
  BYTE    smb_pad1[];     /* (optional) to pad to word or dword boundary */
  BYTE    smb_data[*];    /* data bytes (* = value of smb_drcnt) */
```


9.2.17.1.3. CallNmPipe

This protocol is used to implement DosCallNmPipe remotely.

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

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

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

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

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

9.2.17.1.3. WaitNmPipe

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

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

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

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

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

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

9.2.17.1.3. PeekNmPipe

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

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

PeekNmPipe acts like Read except as follows:

0 The bytes read are not removed from the pipe.

0 The peek may return only part of a message (that part currently in the pipe), even if the size of the peek would accommodate the whole message.

0 PeekNmPipe never blocks, regardless of the blocking mode.

0 Additional information about the status of the pipe and remaining data are returned. The caller can use this, for example, to determine whether the peek returned all of the current message or whether the pipe is at EOF (pipe is at EOF when there are no bytes left in the pipe and Status is Closing or Disconnected).

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

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

The following defines the format of the parameter words.

WORD bytes remaining in the pipe

WORD bytes remaining in current message

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

9.2.17.1.3. QNmPHandState

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

Purpose: Return pipe-specific state information.

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

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

Pipe Handle State Bits

```
        5  4  3  2  1  0  9  8  7  6  5  4  3  2  1  0
        B  E  *  *  T  T  R  R  |- - -  I c o u n t  - -|
```

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 (N/A)

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

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

9.2.17.1.3. SetNmPHandState

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

Purpose: Set pipe-specific handle states.

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

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

Pipe Handle State Bits

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

where:

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

9.2.17.1.3. QNmPipeInfo

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

Purpose: Returns information about a pipe

The request form of this protocol should set smb_pscnt to 2 (one word parameter, the information level). The pipe handle should be in smb_setup2. smb_drcnt should be set to the size specified by the user in which to receive the pipe information. smb_dscnt should be set to 0 and smb_prcnt set to 0.

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:

LEVEL 1:

WORD actual size of buffer for outgoing (server) I/O

WORD actual size of buffer for incoming (consumer) I/O

BYTE Maximum allowed number of instances

BYTE Current number of instances

BYTE Length of pipe name (including the null)

ASCIZ Name of pipe (NOT including \\NodeName - \\NodeName is prepended to this string by the consumer before passing back to the user)

LEVEL 2:

(TBD)

9.2.17.1.3. TransactNmPipe

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

Purpose: Combine a read and write operation on a named pipe.

TransactNmPipe performs a write followed by a read on a message pipe.

It provides an optimum way to implement transaction-oriented dialogs. TransactNmPipe 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, smb_setup2 should contain the pipe handle.

The state of blocking/nonblocking has no effect on this protocol (TransactNmPipe does not return until a message has been read into the response protocol). If smb_drcnt is too small to contain the response message, ERRmoredata will be returned.

9.2.17.1.3. RawReadNmPipe

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

Purpose: Read a named pipe without removing record information.

RawReadNmPipe reads bytes directly from a pipe, regardless of whether it is a message or byte pipe. For a byte pipe, this is exactly like 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).

The request form of this protocol should set smb_pscnt to 0. The pipe handle being "read raw" should be set in smb_setup2. smb_dscnt should be set to 0 (not writing data to pipe). smb_prcnt should be set to 0 and smb_drcnt set to the number of bytes to "read raw".

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

9.2.17.1.3. RawWriteNmPipe

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

Purpose: Write a named pipe without adding record information.

RawWriteNmPipe 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 form of this protocol should set smb_pscnt to 0. The pipe handle being "written raw" should be set in smb_setup2. smb_dscnt should be set to the total amount "writing raw" to the pipe. smb_prcnt should be set to 2 and smb_drcnt set 0.

The response contains no data and one parameter word. If smb_err is 0, the one parameter word indicates the number of the requested bytes that have been "written raw" to the specified pipe.

9.2.18. TREE CONNECT and X

Request Format:

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

+ Additional parameters (compared with core TREE CONNECT protocol)

Response Format:

```
   BYTE    smb_wct;          /* value = 2 */
 + BYTE    smb_com2;         /* secondary (X) command, 0xFF = none */
 + BYTE    smb_res2;         /* reserved (pad to word) */
+ WORD     smb_off2;         /* offset (from SMB hdr start) to next cmd (@smb_wct) */
  WORD     smb_bcc;          /* min value = 3 */
 + BYTE    smb_service[];    /* service type connected to (string) */
```

+ Additional parameters (compared with core TREE CONNECT protocol)

Service Enhancement:

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

The service name in the request (smb_dev) may now include:

- o A: - for file service
- o LPT1: - for a spooled output (DOS standard LPT or COM) service
- o COMM - for direct access communication device service or
  direct access printer device service
- o IPC - for inter-process communication services (named pipes, etc.)
- o ????? - "Wild card" indicating that the consumer does not yet know
  the type of service shared with the given netname and would
  like the service type returned in the smb_service string of the response.

The smb_service string returned should be one of the above service names.

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

If the tree disconnect fails, the error should be ignored.

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

OPEN FILE
OPEN and X
CREATE FILE
MAKE NEW FILE
CREATE DIRECTORY
DELETE FILE
DELETE DIRECTORY
FILE SEARCH
FIND
FIND UNIQUE
COPY FILE
RENAME FILE
MOVE FILE
CHECK PATH
GET FILE ATTRIBUTES
SET FILE ATTRIBUTES
GET SERVER ATTRIBUTES
QUERY SERVER INFO
CREATE PRINT FILE
GET PRINT QUEUE
TRANSACTION

Tree Connect and X may generate the following errors.

Error Class ERRDOS

<implementation specific>

Error Class ERRSRV

ERRerror
ERRbadpw
ERRinvnetname
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.19. WRITE and CLOSE

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

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

Response Format (same as core WRITE):

```
   BYTE    smb_wct;      /* value = 1 */
   WORD    smb_count;    /* number of bytes written */
   WORD    smb_bcc;      /* value = 0 */
```

Service:

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

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

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

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

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

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

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

Write and Close may generate the following errors:

Error Class ERRDOS:

ERRnoaccess
ERRbadfid
ERRlock
ERRbadfiletype
ERRbadaccess

Error Class ERRSRV:

ERRerror
ERRinvnid
<implementation specific>

Error Class ERRHRD:

<implementation specific>

9.2.20. WRITE and Unlock

Request Format (same as core WRITE):

```
  BYTE    smb_wct;      /* value = 5 */
  WORD    smb_fid;      /* file handle */
  WORD    smb_count;    /* number of bytes to write and then unlock */
 DWORD    smb_offset;   /* offset in file to unlock and begin write */
  WORD    smb_remcnt;   /* number of bytes remaining to be written */
  WORD    smb_bcc;      /* minimum value = 3 */
  BYTE    smb_ident1;   /* value = DATA_BLOCK */
  WORD    smb_size;     /* length of data being written */
  BYTE    smb_data[];   /* data */
```

Response Format (same as core WRITE):

```
  BYTE    smb_wct;      /* value = 1 */
  WORD    smb_count;    /* number of bytes written */
  WORD    smb_bcc;      /* value = 0 */
```

Service:

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

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

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

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

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

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

Write and Unlock may generate the following errors:

Error Class ERRDOS:

ERRnoaccess
ERRbadfid
ERRlock
ERRbadaccess

Error Class ERRSRV:

ERRerror
ERRinvdevice
ERRinvnid
<implementation specific>

Error Class ERRHRD:

<implementation specific>

9.2.21. WRITE and X

Request Format:

```
   BYTE    smb_wct;        /* value = 12 */
   BYTE    smb_com2;       /* secondary (X) command, 0xFF = none */
   BYTE    smb_reh2;       /* reserved (must be zero) */
   WORD    smb_off2;       /* offset (from SMB hdr start) to next cmd (@smb_wct) */
   WORD    smb_fid;        /* file handle */
  DWORD    smb_offset;     /* offset in file to begin write */
  DWORD    smb_timeout;    /* number of milliseconds to wait for completion */
   WORD    smb_wmode;      /* write mode:
                              bit0 - complete write before return (write through)
                              bit1 - return smb_remaining (pipes/devices only)
                              bit2 - use WriteRawNamedPipe (pipes only)
                              bit3 - this is the start of a message (pipes only) */
   WORD    smb_countleft;  /* bytes remaining to write to satisfy user's request */
   WORD    smb_rsvd;       /* reserved */
   WORD    smb_dsize;      /* number of data bytes in buffer (min value = 0) */
   WORD    smb_doff;       /* offset (from start of SMB hdr) to data bytes */
   WORD    smb_bcc;        /* total bytes (including pad bytes) following */
   BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
   BYTE    smb_data[*];    /* data bytes (* = value of smb_dsize) */
```

Response Format:

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

Service:

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

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

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

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

READ
READ and X
LOCKING and X
LOCKREAD
CLOSE
CLOSE and DISCONNECT

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

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

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

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

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

The server should "spin" here writing all date to the file/pipe/device if the write is followed by a close protocol (the "X" of WriteAndX present in the same request is a close).

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

Write and X may generate the following errors:

Error Class ERRDOS:

ERRnoaccess
ERRbadfid
ERRlock
ERRbadfiletype
ERRbadaccess

Error Class ERRSRV:

ERRerror
ERRinvnid
ERRtimeout
<implementation specific>

Error Class ERRHRD:

<implementation specific>

9.2.22. WRITE BLOCK MULTIPLEXED

Primary Request Format: (smb_com = SMBwriteBmpx)

```
   BYTE    smb_wct;        /* value = 12 */
   WORD    smb_fid;        /* file handle */
   WORD    smb_tcount;     /* total bytes (including this buf, 65,535 max) */
   WORD    smb_rsvd;       /* reserved */
  DWORD    smb_offset;     /* offset in file to begin write */
  DWORD    smb_timeout;    /* number of milliseconds to wait for completion */
   WORD    smb_wmode;      /* write mode:
                              bit0 - complete write to disk and send final result response
                              bit1 - return smb_remaining (pipes/devices only) */
  DWORD    smb_rsvd2;      /* reserved */
   WORD    smb_dsize;      /* number of data bytes this buffer (min value = 0) */
   WORD    smb_doff;       /* offset (from start of SMB hdr) to data bytes */
   WORD    smb_bcc;        /* total bytes (including pad bytes) following */
   BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
   BYTE    smb_data[*];    /* data bytes (* = value of smb_dsize) */
```

First Response Format (ok send remaining data): (smb_com = SMBwriteBmpx)

```
   BYTE    smb_wct;        /* value = 1 */
   WORD    smb_remaining;  /* bytes remaining to be read (pipes/devices only) */
   WORD    smb_bcc;        /* value = 0 */
```

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

```
   BYTE    smb_wct;        /* value = 8 */
   WORD    smb_fid;        /* file handle */
   WORD    smb_tcount;     /* total bytes to be sent this protocol */
  DWORD    smb_offset;     /* offset in file to begin write */
  DWORD    smb_rsvd;       /* reserved */
   WORD    smb_dsize;      /* number of data bytes this buffer (min value = 0) */
   WORD    smb_doff;       /* offset (from start of SMB hdr) to data bytes */
   WORD    smb_bcc;        /* total bytes (including pad bytes) following */
   BYTE    smb_pad[];      /* (optional) to pad to word or dword boundary */
   BYTE    smb_data[*];    /* data bytes (* = value of smb_dsize) */
```

Final Response Format (write through or error): (smb_com = SMBwriteC)

```
   BYTE    smb_wct;        /* value = 1 */
   WORD    smb_count;      /* total number of bytes written */
   WORD    smb_bcc;        /* value = 0 */
```

Service:

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

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

Note that the first response format will be that of the final response (SMBwriteC) in the case where the server gets an error while writeing the data sent along with the request. Thus the word parameter is

smb_count (the number of bytes which did get written) any time an error is returned. If an error occurs AFTER the first response has been sent allowing the consumer to send the remaining data, the final response should NOT be sent unless write through is set. Rather the server should return this "write behind" error on the next access to the file/pipe/device.

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

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

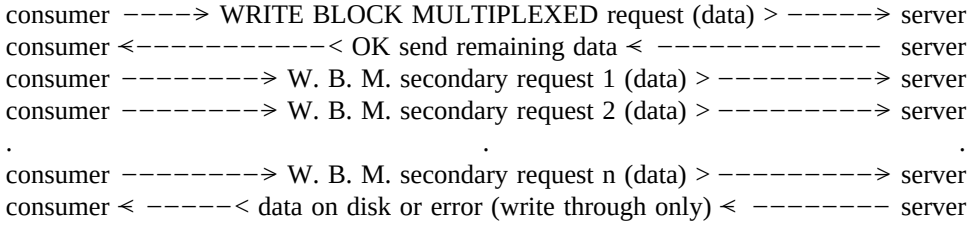
Thus, when the amount of data bytes received by the server (total of each smb_dsize) equals the total amount of data bytes expected (smallest smb_tcount received), then the server has received all the data bytes. This allows the protocol to work even if the "packets" (buffers) are received out of sequence.

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

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

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

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



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

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

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

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

Error Class ERRDOS

ERRbadfid
ERRnoaccess
ERRlock
ERRbadfiletype
ERRbadaccess
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
ERRnoresource
ERRusestd
ERRtimeout
<implementation specific>

Error Class ERRHRD

<implementation specific>

9.2.23. WRITE BLOCK RAW

Primary Request Format: (smb_com = SMBwriteBraw)

```
   BYTE    smb_wct;       /* value = 12 */
   WORD    smb_fid;       /* file handle */
   WORD    smb_tcount;    /* total bytes (including this buf, 65,535 max ) */
   WORD    smb_rsvd;      /* reserved */
  DWORD    smb_offset;    /* offset in file to begin write */
  DWORD    smb_timeout;   /* number of milliseconds to wait for completion */
   WORD    smb_wmode;     /* write mode:
                             bit0 - complete write to disk and send final result response
                             bit1 - return smb_remaining (pipes/devices only) */
  DWORD    smb_rsvd2;     /* reserved  */
   WORD    smb_dsize;     /* number of data bytes this buffer (min value = 0) */
   WORD    smb_doff;      /* offset (from start of SMB hdr) to data bytes */
   WORD    smb_bcc;       /* total bytes (including pad bytes) following */
   BYTE    smb_pad[];     /* (optional) to pad to word or dword boundary */
   BYTE    smb_data[*];   /* data bytes (* = value of smb_dsize) */
```

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

```
   BYTE    smb_wct;       /* value = 1 */
   WORD    smb_remaining; /* bytes remaining to be read (pipes/devices only) */
   WORD    smb_bcc;       /* value = 0 */
```

Secondary Request is the send of the raw data bytes:

Final Response Format (write through or error): (smb_com = SMBwriteC)

```
   BYTE    smb_wct;       /* value = 1 */
   WORD    smb_count;     /* total number of bytes written */
   WORD    smb_bcc;       /* value = 0 */
```

Service:

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

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

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

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

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

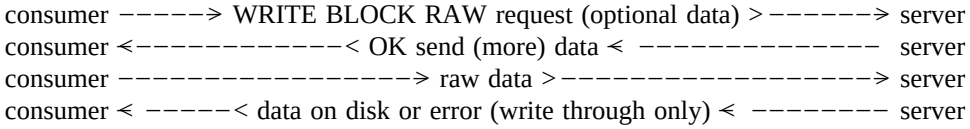
Support of this protocol is optional.

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

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

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

The flow for the Write Block Raw protocol is:



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

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

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

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

does not support this feature), a -1 value should be returned.

Write Block Raw may generate the following errors.

Error Class ERRDOS

ERRbadfid
ERRnoaccess
ERRlock
ERRbadfiletype
ERRbadaccess
<implementation specific>

Error Class ERRSRV

ERRerror
ERRinvnid
ERRnoresource
ERRtimeout
ERRusempx
ERRusestd
<implementation specific>

Error Class ERRHRD

<implementation specific>

10. DATA DEFINITIONS

10.1. COMMAND CODES

The command codes are unchanged for commands that are common with the Core File Sharing Protocol.

The following values have been assigned for the "core" protocol commands.

```
#define SMBmkdir        0x00    /* create directory */
#define SMBrmdir        0x01    /* delete directory */
#define SMBopen         0x02    /* open file */
#define SMBcreate       0x03    /* create file */
#define SMBclose        0x04    /* close file */
#define SMBflush        0x05    /* flush file */
#define SMBunlink       0x06    /* delete file */
#define SMBmv           0x07    /* rename file */
#define SMBgetatr       0x08    /* get file attributes */
#define SMBsetatr       0x09    /* set file attributes */
#define SMBread         0x0A    /* read from file */
#define SMBwrite        0x0B    /* write to file */
#define SMBlock         0x0C    /* lock byte range */
#define SMBunlock       0x0D    /* unlock byte range */
#define SMBctemp        0x0E    /* create temporary file */
#define SMBmknew        0x0F    /* make new file */
#define SMBchkpth       0x10    /* check directory path */
#define SMBexit         0x11    /* process exit */
#define SMBlseek        0x12    /* seek */
#define SMBtcon         0x70    /* tree connect */
#define SMBtdis         0x71    /* tree disconnect */
#define SMBnegprot      0x72    /* negotiate protocol */
#define SMBdskattr      0x80    /* get disk attributes */
#define SMBsearch       0x81    /* search directory */
#define SMBsplopen      0xC0    /* open print spool file */
#define SMBsplwr        0xC1    /* write to print spool file */
#define SMBsplclose     0xC2    /* close print spool file */
#define SMBsplretq      0xC3    /* return print queue */
#define SMBsends        0xD0    /* send single block message */
#define SMBsendb        0xD1    /* send broadcast message */
#define SMBfwdname      0xD2    /* forward user name */
#define SMBcancelf      0xD3    /* cancel forward */
#define SMBgetmac       0xD4    /* get machine name */
#define SMBsendstrt     0xD5    /* send start of multi-block message */
#define SMBsendend      0xD6    /* send end of multi-block message */
#define SMBsendtxt      0xD7    /* send text of multi-block message */
```


The commands added by the LANMAN 1.0 Extended File Sharing Protocol have the following command codes:

```
#define SMBlockread       0x13   /* lock then read data */
#define SMBwriteunlock    0x14   /* write then unlock data */
#define SMBreadBraw       0x1A   /* read block raw */
#define SMBreadBmpx       0x1B   /* read block multiplexed */
#define SMBreadBs         0x1C   /* read block (secondary response) */
#define SMBwriteBraw      0x1D   /* write block raw */
#define SMBwriteBmpx      0x1E   /* write block multiplexed */
#define SMBwriteBs        0x1F   /* write block (secondary request) */
#define SMBwriteC         0x20   /* write complete response */
#define SMBsetattrE       0x22   /* set file attributes expanded */
#define SMBgetattrE       0x23   /* get file attributes expanded */
#define SMBlockingX       0x24   /* lock/unlock byte ranges and X */
#define SMBtrans          0x25   /* transaction - name, bytes in/out */
#define SMBtranss         0x26   /* transaction (secondary request/response) */
#define SMBioctl          0x27   /* IOCTL */
#define SMBioctls         0x28   /* IOCTL  (secondary request/response) */
#define SMBcopy           0x29   /* copy */
#define SMBmove           0x2A   /* move */
#define SMBecho           0x2B   /* echo */
#define SMBwriteclose     0x2C   /* Write and Close */
#define SMBopenX          0x2D   /* open and X */
#define SMBreadX          0x2E   /* read and X */
#define SMBwriteX         0x2F   /* write and X */
#define SMBsesssetup      0x73   /* Session Set Up & X (including User Logon) */
#define SMBtconX          0x75   /* tree connect and X */
#define SMBffirst         0x82   /* find first */
#define SMBfunique        0x83   /* find unique */
#define SMBfclose         0x84   /* find close */
#define SMBinvalid        0xFE   /* invalid command */
```


10.2. ERROR CLASSES AND CODES

The error class and code lists in the section include all classes and codes generated by the Core File Sharing Protocol. Errors listed here are intended to provide a finer granularity of error conditions. These lists are not complete.

The following error classes may be returned by the protocol elements defined in this document.

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.
ERRXOS	0x04	Reserved for XENIX.
ERRRMX1	0xE1	Reserved for iRMX
ERRRMX2	0xE2	Reserved for iRMX
ERRRMX3	0xE3	Reserved for iRMX
ERRCMD	0xFF	Command was not in the "SMB" format.

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

SUCCESS	0	The request was successful.

The following error codes may be generated with the ERRDOS error class. The XENIX errors equivalent to each of these errors are noted at the end of the error description. NOTE - When the extended protocol (LANMAN 1.0) has been negotiated, all of the error codes below may be generated plus any of the new error codes defined for OS/2 (see OS/2 operating system documentation for complete list of OS/2 error codes). When only "core" protocol has been negotiated, the server must map additional OS/2 (or OS/2 like) errors to the errors listed below.

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

ERRbadfunc	1	Invalid function. The server OS did not recognize or could not perform a system call generated by the server, e.g., set the DIRECTORY attribute on a data file, invalid seek mode. [EINVAL]
ERRbadfile	2	File not found. The last component of a file's pathname could not be found. [ENOENT]
ERRbadpath	3	Directory invalid. A directory component in a pathname could not be found. [ENOENT]
ERRnofids	4	Too many open files. The server has no file handles (FIDs) available. [EMFILE]
ERRnoaccess	5	Access denied, the requester's context does not permit the requested function. This includes the following conditions. [EPERM] invalid rename command write to fid open for read only read on fid open for write only Attempt to delete a non-empty directory
ERRbadfid	6	Invalid file handle. The file handle specified was not recognized by the server. [EBADF]
ERRbadmcb	7	Memory control blocks destroyed. [EREMOTEIO]
ERRnomem	8	Insufficient server memory to perform the requested function. [ENOMEM]
ERRbadmem	9	Invalid memory block address. [EFAULT]
ERRbadenv	10	Invalid environment. [EREMOTEIO]
ERRbadformat	11	Invalid format. [EREMOTEIO]
ERRbadaccess	12	Invalid open mode.
ERRbaddata	13	Invalid data (generated only by IOCTL calls within the server). [E2BIG]
ERR	14	reserved
ERRbaddrive	15	Invalid drive specified. [ENXIO]
ERRremcd	16	A Delete Directory request attempted to remove the server's current directory. [EREMOTEIO]
ERRdiffdevice	17	Not same device (e.g., a cross volume rename was attempted) [EXDEV]
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. [ETXTBSY]
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. [EDEADLOCK]
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. [EEXIST]
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.

ERRerror	1	Non-specific error code. It is returned under the following conditions: resource other than disk space exhausted (e.g. TIDs) first command on VC was not negotiate multiple negotiates attempted internal server error [ENFILE]
ERRbadpw	2	Bad password - name/password pair in a Tree Connect or Session Setup are invalid.
ERRbadtype	3	reserved
ERRaccess	4	The requester does not have the necessary access rights within the specified context for the requested function. The context is defined by the TID or the UID. [EACCES]
ERRinvnid	5	The tree ID (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 file handle (FID) and pathname parameters contained an invalid combination of values.
ERRreserved	68	reserved.
ERRbadpermits	69	The access permissions specified for a file or directory are not a valid combination. The server cannot set the requested attribute.
ERRreserved	70	reserved.
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 ID on this session.
ERRusempx	250	Temp unable to support Raw, use MPX mode.
ERRusestd	251	Temp unable to support Raw, use standard read/write.
ERRcontmpx	252	Continue in MPX mode.
ERRreserved	253	reserved.
ERRreserved	254	reserved.
ERRnosupport	0xFFFF	Function not supported.

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

ERRnowrite	19	Attempt to write on write-protected diskette. [EROFS]
ERRbadunit	20	Unknown unit. [ENODEV]
ERRnotready	21	Drive not ready. [EUCLEAN]
ERRbadcmd	22	Unknown command.
ERRdata	23	Data error (CRC). [EIO]
ERRbadreq	24	Bad request structure length. [ERANGE]
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. [ETXTBSY]
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. [EDEADLOCK]
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.

