Table of Contents

RTIP 4.0

SMB Server Manual

Chapter One: Introduction

CHAPTER 1

Revised July 1, 2003

INTRODUCTION TO SMB

INTRODUCTION TO THE RTSMB SERVER

When offering SMB services, client machines will see "shares" available for browsing. Each share corresponds with a directory on the server's file system. The client will attempt to log on to the server and, if successful, will then be able to read and write from the share as if it were a local directory to their machine.

To offer such services with RTSMB, you must first initialize the server, tell it which directories you would like to make available, and then let it process events.

To initialize the server, you just tell RTSMB what ip the server should use, and the broadcast and mask ips to use. (the API call rtsmb_init)

Before anyone can connect to the server, you must tell RTSMB what directories to allow other people to access. You can set a password to allow authorized access only or you can let anyone access the share. (the API call rtsmb_share_add_tree – so called because SMB refers to shares as trees)

If you want, you can, instead of establishing per-directory passwords, set up a list of usernames and passwords to allow access to the whole server or specific shares.

Once you've established what shares you want to offer and what access you want to grant, you must repeatedly call the rtsmb_cycle functions to allow the server to process incoming requests. What this does is accept new connections, respond to requests of the server, and do any necessary bookkeeping.

INTRODUCTION TO THE SMB PROTOCOL

Basics

This package implements an SMB server. SMB stands for Server Message Block and is a protocol designed by Microsoft, IBM, and others to allow networked computers to share files.

Microsoft uses SMB to allow Windows machines to make resources such as disk trees (SMB vernacular for directories) and printers available to others on the local network.

SMB refers both to the protocol and a packet of the protocol. I will refer to the packets in lowercase (e.g. "I just sent three smb's to the SMB server! High five!").

Protocol Hierarchy

SMB needs someway of associating network names with addresses. It commonly uses NETBIOS for this, and RTSMB does the same. RTSMB uses TCP/IP for basic network communication and thus the protocols will look like this:

SMB
NETBIOS
IP

Basic Networking

There are three major ports of interest: 137, 138, and 139.

137 Name Service port (UDP)
138 Netbios Datagram Service port (UDP)
139 Session Service port (TCP)

The name service handles registering of network names and name challenges.

The netbios datagram service handles announcements of server availability.

The session service handles establishing a connection between servers.

When a server starts up, it registers its name across the name service, sends several announcements across the netbios datagram service, and waits for connections on the session service.

Every time the session service sees a connection, it opens a new port and assigns that session to it. All further inter-computer communications occurs on that TCP/IP port.

Flow of SMB communication

A client will connect to a server by initiating a session request on the servers Session Service port. Then, the server waits until the client issues a request, processes each request, and then issues a response, if needed.

Example

Client ————————->>>———————— Server
(Connection to port 139; "I want a session")

Client ————————->>>———————— Server
(New connection opened)

.
.

.

Client ————————->>>———————— Server
("open this file")

Client ————————<<<———————— Server
("success; this is file id of file")

The server never initiates a session or a command within a session. Most SMB requests take exactly one response.

To start a connection, the client sends a "negotiate" request and receives a response indicating which dialect of SMB will be spoken, as well some server-specific information. Then, the client requests a "session setup" which will log

a user in. Finally, to get any work done, that user connects to a share by sending a “tree connect” request. This will allow the user to browse the files on the share (or in the case of IPC$, let the user see what shares are available).

MODULES

The only file you need to include to have access to RTSMB is srvapi.h, but here is a description of all RTSMB source files:

Common Headers:

smb.h - Protocol information
smbobjs.h - Protocol data structures
smbconf.h - Configuration settings (see below)
smbdefs.h - Common defines needed by RTSMB

Porting Modules:

psmbfile - Access to the file system through a common API
psmbnet - Access to the network stack
psmbos - Access to OS-level items like mutexes and timers

Modules of Server Code:

srvans - Code to pack server responses into buffers
srvarg - Parses a config file for share setup
srvassrt - Provides sanity checking for smb processing
srvauth - Password and permission checking
srvcfg - Memory-resident data and configuration
srvcmds - Code to read client requests from buffers
srvfio - Abstraction layer for RTSMB->psmbfile calls
srvnbns - Netbios name service support
srvnbss - Netbios session service support
srvnet - Main networking and control flow code
srvrap - Allows enumeration of shares (IPC$)
srvrsrcs - Semaphore code
srvshrare - Share functions and data structures
srvssn - SMB server packet processing
srvtrans2 - SMB transaction packet processing
srvutil - Server specific support functions

PORTING

If you want to port RT-SMB to a different platform than RTIP/RT-Kernel, Windows, or GNU/Linux, you need to provide three files: psmbfile.c, psmbnet.c, and psmbos.c.

psmbfile.c is an abstraction layer for your file system. This can support Unicode, but does not need to. Look at windows/psmbfile.c for examples.

psmbnet.c is an abstraction layer for your network stack. This emulates unix-style sockets. Look at windows/psmbnet.c for examples.

psmbos.c is an abstraction layer for various kernel functions like mutex allocation and timing methods. Also, printing functions are stored here.

To build with your files, simply add them from the correct subdirectory to your project so that they get compiled along with the rest of the source code.

Instructions for porting using Visual C++:

An example project file is under the toplevel directory “RTSMB Server.”

If you want to make a new project configuration for your target, take all the source code (.c) files in the toplevel src directory, except for ncbc_enc.c, and add them to your project. Then, add one of each of psmbfile.c, psmbnet.c, and psmbos.c from whichever subdirectory is appropriate (or the ones you made).

RT-SMB does not require any command line defines or special link libraries. Just link the libraries you need for your net, os, and file porting files and add the following preprocessor includes (under Project->Settings and then the C/C++ tab, category Preprocessor, field “Additional include directories:”):

..\src\include,..\src\include\OPENSSL,..\src

RTIP 4.0

SMB Server Manual

Chapter Two: Configuring SMB

CHAPTER 2

All of these parameters are located in smbcfg.c.

CFG_RTSMB_MAX_THREADS
This controls how many helper threads the server can spawn at one time to increase responsiveness. Sessions are handed off to these helper threads as they come in. If set to 0, multithreading is turned off. ***The default value is 2***.

You might lower this to save on footprint. If you are experiencing bottlenecks with large I/O operations on one session, you might increase this to allow other sessions to be serviced at the same time.

CFG_RTSMB_MAX_SESSIONS
This controls how many sessions the server can be handling at one time. Sessions are made once per client machine. ***The default value is 5.***

You might increase this to allow more clients to connect to the server at once. If the server is at maximum, new session requests will be denied.

CFG_RTSMB_MAX_UIDS_PER_SESSION
This controls how many users can be logged in on one session at one time. ***The default value is 8.***

Some clients (notably Windows XP) log in many users for one connection and fail if they cannot get enough logins. Thus, lower this value cautiously.

CFG_RTSMB_MAX_FIDS_PER_SESSION
This controls how many open files a session can have at one time. ***The default value is 10.***

You might increase this if your clients are doing a lot of intensive reading and writing such that they would need to have many simultaneously open files.

CFG_RTSMB_MAX_FIDS_PER_TREE
This controls how many open files a tree can have on one session at one time. Setting this to higher than MAX_FIDS_PER_SESSION is pointless. ***The default value is MAX_FIDS_PER_SESSION.***

You might lower this if you want to restrict how many I/O operations are going on for a particular share. Lowering this would prevent any one user from monopolizing disk resources.

CFG_RTSMB_MAX_FIDS_PER_UID
This controls how many open files a user can have on one session at one time. Setting this to higher than MAX_FIDS_PER_SESSION is pointless. ***The default value is MAX_FIDS_PER_SESSION.***

You might lower this if you want to restrict how many open files any user can have at one time. Lowering this would prevent any one user from monopolizing disk resources.

CFG_RTSMB_MAX_SEARCHES_PER_UID
This controls how many simultaneous searches a user can have. It is recommended that you do not change this. ***The default value is 4.***

You might increase this if your clients are doing a lot of simultaneous searches. You might lower it if they are not and you want less footprint.

CFG_RTSMB_MAX_SHARES
This controls how many shares you want the server to be able to offer. ***The default value is 5.***

Increase this if you need to share more directories. If you don't, lower it to decrease footprint.

CFG_RTSMB_MAX_GROUPS
The maximum number of groups you want to keep track of. See auth.c for an explanation. ***The default value is 4.***

Increase this if you need more fine-grained control over your users. If you plan to always run in share mode, this can be set to 0 to save memory.

CFG_RTSMB_MAX_USERS
The maximum number of users you want to keep track of. See auth.c for an explanation. ***The default value is 8.***

Increase this if you want to allow access to many different people (who each need a username/password). If you plan to always run in share mode, this can be set to 0 to save memory.

CFG_RTSMB_SMALL_BUFFER_SIZE
This defines the size of common buffers. Each session has three of these – one for reading, one for writing, and one for scratchwork. ***The default value is 2924.***

Increase this if you want to make network traffic more efficient and can spare the increase in footprint. Decrease this (no lower than 1028!) if you want to save on footprint and can handle more network traffic.

CFG_RTSMB_BIG_BUFFER_SIZE
This defines the size of the big buffers. These are only used to support raw reading and writing. The maximum value is 65539, which you will need to use to support raw reads/writes on pre-NT protocols. ***The default value is 65539.***

If you don't need to support pre-NT clients, but want some of the efficiency of raw reads and writes, set this lower, like around 30000.

CFG_RTSMB_NUM_BIG_BUFFERS
This sets how many big buffers you want available. This controls how many simultaneous raw reads or writes the server can handle. Setting this to 0 disables raw reading/writing. ***The default value is 0.***

Increase this if you can handle the increased footprint and expect to do large reading and writing. Many clients don't always use raw reads/writes when they can, so you may not get as much mileage from this as you might hope.

CFG_RTSMB_MAX_TREES_PER_SESSION
This controls how many shares one session can have open at one time. ***The default value is 10.***

Some clients (notably the Windows NT line) try to connect to many trees at once and fail if they cannot connect. Thus, lower this value cautiously.

RTIP 4.0

SMB Server Manual

Chapter Three: Porting SMB

CHAPTER 3

RTSMB PORTING: OVERVIEW

The interface from RTSMB to the underlying operating system, network stack, and file system is provided through the following files:

psmbos.c - interface to underlying OS, including mutex semaphores, time functions, and printer functions (if printer support is desired)

psmbnet.c - interface to TCP/IP network stack; modeled on the BSD sockets interface

psmbfile.c - interface to the file system

This section describes the functions contained within these files, and how to port RTSMB to an environment.

If you want to port RT-SMB to a new platform, you need to provide the three porting files – psmbfile.c, psmbnet.c, and psmbos.c.

To build with your files, simply add them from the correct subdirectory to your project so that they get compiled along with the rest of the source code.

INSTRUCTIONS FOR PORTING USING VISUAL C++:

An example project file is under the toplevel directory "RTSMB Server."

If you want to make a new project configuration for your target, take all the source code (.c) files in the toplevel src directory, except for ncbc_enc.c, and add them to your project. Then, add one of each of psmbfile.c, psmbnet.c, and psmbos.c from whichever subdirectory is appropriate (or the ones you made).

RT-SMB does not require any command line defines or special link libraries. Just link the libraries you need for your net, OS, and file porting files and add the following preprocessor includes (under Project->Settings and then the C/C++ tab, category Preprocessor, field "Additional include directories:"):

..\src\include,..\src\include\OPENSSL,..\src

RTSMB PORTING: PERIODIC CLOCK SUPPORT (PSMBOS.C)

RTSMB requires a periodic clock for protocol-related timeouts. A single function provides the interface to this service:

unsigned long rtsmb_osport_get_msec (void)

Returns the system clock time in milliseconds.

RTSMB Porting: Mutex Semaphore Support (psmbos.c)

Note: Mutex support is only required by RTSMB in a multi-tasking environment. If RTSMB is invoked from a single thread in polled mode, mutex semaphore support is not necessary.

FUNCTIONS NECESSARY TO PROVIDE MUTEX SEMAPHORE SUPPORT FOR RTSMB:

int **rtsmb_osport_create_mutex**

(unsigned long *mutexHandle)

This routine must allocate and initialize a mutex, to the unclaimed status. It mustset the unsigned long pointed to by mutexHandle to a value that can be used as a handle to the mutex. If this routine is successful, this routine returns 0. Otherwise, it returns a negative value and the value of *mutexHandle is undefined.

void **rtsmb_osport_claim_mutex**

(unsigned long mutexHandle)

This routine takes a mutex handle returned by rtsmb_osport_create_mutex and returns nothing. If the mutex is already claimed, this routine must wait for the mutex to be released and then claim it and return.

void **rtsmb_osport_release_mutex**

(unsigned long mutexHandle)

This routine takes a mutex handle returned by rtsmb_osport_create_mutex and releases it. It returns nothing.

RTSMB PORTING: THREAD SUPPORT (PSMBOS.C)

These functions are only required if RTSMB is to be run in multi-threaded mode.

int **rtsmb_osport_create_thread**

(RTSMB_THREAD_FN fn, void *context)

This routine takes two parameters, a thread entry point function and a context pointer to pass into this entry point. It returns 0 if the thread is successfully spawned, a negative value otherwise. The entry point function takes a single value, a void pointer, and returns nothing. This routine should create a thread and start it running, starting at the given function. The context parameter should be passed into the entry point function.

void **rtsmb_osport_exit_thread** (void)

This routine should perform any kernel specific thread termination and clean up operations. It will be the last thing done in any thread created by RTSMB using rtsmb_osport_create_thread. It signals the termination of the thread from which it is called.

unsigned **int rtsmb_osport_get_thread_id** (void)

This routine returns some unique identifier for the current thread. This identifier is implementation-specific and no special meaning will be attached to it (except that a one-to-one correspondence from thread to thread id exists).

RTSMB PORTING: PRINTER SUPPORT (PSMBOS.C)

These functions are only required if printer sharing support is enabled in RTSMB.

A printer, for the purposes of RTSMB, is defined as an output-only device that is uniquely identified by a logical port number. A printer must be opened for exclusive use before data can be written to it, and closed once all data has been written. In addition, an routine is provided to initialize system printers before they are made available over a network.

FUNCTIONS IMPLEMENTED TO PROVIDE PRINTER SUPPORT

int **rtsmb_osport_printer_init** (int ioPort)

This routine is called once to initialize each printer (once per logical printer port). The only parameter is an integer identifying the port to initialize. This routine should return 0 if successful, or a negative value if the given port could not be initialized.

int **rtsmb_osport_printer_open** (int ioPort)

This routine opens a printer for writing data. The printer is identified by the ioPort paramater. This routine should return 0 if successful, or a negative value otherwise.

long **rtsmb_osport_printer_write**

(int ioPort, unsigned char *buffer, long size)

This routine will only be called on printer ports that have been initialized using **rtsmb_osport_printer_init**, and opened using **rtsmb_osport_printer_open**. It should write size bytes from the given buffer to the printer port, and return the number of bytes successfully written. Either 0 or a negative value may be returned in the event of an error.

int **rtsmb_osport_printer_close** (int ioPort)

This routine is called once all data has been written to a printer. It should free any resources used in the writing process and make the printer available to be opened again by another party. It should return 0 if successful, or a negative value on error.

RTSMB PORTING: NETWORK SUPPORT (PSMBNET.C)

The RTSMB interface to the underlying TCP/IP network stack is modeled after the BSD sockets API, but with some modifications to eliminate dependencies on platform-specific types and structures. As much as possible, the intention is to provide a very direct mapping between RTSMB network porting functions and calls in the sockets API. Referring to a document describing the sockets API, such as the RTIP manual, may therefore be helpful in porting psmbnet to a particular network stack.

FUNCTIONS THAT MUST BE PROVIDED AS AN INTERFACE TO THE NETWORK STACK

Note: all network addresses are arrays of 4 unsigned chars. All port values are integers in host byte order (port values may need therefore to be converted to network byte order within these routines).

int **rtsmb_netport_init** (void)

This function should initialize the network stack for operation. If successful, it returns 0, otherwise, it returns a negative value.

int **rtsmb_netport_accept**

(int *accepted, int socketId, unsigned char *remoteAddr, int *remotePort)

newConnection	-	pointer to a socket handle to set to the socket id of the new connection
socketId	-	the socket to accept the connection on
remoteAddr	-	(optional) array to fill with the ip address of the remote host
remotePort	-	(optional) pointer to int to set to the port of the remote host

Description:

This function should block waiting for a remote host to connect to the port/ip address to which socketId is bound. If a connection is successfully established, it must set *newConnection to the socket for the new connection (socketId continues to listen on its port/ip), and return 0.

In the event of a successful connection, remoteAddr and remotePort should also be set to the ip/port of the remote host that connected. If no connection can be established or an error occurs, the return value is negative, and the values of *newConnection, *remoteAddr, and *remotePort are undefined.

The behavior of this function is undefined in the following cases:

- socketId is not a valid, stream-type socket
- rtsmb_netport_bind was never called on socketId
- rtsmb_netport_listen was never called on socketId

See Also:

rtsmb_netport_bind, rtsmb_netport_listen, rtsmb_netport_socket_stream

Returns:

0 on success, negative on failure

int **rtsmb_netport_allow_broadcast**

(int socketId)

socketId - the socket for broadcasting

Description:

This function must be called on any socket over which broadcast messages are to be sent. If it is not called, broadcast messages are not guaranteed to work.

Returns:

0 on success, negative on failure

int **rtsmb_netport_bind**

(int socketId, unsigned char *ipAddr, int port)

socketId - the socket to bind
ipAddr - (optional) the ip address to associate with this socket
port - the port to associate with this socket

Description:

This function should be called on a socket before calling rtsmb_netport_listen/rtsmb_netport_accept (if the socket is stream-type), or rtsmb_netport_recvfrom (if the socket is datagram-type). The behavior of this function is undefined if socketId is not a valid socket handle.

If the port specified is already in use by another socket, this function must return a negative value.

See Also:

rtsmb_netport_accept, rtsmb_netport_listen, rtsmb_netport_socket_stream

Returns:

0 on success, negative on failure

int **rtsmb_netport_closesocket**

(int socketId)

socketId - the socket to close

Description:

This function should be called to release a socket and shut down any open connection it has. It is not defined whether this is a hard close.

See Also:

rtsmb_netport_accept, rtsmb_netport_connect

Returns:

0 on success, negative on failure

int **rtsmb_netport_connect**

(int socketId, unsigned char *ipAddr, int port)

socketId - the socket to connect
ipAddr - the address to connect to
port - the port to connect to

Description:

This function is called on a stream-type socket to initiate a connection to a specific ip address and port. This function can block until the connection is established. Behavior is undefined if socketId is not a valid, stream-type socket handle.

See Also:

rtsmb_netport_accept, rtsmb_netport_socket_stream, rtsmb_netport_closesocket

Returns:

0 on success, negative on failure

int **rtsmb_netport_listen**

(int socketId, int queueSize)

socketId - the socket handle
queueSize - the max number of requested connections to queue for acceptance

Description:

This function is called on a stream-type socket after rtsmb_netport_bind but before rtsmb_netport_accept to get a socket ready to accept connections from remote hosts.

See Also:

rtsmb_netport_accept, rtsmb_netport_bind, rtsmb_netport_socket_stream

Returns:

0 on success, negative on failure

long **rtsmb_netport_recv**

(int socketId, unsigned char *buffer, long size)

socketId - the socket to read from
buffer - pointer to a buffer to place received data in
size - the maximum number of bytes to read

Description:

This function is called on a connected socket to read data off that socket. It can block until up to size bytes are read or until the connection is closed, signifying that no more bytes are coming. If the connection has been terminated, it can return either 0 or a negative value.

If socketId is not a valid handle to a stream-type socket, behavior is undefined.

See Also:

rtsmb_netport_recvfrom, rtsmb_netport_send, rtsmb_netport_sendto, rtsmb_netport_select_n_for_read

Returns:

number of bytes read on success, negative on failure

long **rtsmb_netport_recvfrom**

(int socketId, unsigned char *buffer, long size, unsigned char *ipAddr, int *port)

socketId - the socket to read from
buffer - pointer to a buffer to place received data in
size - the maximum number of bytes to read
ipAddr - (optional) pointer to a 4-byte array to fill with the ip address of the sender
port - (optional) pointer to an int to set to the port of the sending socket

Description:

This function is called on a connectionless socket to read data. This function should block until data is ready, then read at most size bytes into buffer. If the datagram is smaller than size bytes, then only the number of bytes in the datagram must be read into the buffer, and the function must return.

See Also:

rtsmb_netport_recv, rtsmb_netport_send, rtsmb_netport_sendto, rtsmb_netport_select_n_for_read

Returns:

number of bytes read on success, negative on failure

---

int **rtsmb_netport_select_n_for_read**

(int *socketList, int listSize, long timeoutMsec)

socketList - an array of socket ids to select from
listSize - the number of elements in socketList
timeoutMsec - the maximum number of milliseconds to wait before timing out. negative value means wait forever.

Description:

This function must block for at most timeoutMsec milliseconds (or forever if this value is negative) waiting for data to become available for reading on at least one of the sockets listed in the socketList array.

When this function returns, it must have modified the contents of socketList to contain only those sockets which are ready for reading. The return value is the number of sockets on this modified list (i.e. the number of sockets that have data ready to be read).

See Also:

rtsmb_netport_recv, rtsmb_netport_recvfrom

Returns:

the number of sockets that have data ready to read.

---

long **rtsmb_netport_send**

(int socketId, unsigned char *buffer, long size)

socketId - the socket to send over
buffer - pointer to data to send
size - the number of bytes to send

Description:

This function is used to send data over a connected socket. The socket must be a stream-type; behavior is undefined if socketId is a datagram-type (connectionless) socket.

See Also:

rtsmb_netport_sendto, rtsmb_netport_recv, rtsmb_netport_recvfrom

Returns:

The number of bytes sent if successful, negative if an error occurred

---

long **rtsmb_netport_sendto**

(int socketId, unsigned char *buffer, long size, unsigned char *ipAddr, int port)

socketId - the socket to send over
buffer - pointer to data to send
size - the number of bytes to send
ipAddr - the IP address to send to
port - the port number to send to

Description:

Sends size bytes from buffer to the specified ip address/port. The given socket must be a connectionless (datagram-type); otherwise, behavior is undefined.

See Also:

rtsmb_netport_send, rtsmb_netport_recv, rtsmb_netport_recvfrom

Returns:

Returns the number of bytes sent if successful, negative if an error occurred

---

int **rtsmb_netport_socket_stream**

(int *socketId)

socketId - pointer to an int to set to the socket handler

Description:

Allocates, if possible, a stream (TCP, connection-based) type socket. If return value is negative, the value of *socketId is undefined.

Returns:

0 if successful, negative otherwise

---

int **rtsmb_netport_socket_datagram**

(int *socketId)

socketId - pointer to an int to set to the socket handler

Description:

Allocates, if possible, a datagram (UDP, connectionless) type socket. If return value is negative, the value of *socketId is undefined.

Returns:

0 if successful, negative otherwise

RTSMB PORTING: FILE SYSTEM SUPPORT (PSMBFILE.C)

The interface to the underlying file system is provided through a structure containing a set of pointers to routines for performing file operations. This structure is called SMBFILEAPI. The default file system interface is pointed to by the global symbol prtsmb_filesys (a pointer to an SMBFILEAPI). The following routines are required to port RTSMB to a particular file system:

---

int **rtsmb_fileport_init** (void)

This routine must do any file system specific initialization, initialize prtsmb_filesys to point to an SMBFILEAPI instance, and initialize the members of prtsmb_filesys to point at functions for accessing the underlying file system. If all of these operations are successful, this routine returns 0; otherwise it returns a negative value.

SMBFILEAPI is defined as follows:

```
typedef struct smbfileapi
{
    RTSMB_FS_OPENFN         fs_open;
    RTSMB_FS_READFN         fs_read;
    RTSMB_FS_WRITEFN        fs_write;
    RTSMB_FS_LSEEKFN        fs_lseek;
    RTSMB_FS_TRUNCATEFN     fs_truncate;
    RTSMB_FS_FLUSHFN        fs_flush;
    RTSMB_FS_CLOSEFN        fs_close;
    RTSMB_FS_RENAMEFN       fs_rename;
    RTSMB_FS_DELETEFN       fs_delete;
    RTSMB_FS_MKDIRFN        fs_mkdir;
    RTSMB_FS_RMDIRFN        fs_rmdir;
    RTSMB_FS_SETCWDFN       fs_set_cwd;
    RTSMB_FS_PWDFN          fs_pwd;
    RTSMB_FS_GFIRSTFN       fs_gfirst;
    RTSMB_FS_GNEXTFN        fs_gnext;
    RTSMB_FS_GDONEFN        fs_gdone;
    RTSMB_FS_STATFN         fs_stat;
    RTSMB_FS_CHMODEFN       fs_chmode;
    RTSMB_FS_GET_FREEFN     fs_get_free;
} SMBFILEAPI;
```

The following function pointers may be null if the file system does not support unicode. Their semantics are the same as their ASCII counterparts, but instead of a 'char *' for the filename, they take an 'unsigned short *'.

```
    RTSMB_FS_GET_FREEFN     fs_get_free;
    RTSMB_FS_WOPENFN        fs_wopen;
    RTSMB_FS_WRENAMEFN      fs_wrename;
    RTSMB_FS_WDELETEFN      fs_wdelete;
    RTSMB_FS_WMKDIRFN       fs_wmkdir;
    RTSMB_FS_WRMDIRFN       fs_wrmdir;
    RTSMB_FS_WSETCWDFN      fs_wset_cwd;
    RTSMB_FS_WPWDFN         fs_wpwd;
    RTSMB_FS_WGFIRSTFN      fs_wgfirst;
    RTSMB_FS_WSTATFN        fs_wstat;
    RTSMB_FS_WCHMODEFN      fs_wchmode;
    RTSMB_FS_WGET_FREEFN    fs_wget_free;
```

The functions in SMBFILEAPI that must be pointed to by the prtsmb_filesys struct are defined as follows:

int **fs_open** (char * name, unsigned short flag, unsigned short mode)

name - name of the file to open
flag - one or more of the following flags bit-wise OR'ed together:
RTSMB_O_RDONLY - open for reading only
RTSMB_O_WRONLY - open for writing only
RTSMB_O_RDWR - open for reading and writing
RTSMB_O_APPEND - open for appending
RTSMB_O_CREAT - create file if it does not exist
RTSMB_O_TRUNC - truncate file
RTSMB_O_EXCL - open exclusive
RTSMB_O_BINARY - open in binary mode (DOS/Windows)
RTSMB_O_TEXT - open in text mode (DOS/Windows)
mode - one of the following:
RTSMB_S_IWRITE - sets the file as writable when creating a file
RTSMB_S_IREAD - sets the file as readable when creating a file

Description:

Opens a file according to the flags and mode specified

Returns:

file descriptor value on success, negative on failure

---

long **fs_read**

(int fd, unsigned char * buf, long count)

fd - file descriptor to read from
buf - buffer to put data in
count - max number of bytes to read

Description:

Reads data from an open file.

Returns:

number of bytes read on success, negative on failure

---

long **fs_write**

(int fd, unsigned char * buf, long count)

fd - file descriptor to write to
buf - buffer of data to be written
count - max number of bytes to write

Description:

Writes data to an open file.

Returns:

number of bytes written on success, negative on failure

---

long **fs_lseek**

(int fd, long offset, int origin)

fd - file descriptor to set pointer of
offset - position (byte offset) to seek to
origin - point of reference for offset; one of the following:

RTSMB_SEEK_SET - seek from beginning of file
RTSMB_SEEK_CUR - seek from the current read/write position
RTSMB_SEEK_END - seek from end of file

Description:

Sets the file read/write position.

Returns:

the new position in bytes from the beginning of the file

---

int **fs_close**

(int fd)

fd - file descriptor to close

Description:

Closes an open file.

Returns:

0 on success, negative on failure

---

char **fs_truncate**

(int fd, long offset)

fd - file descriptor
offset - position to truncate at

Description:

Cuts off a file to offset bytes.

Returns:

non-zero on success, 0 on failure

---

char **fs_flush**

(int fd)

fd - file descriptor

Description:

Flushes all buffers associated with the given open file to the disk.

Returns:

non-zero on success, 0 on failure

---

char **fs_rename**

(char * name, char * newname)

from - the current file path
to - the new file path

Description:

Renames/moves a file.

Returns:

non-zero on success, 0 on failure

---

char **fs_delete**

(char * name)

d - file name to delete

Description:

Deletes the given file.

Returns:

non-zero on success, 0 on failure

---

char **fs_mkdir**

(char * name)

d - path of new directory to create

Description:

Creates a directory if not already present.

Returns:

non-zero on success, 0 on failure

---

char **fs_rmdir**

(char * name)

d - path of directory to remove

Description:

Removes a directory if present.

Returns:

non-zero on success, 0 on failure

---

char **fs_set_cwd**

(char * name)

to - path of new current working directory

Description:

Sets the current working path for this thread. This is the path prepended to relative paths to resolve an absolute path.

Returns:

non-zero on success, 0 on failure

---

char **fs_pwd**

(char * name, long size)

to - string buffer to print the working directory path in
size - max bytes to write to the buffer

Description:

Retrieves the current working directory path into the given buffer.

Returns:

non-zero on success, 0 on failure

char **fs_gfirst**

(PSMBDSTAT dirobj, char * name)

dirobj - pointer to an uninitialized SMBDSTAT struct
name - the pattern to use in searching (can include path)

Description:

This routine retrieves the first entry in the given directory. It must populate the given SMBDSTAT struct with the relevant information for the first directory entry. The SMBDSTAT struct is defined as:

```
struct smbdstat
{
    PSMBFILEAPI fs_api;
    #if (INCLUDE_RTSMB_UNICODE)
    char    filename[SMBF_FILENAMESIZE * 2 + 2];
    char    short_filename[26];
    #else
    char    filename[SMBF_FILENAMESIZE + 1];
    char    short_filename[13];
    #endif

    char unicode; /* set it to zero if
    filename is ascii, or non-zero when it is unicode */
    unsigned short fattributes;
    unsigned long fsize;

    TIME fatime64; /* last access time */
    TIME fwtime64; /* last write time */
    TIME fctime64; /* last create time */
    TIME fhtime64; /* last change time */

    unsigned char fs_obj[RTSMB_DSTAT_FS_OBJ_SIZE];
};
```

where fs_api is a pointer to the file system interface being used, filename is the last found found, unicode is a boolean value indicating whether the contents of filename are in unicode or not, fattributes is one or more of the following, bitwise OR'd together:

SMBF_ATTRIB_ISROOT - current entry is the root directory
SMBF_ATTRIB_ISDIR - current entry is a directory
SMBF_ATTRIB_ISVOL - current entry is a volume
SMBF_ATTRIB_RDONLY - current entry is marked read-only
SMBF_ATTRIB_WRONLY - current entry is marked write-only
SMBF_ATTRIB_RDWR - current entry is marked read and write

The TIME type is a 64-bit struct defined as follows:

```
typedef struct {
dword low_time;
dword high_time;
} TIME;
```

It holds the number of 100-nanosecond intervals since January 1st, 1601. There are conversion routines (to and from) for both unix epoch time and DOS time if those are simpler for your file system to use.

Returns:

non-zero on success, 0 on failure (directory is empty)

char **fs_gnext**

(PSMBDSTAT dirobj)

dirobj - pointer SMBDSTAT initialized by gfirst

Description:

Gets the next entry in a directory. See gfirst for description of the format of the entry information contained in the SMBDSTAT struct.

Returns:

non-zero on success, 0 on failure (there are no more entries)

void **fs_gdone**

(PSMBDSTAT dirobj)

dirobj - pointer SMBDSTAT initialized by gfirst

Description:

When a directory's entries have been enumerated through gfirst, gnext, it must finally be closed using this routine. This routine should free any resources allocated by gfirst/gnext.

Returns:

nothing

char **fs_stat**

(char * name, PSMBFSTAT stat)

name - file path to retrieve information for
vstat - pointer to an SMBFSTAT struct to fill with file information

Description:

If the file is present, fills the given SMBFSTAT struct with information about the file. SMBFSTAT is defined as:

```
struct smbfstat
{
    unsigned short f_mode;
    unsigned long f_size; /* file size, in bytes */
    TIME f_atime64; /* last access time */
    TIME f_wtime64; /* last write time */
    TIME f_ctime64; /* last create time */
    TIME f_htime64; /* last change time */
};
```

where f_mode is the file mode; one or more of :

RTSMB_FMODE_IFDIR - is a directory
RTSMB_FMODE_IFREG - mode is regular (?)
RTSMB_FMODE_IWRITE - Write permitted
RTSMB_FMODE_IREAD - Read permitted.

The TIME type is a 64-bit struct defined as follows:

```
typedef struct {
dword low_time;
dword high_time;
} TIME;
```

It holds the number of 100-nanosecond intervals since January 1st, 1601. There are conversion routines (to and from) for both unix epoch time and DOS time if those are simpler for your file system to use.

Returns:

non-zero on success, 0 on failure

---

char **fs_chmode**

(char * name, unsigned char attributes)

name - file path

attributes - new attributes

Description:

Changes the attributes of a file. See open for a description of attribute flags.

Returns:

non-zero on success, 0 on failure

---

char **fs_get_free**

(char * name, unsigned long *blocks, unsigned long *bfree, unsigned long *sectors_per_block, unsigned short *bytes_per_sector)

name	- a filename on the hard disk to check
blocks	- a return value for the total number of units
bfree	- a return value for the number of free units
sectors_per_block	- a return value for the number of sectors per unit
bytes_per_sector	- a return value for the number of bytes per sector

Description:

Gets the number of total units and free units in the file system.

Returns:

non-zero on success, 0 on failure

RTIP 4.0

SMB Server Manual

Chapter Four: SMB API

CHAPTER 4

API FUNCTIONS

The following pages document each call to the RTSMB API. But first, a quick categorical overview of the functions.

PROGRAM FLOW

rtsmb_srv_init

This initializes the server and allows the rest of the API functions to be used.

rtsmb_srv_cycle

This runs one 'iteration' of the server – that is, the server will handle at most one waiting request on each connection. This blocks waiting for network input up to a specified timeout.

rtsmb_srv_shutdown

This stops the server.

CONFIGURATION

rtsmb_srv_read_config

This reads the specified file and sets up users/shares according to what the file specifies. The file format is similar to .ini files; you can see an example in the file smbconf.txt.sample.

rtsmb_srv_set_mode

This sets the server mode to either user- or share-mode.

rtsmb_srv_get_mode

This returns whether the server is in user- or share-mode.

USER MANAGEMENT

rtsmb_srv_register_group

This initializes the specified group.

rtsmb_srv_register_user

This initializes the specified user, granting him or her access.

rtsmb_srv_delete_user

This deletes the specified user, denying access to him or her.

rtsmb_srv_add_user_to_group

This adds a user to the specified group.

rtsmb_srv_remove_user_from_group

This removes a user from the specified group, dropping their access rights from that group.

rtsmb_srv_set_group_permission

This sets what shares to which a group has access and what kind of access.

SHARE MANAGEMENT

rtsmb_srv_share_add_tree

This sets up a directory to be shared to SMB clients.

rtsmb_srv_share_add_printer

This sets up a printer to be shared to SMB clients.

rtsmb_srv_share_add_ipc

This sets up the required administrative IPC share with the given password.

rtsmb_srv_share_remove

This stops sharing the specified share.

Any function that takes a string has a complement function with the same name but a suffix of "_uc" to indicate it takes a Unicode string instead of an ASCII string. These functions only exist if the server was compiled with Unicode support.

RTSMB_SRV_SHARE_ADD_TREE ()

Function:

Set up a new disk share.

Summary:

#include “srvapi.h”

int rtsmb_share_add_tree (name, comment, api, path, flags, permissions, password)

PFCHAR name	- The name of share
PFCHAR comment	- Comment about share
PVFILEAPI	- The vfile api used for this share
PFCHAR path	- The file path to share directory
byte flags	- Flags detailing share properties
byte permissions	- Default permissions for share. Only used when server is in share mode.
PFCHAR password	- Password used to give default permissions. Setting this to NULL will disable password checking for this share. Only used while server is in share mode

Description:

This function initializes a share for use. NULL can be passed for the comment parameter. If the api parameter is NULL, the default filesystem API is used (this is usually what you want if you have only one filesystem). The name parameter cannot be longer than 12 characters. The comment parameter cannot be longer than 30 characters. The permissions parameter can be one of SECURITY_READ, SECURITY_WRITE, SECURITY_READWRITE, or SECURITY_NONE. It is only used in share mode.

The flags parameter is a bitwise or of zero or more of the following flags:

SHARE_FLAGS_8_3

SHARE_FLAGS_8_3 causes RTSMB to mangle all incoming filenames so that they fit the DOS 8.3 filename format. Use this if your file system cannot handle long filenames.

SHARE_FLAGS_CASE_SENSITIVE

SHARE_FLAGS_CASE_SENSITIVE causes RTSMB to uppercase all incoming filenames so that Windows clients don’t get confused when a filename they create as FileName.txt cannot be accessed as FILENAME.TXT. Use this if your filesystem is case sensitive.

SHARE_FLAGS_CREATE

SHARE_FLAGS_CREATE causes RTSMB to create the shared directory. Thus, sharing “C:\path\that\does\not\exist” would cause RTSMB to call mkdir on each component of it.

Returns:

Returns 0 on success and a negative value on failure (not enough share buffers left).

RTSMB_SRV_SHARE_ADD_IPC ()

Function:

Set up an IPC$ share.

Summary:

#include “srvapi.h”

int rtsmb_share_add_ipc (password)

PFCHAR password	- Password used to give access. Setting this to NULL will disable password checking for this share. Only used while server is in share mode

Description:

This function initializes the Inter-Process Communication share. Clients connect to this share to get the list of other shares. After this call, a share called “IPC$” is created.

You will *need* to call this at least once. Clients expect the IPC share to exist. However, you don’t need to call it more than once.

Returns:

Returns 0 on success and negative value on failure (not enough share buffers left).

RTSMB_SRV_SHARE_REMOVE ()

Function:

Removes a share.

Summary:

#include “srvapi.h”

int rtsmb_share_remove (name)

PFCHAR name - The name of share to remove

Description:

This function unshares a previously added share. Access to it from this point on will be denied (after requests currently being processed are done). This function may block for a little while in order to give processes using the share a chance to quit (if not in POLLOS mode).

Returns:

Returns 0 on success and negative value on failure (share not found).

RTSMB_SRV_SET_MODE ()

Function:

Sets the authorization mode of server.

Summary:

#include “srvapi.h”

void rtsmb_set_mode (mode)

byte mode - The authorization mode to use. Can be AUTH_USER_MODE or AUTH_SHARE_MODE

Description:

This function tells RTSMB how to authorize clients. If the mode is user-based, each client will have to provide a valid username/password pair. If the mode is share-based, each client provides a password for each share they want to use. The default is share mode.

Returns:

Nothing.

RTSMB_SRV_GET_MODE ()

Function:

Gets the authorization mode of server.

Summary:

#include “srvapi.h”

byte rtsmb_get_mode ()

Description:

This function gets the authorization mode of the server.

Returns:

This function returns the authorization mode of the server. The mode is either AUTH_SHARE_MODE if operating in share mode or AUTH_USER_MODE if operating in user mode.

RTSMB_SRV_REGISTER_GROUP ()

Function:

Registers a group.

Summary:

#include “srvapi.h”

BBOOL rtsmb_register_group (name)

PFCHAR name - The name of the group

Description:

This function registers a group name to be used by:

rtsmb_set_group_permission,
rtsmb_add_user_to_group, and
rtsmb_remove_user_from_group.

Note: Only used if server is in user mode.

Returns:

This function returns 1 on success, 0 on failure (if too many groups already registered).

RTSMB_SRV_REGISTER_USER ()

Function:

Registers a user.

Summary:

#include “srvapi.h”

BBOOL rtsmb_register_user (name, password)

PFCHAR name - The name of the user

PFCHAR password - The plaintext password of the user

Description:

This function registers a username/password pair to be used when authenticating clients. This username can be passed to:

rtsmb_delete_user,
rtsmb_add_user_to_group, and
rtsmb_remove_user_from_group.

Note: Only used if server is in user mode.

If the special constant AUTH_GUESTNAME is passed as a username, the server will allow users who cannot login (i.e. do not know a username/password) to login with this user’s access rights. If you enable this, it is wise to limit this guest’s access to shares. The guest’s password is ignored.

Returns:

This function returns 1 on success, 0 on failure (if too many users already registered).

RTSMB_SRV_DELETE_USER ()

Function:

Deletes a user’s information.

Summary:

#include “srvapi.h”

BBOOL rtsmb_delete_user (name)

PFCHAR name - The name of the user

Description:

This function deletes a username/password pair registered by:

ebmsb_register_user

Note: Only used if server is in user mode.

Returns:

This function returns 1 on success, 0 on failure (invalid name).

RTSMB_SRV_INIT ()

Function:

Initializes RTSMB.

Summary:

#include “srvapi.h”

void rtsmb_srv_init (ip, mask, net_name, group_name)

PFBYTE ip	- The four-byte array representing the server’s ip
PFBYTE mask	- The four-byte array representing the server’s ip mask
PFCHAR net_name	- The ASCII name that the server will use as a Netbios name
PFCHAR group_name	- The ASCII workgroup that the server will connect to

Description:

This function initializes the SMB server for use. This must be called before any other API routine. In addition, the native network stack must be initialized before calling this.

If mask is passed as NULL, then the default mask of 255.255.255.0 will be used. If net_name is NULL, then the value of **CFG_RTSMB_DEFAULT_NET_NAME** in smbconf.h will be used. If group_name is NULL, then the value of **CFG_RTSMB_DEFAULT_GROUP_NAME** will be used.

Returns:

Nothing.

RTSMB_SRV_SET_GROUP_PERMISSION ()

Function:

Determines a group’s access rights to one share.

Summary:

#include “srvapi.h”

BBOOL rtsmb_set_group_permission (group, share, mode)

PFCHAR group	- The name of the group
PFCHAR share	- The name of the share
Byte mode	- The access mode for group. May be **SECURITY_READ**, **SECURITY_WRITE**, **SECURITY_READWRITE**, **or SECURITY_NONE**

Description:

This function assigns a group certain access rights to one share. Whenever a user belonging to this group logs in, they are granted at least these rights. If this is not called for a particular share, the group has SECURITY_NONE access to that share by default.

Returns:

This function returns 1 on success, 0 on failure (invalid group name, share name, or mode).

RTSMB_SRV_ADD_USER_TO_GROUP ()

Function:

Adds a user to a group.

Summary:

#include “srvapi.h”

BBOOL rtsmb_add_user_to_group (user, group)

PFCHAR user - The name of the user.

PFCHAR group - The name of the group.

Description:

This function assigns a user to a group. This means they have whatever access rights the group as a whole has.

Returns:

This function returns 1 on success, 0 on failure (invalid user or group name).

RTSMB_SRV_REMOVE_USER_FROM_GROUP ()

Function:

Removes a user from a group.

Summary:

#include “srvapi.h”

BBOOL rtsmb_remove_user_from_group (user, group)

PFCHAR user - The name of the user.

PFCHAR group - The name of the group.

Description:

This function removes a user from a group. They no longer will have the same access rights as the group.

Returns:

This function returns 1 on success, 0 on failure (invalid user or group name).

RTSMB_SRV_SHUTDOWN ()

Function:

Shuts down RTSMB.

Summary:

#include “srvapi.h”

void rtsmb_shutdown ()

Description:

This function shuts down the SMB server.

Returns:

Nothing.

RTSMB_SRV_READ_CONFIG ():

Function:

Parses a configuration file.

Summary:

#include “srvapi.h”

void rtsmb_read_config (filename)

PFCHAR filename - The filename of the config file to read.

Description:

This function reads in the config file and parses it for parameters. By doing this, you can avoid recompiling a program to change how shares are set up or what the password for user ‘bob’ is. See Appendix B for an example config file.

Returns:

This function returns 0 on success, else on failure (invalid syntax, couldn’t read file).

RTSMB_SRV_CYCLE ():

FUNCTION:

Runs one step of RTSMB server in blocking mode.

Summary:

#include “srvapi.h”

void rtsmb_cycle (timeout)

long timeout - The maximum amount of time to block in milliseconds. If negative, there is no maximum.

Description:

This function process incoming requests and performs internal housekeeping. This needs to be called periodically for RTSMB Server to work correctly. If you can’t afford to block on the network, just pass in 0 as a timeout.

Returns:

Nothing.

RTSMB_SRV_SET_IP ():

Function:

Changes RTSMB’s idea of what its IP is.

Summary:

#include “srvapi.h”

void rtsmb_srv_set_ip (ip, mask)

PFBYTE ip - The four-byte array representing the server’s IP.

PFBYTE mask - The four-byte array representing the server’s subnet mask.

Description:

This function changes the IP’s that RTSMB uses to send data to and from.

Use this if your IP has changed or you want to switch subnets.

If mask is passed as NULL, then the default mask of 255.255.255.0 will be used.

RTIP 4.0

SMB Server Manual

Appendix A: Examples

APPENDIX A

APPENDIX A – EXAMPLE CODE

The following program shows how to initialize and run the RTSMB server.

Before this code sample is run, it is assumed that networking has been initialized.

This code initializes the server, configures the available shares, configures the users allowed to log on, and calls the server cycle functions, which allow the server to process requests.

As a result of this code, the server makes itself a member of the network group and makes its shares available. Any requests that come in are handled.

Sample Program

```
#define "srvapi.h"
// Before this function is called, the network stack must be initialized
int run_rtsmb (PFBYTE ip, PFBYTE mask, PFCHAR name, PFCHAR
workgroup)
{
// Initialize rtsmb server with the specified tcp/ip information
rtsmb_srv_init (ip, mask, name, workgroup);

// To make it interesting, let's set the server to user mode.
// The default is share mode.
// In user mode, users must authenticate themselves to the server
using a username and password pair.  In share mode, they must
provide a share-specific password for each share. //
// Because we set this as user mode, we can safely set no
passwords for the shares we set up, since those passwords are only
used in share mode.
rtsmb_srv_set_mode (AUTH_USER_MODE);
// Now, let's set up a share for the users to enjoy.
// We'll set up a basic share, one that uses the default filesystem API
// This is a windows share, so we specify that the server should use
the 8.3 file naming format (where any filename must be only 8
characters long with an extension of three characters).
rtsmb_srv_share_add_tree ("c", "c drive", NULL, "C:\\",
SHARE_FLAGS_8_3, SECURITY_NONE, NULL);
// Now, let's add an IPC share without a password so users can
browse the other shares.
rtsmb_srv_share_add_ipc (NULL);
// We need at least one group to add users to.  We also define the
access rights "colors" has to each share.  If not specified, no rights
are granted.
rtsmb_srv_register_group ("colors");
rtsmb_srv_set_group_permission ("colors", "c",
SECURITY_READWRITE);
rtsmb_srv_set_group_permission ("colors", "IPC$",
SECURITY_READWRITE);
// Now, we setup all the users we want to allow, with passwords.
// In this case, it's just the user "red" and the user "blue."
// We also add them to the group colors, so they can actually access
some shares.
rtsmb_srv_register_user ("red", "12beh;bal3");
rtsmb_srv_add_user_to_group ("red", "colors");
rtsmb_srv_register_user ("blue", "hello");
rtsmb_srv_add_user_to_group ("blue", "colors");
//Main Loop
// Now we'll continually run the server and process requests until the
user hits a 'g' on the console.
while(1)
{
      // Process one packet from all the main thread sockets.
      // These include the session socket and name service socket as
well as
      any sessions the main thread happens to be handling.
      rtsmb_srv_pollos_cycle ();

      // Quick check to see if user at console wants to stop server.
      if(kbhit())
{
int ch = getch();
if (ch == 'd')
{
// Stop main loop if user hits 'd'.
break;
            }
else if (ch == 'o')
            {
// Just for kicks, let's revoke blue's access if user hits 'o'.
rtsmb_srv_delete_user ("blue");
            }
else if (ch == 'g')
            {
// And add him again if user hits 'g'.
rtsmb_srv_register_user ("blue", "hello");
rtsmb_srv_add_user_to_group ("blue", "colors");
                                                }
            }
      }

// Shutdown and stop the server
rtsmb_srv_shutdown ();
return 0;
}
Here is a code snippet that shows how to use groups effectively.
Multiple Groups
// Assume that before this, the share "cat" was created, as well as the
standard IPC share.
//Put the server into user mode. (It defaults to share mode)
rtsmb_srv_set_mode (AUTH_USER_MODE);
rtsmb_srv_register_group ("a");
rtsmb_srv_set_group_permission ("a", "cat", SECURITY_WRITE);
rtsmb_srv_set_group_permission ("a", "IPC$", SECURITY_READ);

/* Set up a group called 'b' which can read files in the test directory.
Note that both 'a' and 'b' groups can browse the files in the "cat" share.
Only 'a' can write to them and only 'b' can read them, though.  To
prevent someone from browsing the shares, the user needs to have
SECURITY_NONE privileges. */
rtsmb_srv_register_group ("b");
rtsmb_srv_set_group_permission ("b", "cat", SECURITY_READ);
rtsmb_srv_set_group_permission ("b", "IPC$", SECURITY_READ);

/*
Set up a group for those we don't trust.  This will be used for guests.
*/
rtsmb_srv_register_group ("untrusted");
rtsmb_srv_set_group_permission ("untrusted ", "cat",
SECURITY_NONE);
rtsmb_srv_set_group_permission ("untrusted ", "IPC$",
SECURITY_READ);
/*
Define a guest account, but attach it to group 'untrusted' only (don't
want to give guests write access here).  SMB_GUESTNAME defaults
to "".
*/
rtsmb_srv_register_user (SMB_GUESTNAME, NULL);
rtsmb_srv_add_user_to_group (SMB_GUESTNAME, "untrusted");

/*
Set up a real user, called 'bob' with password 'finagle'.  Add him to the
'a' group.
User names are not case sensitive.  Passwords may be depending on
protocol negotiated.
*/
```

```
rtsmb_srv_register_user ("BOB", "finagle");
rtsmb_srv_add_user_to_group ("Bob", "a");

/*
Add the user kris, with 'blarg' as the password.  Here, she is a
member of both the 'a' and 'b' groups.  She can now read and write to
the "cat" share.  Again, as you can see, user names are not case
sensitive.
*/
rtsmb_srv_register_user ("EmIIY", "blarg");
rtsmb_srv_add_user_to_group ("eMily", "a");
rtsmb_srv_add_user_to_group ("emily", "b");

/*
Now, Bob can write all he wants to "cat", but cannot read files in it.
Meanwhile, Kris can read and write to her heart's content.  Guests are
given no access to the share, but can browse the list of shares (via
their read privileges with the IPC$) wistfully.  A user has, for a given
share, the sum of all access rights of his/her groups.
As long as the user has non-NONE status, they can connect to the
share and browse through the files on the share.  Thus, as for the
IPC$, all that matters is you have some access to browse the list of
shares.  Having NONE access on the IPC$ does not prevent you
from accessing shares you know the name of.
*/

Here we define a main loop that blocks on network sockets.
Smarter Main Loop

/*
One problem with the main loop in the first example is that you waste
a lot of CPU cycles calling rtsmb_srv_cycle in non-blocking mode.
Here is an example to use blocking mode effectively.
*/

void rtsmb_srv_loop ()
{
while (1)
rtsmb_srv_cycle ();
}

// This function is only called after user-initialization and shares have
been set up.
// For an example of how to do that, look at example 1.
void rtsmb_srv_start ()
{
// spawn off a process to handle main loop just by itself.
os_spawn_task (TASKCLASS_USER_APPTASK, rtsmb_srv_loop, 0, 0,
0, 0);
// now check for input like we did before, sleeping for 1 sec each time
while(1)
{
sleep (1);
// Quick check to see if user at console wants to stop server.
if(kbhit())
{
int ch = getch();
if (ch == 'd')
{
// Stop main loop if user hits 'd'.
break;
            }
else if (ch == 'o')
            {
// Just for kicks, let's revoke blue's access if user hits 'o'.
rtsmb_srv_delete_user ("blue");
            }
else if (ch == 'g')
            {
// And add him again if user hits 'g'.
rtsmb_srv_register_user ("blue", "hello");
rtsmb_srv_add_user_to_group ("blue", "colors");
            }
            }
      }

rtsmb_srv_shutdown ();
}
APPENDIX B – EXAMPLE CONFIG FILE

# Comments are introduced by a '#' character

# A config file is broken up into sections.  Valid sections include
# global, ipc, share, printer, group, and user.
#
# Each section is processed as the file is read.  So, it is best to
specify
# all shares first, then users and a global.

# The ipc section has only one optional variable, password.  This
controls the # password needed to get a list of available shares on the
server.  If the
# password is unspecified, no password is needed.

[ipc]
[end]

# A share section will set up a file share's availability.  There can be
many share sections.  Each share uses the default file system.  The
following options are available:
#
# name : The share's name as it appears in Network Neighborhood
# comment : The share's comment  (optional)
# path : The directory to share
# flags : Configurable options for share.  Valid flags are dos_names,
#      case_sensitive, and create
#            dos_names : This enforces 8.3 syntax to filenames
#            case_sensitive : The filesystem is case sensitive
#            create : The path will be created if it doesn't exist
#      The default for all of these is off.  Adding the flag names in a
space
#      delimited line turns the options on.
# permission : The permission granted to users who specify the
correct
#      share password.  This is only used when the server is in share
mode.
# password : The password for share mode.  If omitted, no password
is needed.
[share]
name = test
comment = native fs
path = c:\rtsmb
flags = create
permission = rw
password = smb
[end]

# Printer shares are defined by [printer] sections.
#
# It has all the same variables as the [share] section, except for a
few changes.
# There isn't a permission variable, since that is not customizable for
a printer,
# and there are a few additional configuration parameters.  The reason
you
# need to set up all the share variables is that each printer must have
some
# filesystem space for its temporary files.
#
# The unique variables:
#
```

```
# number : the number of the port the printer is connected to, 1-based.
#           (i.e. parallel port 0 is number = 1)
# drivername : the name of the driver to use (you have to know correct string)
#       This is the same as the string Windows uses to describe the driver.

[printer]
name = Printer
comment = Test Printer
path = c:\rtsmb\printer
flags = create
#password =
number = 1
drivername = HP LaserJet 1100
[end]

# A group section will register a group to which users can belong.  The group
# concept is invisible to the user.  It is only included to make administration
# easier.  There can be many group sections.  The valid variables are name and shares.
#
# name : Specify the name of the group.
# shares : A space-delimited list of shares and permissions.  The syntax is as
#      follows: share:permissions
#      For example, if a share is called "music" and you wanted to give this
#      group write only access to it, you would say music:wo.  If you also wanted to give this group read-write access to the share "books," you could say "music:wo books:rw".
#           The valid permission flags are
#                                       ro : read only
#                                       wo : write only
#                                       rw : read and write
#      If you omit a share:permission pair, or type "share:" (e.g. "music:")
#      with no permissions, access is none.
#   When a permission is set to write only, the user can still browse the listings
#   of files, but cannot read any of them.  Therefore, they can only upload files,
#   and can't download them.
#
#   When a permission is set to read only, the user can still browse the listings and view the contents of files, but cannot delete, create, or modify any files on the share.
#
#      Under almost any circumstance, you will want to grant every group read
#      and write access to the ipc$ share (this allows them to browse the list of
#      available shares).

[group]
name = trusted
shares = test:rw ipc$:rw Printer:rw
[end]

[group]
name = marketing
shares = test:ro ipc$:rw Printer:rw
[end]

[group]
name = untrusted
shares = test:ro ipc$:rw Printer:
[end]

# A user section sets up a user with a name, password, and group designations.
# Valid variables are name, password, and groups.
#
# name : User name.  This is not case sensitive.
# password : Password for user.  If omitted, user needs none.
# groups : Space-delimited list of groups user belongs to.  This user will have
# the most favorable access to any given share that any one of his or her
# groups allows.

[user]
name = mike
password = blarg
groups = trusted marketing
[end]

[user]
name = bob
password = hopscotch
groups = marketing
[end]

# The global section has three variables:  mode, guest, and guestgroups.
# If mode is set to "user", user-mode authentication will occur.
# If mode is set to "share", share-based authentication will occur.
# The default is share.
#
# If guest is set to "yes", a guest account will be created that will be used if a
# user does not have a username.
#
# If guest is set to "yes", then guestgroups will be a space-delimited list of groups
# guest belongs to and the permissions.  See the user section above for specifics
# on the syntax.
#
# You only need one global section.

[global]
mode = user
guest = yes
guestgroups = trusted
[end]
```