PORTABLE FTP SERVER TECHNICAL REFERENCE

51 E Campbell Ave
Suite 160
Campbell, CA. 95008

email: support@iniche.com
http://www.iniche.com
support: 408.540.1160
fax: 408.540.1161

InterNiche Technologies Inc. has made every effort to assure the accuracy of the information contained in this documentation. We appreciate any feedback you may have for improvements. Please send your comments to **support@iniche.com**.

The software described in this document is furnished under a license and may be used, or copied, only in accordance with the terms of such license.

Rev-10.2005

Portions of the InterNiche source code are provided under the copyright of the respective owners and are also acknowledged in the appropriate source files:

TABLE OF CONTENTS

1. OVERVIEW

This technical reference is provided with the InterNiche portable FTP server. The purpose of this document is to provide enough information so that a moderately experienced "C" programmer with a reasonable understanding of TCP/IP protocols can port the InterNiche FTP server to a new environment.

It is assumed that one of the InterNiche demo packages is available as a working reference. Demos are currently available for MS-DOS systems; or on Windows 95, where the demo server runs as a Windows Application and uses the native Windows WinSock TCP/IP. The DOS Demo executable requires a PC running MS-DOS 3.0 or newer and access to ethernet via an ODI driver or an FTP Software type "Packet Driver". Demos can be provided on request which support PPP over a dialup link.

1.1 Terms and Conventions

In this document, the term "stack", when used without other qualification, means the InterNiche TCP/IP and related code as ported to an embedded system. "System" refers to your embedded system. "Sockets" refers to the TCP API developed for UNIX at U.C. Berkeley. A "user" or "porting engineer" usually refers to the engineer who is porting the server. An "end user" refers to the person who ultimately ends up using the "user's" product. "FCS" is an acronym for "First Customer Ship", the point in the software development cycle when the product is declared ready to ship. A "packet" is sequence of bytes sent on network hardware, also known as a "frame" or a "datagram".

Names of files, C structures, and C routines are displayed as follows: **c_routine()**

Samples of source code from C programs are displayed in these boxes:

```
/* C source file - the world's 1 millionth hello program. */
main()
{
    printf("hello world.\n");
}
```

1.2 A Note About DOS

We apologize in advance to those engineers who resent our distribution and documentation seeming PC-DOS or Intel x86 oriented. The FTP server has been ported to numerous CPUs and operating systems and has no internal DOS or Intel dependencies.

The DOS orientation is not because our engineers are enamored of DOS or the x86's technical elegance (they're not), but simply a matter of market realities. There are more embedded systems being developed on and for Intel x86 chips than any other, and more development environments based on DOS and Windows than any other platform. Every client we've had has had at least one PC to unzip the files, read the documentation, and build the

demo package. And PC based C Compilers and source level debuggers are affordable and varied.

An additional argument for using DOS to develop an embedded systems product is its lack of advanced features. We've yet to encounter an RTOS vendor who asserts their product has weaker multitasking or memory management features than DOS. By demonstrating that our products turn in excellent performance on plain vanilla DOS, we show that the stack has only minimal requirements from its host system.

1.3 What Does an FTP Server Do?

FTP stands for File Transfer Protocol. It is the basic mechanism for moving files between machines over TCP/IP based networks such as the Internet. FTP is a "client/server" protocol, meaning that one machine, the client, initiates a file transfer by contacting another machine, the server and making requests. The server must be operating before the client initiates his requests. Generally a client communicates with one server at a time, while most servers (including the InterNiche server) are designed to work with multiple simultaneous clients.

FTP is formally defined by the IETF document RFC0959. This software is compliant with that specification, and the RFC should be consulted for any detailed questions about the protocol itself. However a brief overview of the protocol is presented here.

When an FTP client contacts a server, a TCP connection is established between the two machines. The server does a passive open (a Sockets **listen**) when it begins operation; thereafter clients can connect with the server via active opens. This TCP connection persists for as long as the client maintains a session with the server, (usually determined by a human user) and is used to convey commands from the client to the server, and the server replies back to the client. This connection is referred to as the FTP command connection.

The FTP commands from the client to the server consist of short sets of ASCII characters, followed by optional command parameters. For example, the FTP command to display the current working directory is "XPWD". All commands are terminated by a carriage return-linefeed sequence (CRLF) (ASCII 10,13; or Ctrl-J, Ctrl-M). The servers replies consist of a 3 digit code (in ASCII) followed by some explanatory text. Generally codes in the 200s are success and 500s are failures. See the RFC for a complete guide to reply codes. Most FTP clients support a verbose mode which will allow the user to see these codes as commands progress.

If the FTP command requires the server to move a large piece of data (like a file), the server opens a second TCP connection to do this. This is referred to as the FTP data connection (as opposed to the aforementioned command connection). The data connection is opened, usually by the server back to a listening client, only for transporting the required data; and is closed as soon as all the data has been sent.

1.4 What Is a Port?

In the world of portable networking code, the code designer does not know what tasking system, user applications, or interfaces will be supported in the target system. So a "portable" stack is one that's designed with simple, generic interfaces in these areas, and a "glue" layer is created which maps this generic interface into the specific interfaces available on the target system. Using the example of sending a packet, the stack would be designed with a generic **send_packet()** call, and the porting engineer would code a "glue" routine to send the packet on the target system's network interface hardware.

Making a stack portable involves minimizing the number of calls which have to go across glue routines, and keeping the glue routines simple and therefore easy to implement. The glue routines also need to be well documented. The interfaces to the InterNiche FTP server have evolved through years of porting to a variety of processors, network media, and tasking systems. Wherever possible we have used standard interfaces (e.g. Sockets, ANSI C library) or included glue routines to illustrate their use.

The bulk of the work in porting a stack is understanding and implementing these glue routines. The InterNiche FTP server has three kinds of glue routines: the network API (usually Sockets), the user/password lookup, and the file system API.

1.5 Requirements

Before beginning a port, the programmer should ensure that the necessary resources are available in the target environment. Here is a brief summary of services the InterNiche FTP server needs from the system:

- A timer which ticks at least once a second.
- A non-volatile read/write method for storing database items (e.g. disk or flash memory)
- Memory as described below.
- A file system. This may be a conventional disk based file system, a flash device which supports file system like reads and writes, or a ram-based "virtual file system" like the one in the InterNiche WebPort portable Web server.

1.5.1 Memory Requirements

There is no easy way to determine the exact memory sizes required, however a rough idea can be obtained by examining the DOS DEMO executables. Some figures for prom/flash type memory are given below. This program is compiled for the Intel 8088 processor with Microsoft C 8.0, using default optimization options. It implements an FTP server and TCP/IP on a single ODI Ethernet driver. (Note: These figures are subject to change without notice, but are current as of 1/20/98):

BYTES	USE
17,084	static code
780	static data
17,864	**Total static size, when used as described.**

These sizes may also improve slightly when compiled for an Intel 186 or 386 chip. They generally worsen when ported to older RISC processors, and improve with some newer architectures such as ARM.

In addition, dynamic memory is required for each open session on the server. The exact amount can be varied to favor performance vs. memory usage. The usual recommended range is 600 to 8300 bytes per session.

1.5.2 Operating System Requirements

The FTP Server also requires a few basic services from the operating system. These are listed here:

clock tick - An unsigned 32 bit counter needs to be available to FTP which is incremented from 1 to 20 times a second. The incrementing must be regular and indicated by the value of **FTPTPS**.

memory access - The FTP server allocates memory via the macro **FTPSALLOC()** which always allocates a single fixed size block. This can be mapped directly to the standard **calloc() and free()** library calls or to a "partition" based system with very little effort.

CPU cycles - The routine **ftps_loop()** must be called periodically. Systems which support true multitasking can create a task which calls it whenever FTP work is pending, and otherwise puts it to sleep. It can also be driven by regular polling.

1.6 Demo Package Directories List

The InterNiche sources are typically distributed with InterNiche TCP/IP as a DOS **zip** file, **TCPIPSRC.zip**. This file should be unzipped with **pkunzip** (or a compatible utility) in such a way as to preserve the underlying directory structure. It includes the required libraries, makefiles, and include files needed to implement a Demo package build system. If you do not use InterNiche TCP, contact InterNiche for a package which includes TCP/IP libraries for your choice of compiler.

On DOS, the command to unpack the code is:

```
c:> pkunzip -d ftpsrc.zip
```

TAR and **gzip** versions are also available.

Assuming you have unpacked into a directory named **\demo**, the **pkunzip** should create the following directories in the **\demo** directory:

dosmain - **main()** routine in file **dosmain.c**
ftp - the FTP server application, along with sample FTP Client code
inet - IP, UDP and related sources (excluding TCP). Includes startup, interface and buffer management code
misclib - demo menu system, utility routines, and some file IO

Other sub-directories may be present depending on you target system, however these contain no code required for the FTP Server per-se. Please refer to **toolnote.doc** for information about building the sources.

2. STEP BY STEP PORTING GUIDE

The section describes the steps needed to port the InterNiche FTP Server to a new environment. The discussions below generally assume that the stack is being ported to a small or embedded system with a sockets API interface and that a minimal ANSI C library is available.

The recommended steps to getting the server working on your target system are as follows:

1. Copy the portable source files into your development environment.
2. Create your version of **ftpport.h** and compile portable sources.
3. Code your glue layers (**ftpsport.c**, etc.) and compile.
4. Build a system, test, and debug.

2.1 Port Dependent Files

Before beginning step one, you should be aware of which files in the InterNiche distribution are the "portable" files, and which are not. The portable files are those which should be compiled and used on any target system without modification. The unportable, or "port dependent" files, are those which will need to be replaced or modified for different target systems. The first file listed below is the primary FTP Server source file, which should NOT need to be modified in the course of a normal port. If you feel you need to modify this file in the course of a routine port, please discuss it with InterNiche's technical support staff first, so we can either suggest an alternative, or modify our sources to reflect the change.

2.2 Source File Lists

The portable FTP source files. **Do Not Modify.**

ftpsrv.c
ftpsrv.h

The network (Sockets) glue files, may need modification:

ftpssock.c

The DOS Demo port file for the FTP Server, will need rewriting for other OSes:

ftpsdos.c
ftpsport.h

InterNiche FTP Client files - can be used as is with InterNiche menuing system, else may be replaced or omitted:

ftpclnt.c - basic FTP Client code - portable, should not need to be modified.
ftpcprn.c - message printer - may need modification.
ftpmenu.c - menuing InterNiche system extensions for FTP Client user commands.
ftpclnt.h - FTP Client definitions - portable, should not need to be modified.

2.3 Master FTP Server Port File: ftpport.h

Before you compile these files you should create a version of the file **ftpport.h.** This file contains most of the port dependent definitions in the stack. CPU architectures (big vs. little endian), compiler idiosyncrasies, and optional features (e.g. support for disk drive letters) are controlled in this file. A single mistake in this file (such as getting big and little endian confused) will guarantee that your port won't work properly. Taking a few hours up front to implement the file line by line is time well spent. This section outlines the basic contents of **ftpport.h**.

2.3.1 Standard Macros and Definitions

The InterNiche FTP Server code expects **TRUE**, **FALSE**, and **NULL** to be defined within the scope of **ftpport.h.** The best way to do this is usually to include the standard C library file **stdio.h** inside **ftpport.h**. If **stdio.h** is impractical to use or missing, the examples below will work for almost every C environment:

```
#ifndef TRUE
#define TRUE -1
#define FALSE 0
#endif
#ifndef NULL
#define NULL (void*)0;
#endif
```

2.3.2 Memory Allocation

The FTP Server code allocates and frees memory blocks dynamically as it runs. It uses the macros listed below to do this. If your target system supports standard C **calloc()** and **free()**, the macros map directly as follows:

```
#define FTPSALLOC(size) calloc(1,size)  /* get ftp session structure & buffer */
#define FTPSFREE(ptr) free(ptr)
```

Many RTOS systems do not use **calloc()** due to performance issues. Generally, they use a system which supports allocations of fixed size "partitions" (blocks) instead. The macros above are designed to support this - each of the **ALLOC()** macros only allocates a single size. Thus the macros can be mapped to a call to allocate the next largest partition size.

2.3.3 CPU Architecture

Four common macros are used from Berkeley UNIX for doing byte order conversions between different CPU architecture types. These are **htons()**, **htonl()**, **ntohs(),** and **ntohl().** They may be either macros or functions. They accept 16 and 32 bit quantities as shown, and convert them from network format ("big-endian") to the local CPU's format.

Most IP stacks already have these byte ordering macros defined. If this is the case you should try to find the existing **include** file which defines them and use it rather than duplicate

them. The information below is for the rare situations where these macros are not already available.

For Motorola 68000 family and most RISC chips, these can just return the variable passed, as in this example:

```
#define htonl(long_var)     (long_var)
#define htons(short_var)    (short_var)
#define ntohl(long_var)     (long_var)
#define ntohs(short_var)    (short_var)
```

The Intel 8086 and its descendants require the byte order in the word or long to be swapped. The **lswap()** and **bswap()** primitives provided with the InterNiche DEMO package can be used as illustrated here:

```
#define htonl(long_var)     lswap(long_var)
#define htons(short_var)    bswap(short_var)
#define ntohl(long_var)     lswap(long_var)
#define ntohs(short_var)    bswap(short_var)
```

2.3.4 Debugging Aids

dtrap() is a macro called by the Server code whenever it detects a situation which should not be occurring. The intention is for the **dtrap** routine or macro to try to trap to whatever debugger may be in use by the programmer. Think of it as an embedded break point. For most Intel x86 processor debuggers, this can be done with an **int 3** opcode. The macro below is effective if your Intel C compiler accepts inline assembly:

```
#define dtrap();     _asm{ int 3 }
```

You may need to play with the exact syntax to get it to compile. The stack code will generally continue executing after a **dtrap()**, but the **dtrap()**s usually indicate that something is wrong with the port. **NO PRODUCT BASED ON THIS CODE SHOULD BE SHIPPED UNTIL THE CAUSES OF ALL CALLS TO dtrap() HAVE BEEN ELIMINATED!** When it comes time to ship code, the **dtrap()**s can be redefined to a null function to slightly reduce code size.

The next few primitives have the same function and syntax as **printf()**. They have separate names so that they can have their output redirected or be completely disabled independently of each other. The first, **dprintf()**, is used throughout the stack code to print warning messages when something seems to be wrong. This should be mapped to a debugging console or log during development, and generally **ifdef**ed away for FCS. The **ns_printf()** call is for printing statistical information from the FTP client **menus** functions. These will certainly be useful during product development, and depending on the nature of the product may be needed in the end user's release.

In most ports, these can both be mapped to **printf()** as shown while the product is under development.

Note: This example works on Microsoft C, but some compilers will complain about this syntax since it ignores the fact that these names have parameters. You may have to experiment.

```
#define ns_printf   printf   /* same parms as printf, but works at init time */
#define dprintf     printf   /* same parms as printf, but works during run time */
```

For some products, it may make sense to define these away before FCS.

```
#define ns_printf(…)      /* define to nothing */
#define dprintf    (…)    /* define to nothing */
```

The last debugging tool in **ftpport.h** is the **#define NPDEBUG**. Defining this will cause the debug code to be compiled into the build. This code does things like check for valid parameters and sensible configurations during runtime. It frequently invokes **dtrap()** or **dprintf()** to inform the programmer of detected potential problems. You will want make sure it is defined during development. Unless PROM space is tight, it is OK to leave it defined for FCS - there will be no noticeable performance hit from this code.

```
#define NPDEBUG          1        /* enable debug checks */
```

2.3.5 Features and Options

The FTP Server has only one configurable feature: whether or not to support a disk drive letter as part of the current working directory. On systems which let the user distinguish between different disk devices, such as DOS, this is usually desired. Other system such as UNIX treat all the available disk devices as part of a single huge file system space. On these systems the feature is irrelevant and would only be confusing.

To enable the support for disk drive letters, include the following lines in your **ftpport.h** file:

```
/* set up disk options for DOS */
#define DRIVE_LETTERS       1       /* track drive as well as directory */
#define DEFAULT_DRIVE       "c:"    /* set default drive */
```

You can, of course, set your default drive to any letter. The syntax of the drive letter is the same as for MS-DOS.

There are also a series of configurable parameters which apply throughout the FTP Server. These are primarily buffer sizes for data such as user names and file IO. For most ports, the defaults in the demo files as shipped will be a good starting point for these values. If memory is tight, these numbers may be reduced, although at the cost of either performance or functionality. These number may also be increased to accommodate systems with unusually long user names, file paths, etc.

```
These parameters are:/* Implementation defines: */
#define FTPMAXPATH     128     /* maximum path length, excluding file name */
#define FTPMAXFILE     16              /* maximum file name length w/o path */
#define CMDBUFSIZE     256             /* size of buffer for FTP command */
#define FILEBUFSIZE    8192            /* buffer for file/network IO */
#define FTPMAXUSERNAME      32      /* maximum length of a user name */
#define FTPMAXUSERPASS      24      /* maximum length of a password */
#define MAXSENDLOOPS    4   /* maximum file buffers to send at once */
```

2.3.6 Network API

The FTP Server requires a TCP layer on which to run. The interface to TCP is a subset of the standard BSD sockets implementation, designed to avoid the most unportable sockets calls and also to be "map-able" to non-sockets TCP APIs, such as NCSA Telnet. All TCP calls made by the FTP code are prepended with the string **sys_**, so they can be **#define**d to the appropriate Sockets (or other TCP) calls in **ftpport.h**. The following definitions work for InterNiche Sockets. Similar definitions will work for any standard sockets port.

```
/* map sockets calls to InterNiche Sockets library */
#define SOCKTYPE long                   /* type for socket descriptor */
#define sys_closesocket(sock) t_socketclose(sock)
#define sys_send(sock, buf, len, flags) t_send(sock, buf, len, flags)
#define sys_recv(sock, buf, len, flags) t_recv(sock, buf, len, flags)
#define sys_errno(sock) t_errno(sock)
#define SYS_SOCKETNULL      -1L     /* socket error */
```

The semantics of two of the above items vary slightly from the traditional UNIX Sockets. **sys_errno()** is used to get the error value which would be set in the global application parameter **errno** on UNIX. Embedded TCP systems generally use a per-socket error call to perform the **errno** function since embedded systems often don't have a distinct address space for every task like UNIX has. The following definition would map **sys_errno()** to a UNIX-like Sockets system:

```
#define sys_errno(sock)   errno
```

The only system error returned from this call which the FTP Server acts upon is the **EWOULDBLOCK** error - this is required, since the FTP Server is coded to run on either blocking or non-blocking sockets. Thus the value **EWOULDBLOCK** must be defined in the scope of **ftpport.h**, usually by including the appropriate sockets header file.

2.4 ftpport.c - The "glue" Layers

Once you've developed your **ftpport.h** file as described in the previous section, the next step is to code the glue layers. These are the routines which map the generic service requests FTP makes to specific services your target system provides. You may have already handled many of them through **#define** mapping in **ftpport.h**. The rest need to be implemented as minimal layer of C code. In the demo packages most these are collected in the files **ftpsport.c** and **ftpsdos.c**. You can name these files anything you like, or implement these routines in a single port file The two files' names reflect their functions: **ftpssock.c** contains

code mapping the FTP server's generic TCP calls to the subset of Sockets functions described in the previous section. The other, **ftpsdos.c**, implements the FTP server's system-specific functions, primarily making file directories and implementing **user/password** functions on DOS.

2.4.1 TCP glue Routines

The network portion of the glue layers require two simple TCP routines - an **active open**, and a **passive open**, (**listen**). If you are using Sockets for a TCP API, then you can just recompile the supplied **ftpssock.c** file with the **ftpport.h** file you created in the previous section - no new sockets coding is required.

2.4.2 TCP APIs Other Than Sockets

If you port to a non-sockets TCP API, you will need to provide routines which map the FTP server's generic TCP connection open functions to your API. There are two of these functions: **t_tcplisten()** and **t_tcpopen()**. **t_tcplisten()** implements a **passive open** with a callback to the server when a connection is established. **t_tcpopen()** implements an **active open** and is usually used for FTP data connections. These routines are detailed starting on page 20.

2.4.3 System glue Routines

The system **glue** routines need to be defined for each operating system the FTP Server is ported to. These implement the details of file systems and user names which differ from one system to another. The routines are listed below. Detailed information for them begins on page 23.

```
fs_dodir(ftpsvr * ftp, u_long ftpcmd)        /* do DIR or LS cmd */
lslash(char * path)                                /* convert slashes in pathname */
fs_dir(ftpsvr * ftp)                         /* verify directory exists */
fs_permit(ftpsvr * ftp)                      /* verify user permission for cmd*/
fs_lookupuser(ftpsvr * ftp, char * username)      /* lookup user name/password */
```

2.4.4 Timers and Multitasking

Like all InterNiche applications, the FTP Server is designed to be driven by either continuous polling ("superloop") or as a task in an RTOS. In either case, the routine named **ftp_loop()** must be called periodically whenever there is pending work for FTP.

In a polling, or superloop system, (so called because all events are driven by polling from a main **for()** loop somewhere) you simply call **ftp_loop()** as often as you can - it is designed not to block for long periods, and will return rapidly if there is no outstanding FTP work. This approach is very simple to implement, and allows you to provide FTP services on systems with no multitasking capabilities. The downside is that it burns CPU cycles even when there is no pending FTP work.

In a real multitasking system, the FTP Server can be driven by a task which goes to sleep when there is no pending FTP work. Pending work is indicated by then receipt of an FTP command on an FTP **command** socket, or by the existence of an FTP **data** socket. Sample code for this is provided in reference ports to several RTOS systems, and is also available from InterNiche upon request.

The final aspect of multitasking is to protect sensitive structures from being corrupted by code re-entry. This is accomplished by two macros which protect critical sections of code:

```
#define ENTER_CRIT_SECTION();   {_asm{ pushf };  _asm{ cli } }
#define EXIT_CRIT_SECTION();   _asm{ popf };
```

The examples given are for the DOS port, where simply disabling interrupts for a brief period is sufficient. On a true real-time system, these should be mapped to a **mutex**.

2.5 Testing

Once your **ftpport.h** file is set up and your glue layers are coded, compiled, and linked, you are ready to test your FTP Server. Before you start, you'll need to have a valid user and password and access to at least part of the target system's file system.

Perhaps the most common test of our FTP Server is from a Windows 95 FTP client. Just open a DOS box on a connected Windows95 machine, and type "**ftp X.X.X.X<CR>**" (X.X.X.X is the IP address of your target system). Windows 95 will establish a TCP connection to the FTP command port (port 21). The FTP client on Windows 95 should then ask for your user name and password, which you must type in at the keyboard. From there on, the FTP Server will respond to all the FTP commands supported by the Windows 95 client. Most UNIX systems support an FTP client with a very similar look and feel.

After basic connectivity is verified, you will probably want to do some speed testing. To do this, we recommend you use binary mode, and the fastest file device possible. Note that the Windows 95 FTP client supports separate source and destination file names. We recommend using this feature to avoid overwriting existing files. It also allows you to copy a file from one machine to another, then back to the first machine and then do a compare of the two files to be sure the content was not corrupted. The Windows command to compare two files is **fc** with the **/b** option.

Another feature of the FTP Client we recommend using during testing is **hash**. Enabling this causes the client to print a stream of hash marks (…#########…) to the screen as the file is transferred. The marks should appear smoothly and continuously. If they appear in bursts, with pauses in between, this means you are getting less than ideal performance. Usually this is the result of packet loss and/or resource problems.

Another test you should run is the "slowness" test - forcing your FTP Server code to block for long periods waiting for disk IO. FTP ports which run well at blinding speeds often fail this test. The easiest way to stage this sort of test is usually to read-and-write files to a floppy

disk. If available, use a floppy on both the client and the server, as these will create different timing scenarios.

3. TROUBLESHOOTING

In the event the FTP Server port has problems, there are several troubleshooting techniques you can use.

The FTP Server, unlike many networking protocols, is quite amenable to source level debugging with breakpoints. Since the FTP commands are sent from the Windows 95 (or UNIX) client manually, it is easy to set a breakpoint inside **ftp_loop()**'s case statement and follow the action. Many problems quickly become obvious this way.

A Packet Analyzer is another invaluable tool for debugging. These are available as software programs for Windows 95, or as dedicated hardware devices. An analyzer will capture packets on the LAN to which it is attached, and save them for later review. Most support filters, so you can set them to capture only the packets of interest - in this case FTP packets. Older analyzers may only filter at a coarser level, such as all IP packets, or all TCP packets.

USER PROVIDED FUNCTIONS

The functions described in this section must be provided by the porting programmer as part of porting the InterNiche FTP. The DOS demo package can be referenced for examples. If you are using the InterNiche IP stack, all these functions are already provided.

In the demo packages these functions are either mapped directly to system calls via MACROS in **ftpport.h,** or they are implemented in one of the glue layer source files **ftpsport.c** and **ftpsdos.c**

3.1 General Functions

NAME

dtrap()

SYNTAX

void dtrap(void);

DESCRIPTION

This primitive is intended to hook a debugger whenever it is called.

See the detailed description in the Debugging Aids section starting on page 11.

RETURNS

Usually nothing, depends on user modifications.

NAME

dprintf()
ns_printf()

SYNTAX

```
void dprintf(char *, ...);
void ns_printf(char *, ...);
```

DESCRIPTION

These two routines are functionally the same as **printf**. Both are called by the stack code to inform the programmer or end user of system status. **dprintf()** prints error warnings during runtime, and **ns_printf()** is used by the menu routine to display state information.

See the detailed description in the Debugging Aids section starting on page 11.

NAME

ENTER_CRIT_SECTION()
EXIT_CRIT_SECTION()

SYNTAX

```
void ENTER_CRIT_SECTION(void);
void EXIT_CRIT_SECTION(void);
```

PARAMETERS

None

DESCRIPTION

These two primitives should be designed to be paired around sections of code that must not be interrupted or pre-empted. Generally these simply need to disable and re-enable interrupts. On UNIX-like systems they can be mapped to the **spl()** primitive. On Windows DLLs they can be defined to NULL functions since Windows message based system always runs to completion. Examples for embedded Intel x86 processors are provided in the demo. Only the definitions are given here. For examples see the source code.

The stack source code always pairs these two in the same routines. The implementers can push a value on the stack in **ENTER** and retrieve it in the following **EXIT**. The Intel x86 example takes advantage of this to push the existing **flags** register on the stack, saving the interrupt flag state, and retrieves the value for the **flags** register later, restoring the interrupt flag as it was before the **ENTER** call.

3.2 Transport Network API Layer

NAME

t_tcplisten()

SYNTAX

SOCKTYPE t_tcplisten(u_short * lport);

PARAMETERS

Local port (**lport**) is an unsigned 16 bit TCP port value passed in local endian. For the FTP Server this will usually be the FTP command port, 21.

DESCRIPTION

t_tcplisten() will perform a passive open on the port number passed. For FTP the port number will usually be the standard FTP command port: 21. The implementer must also provide logic to accept incoming connections to the **listening** port and call **ftps_connection()** (pg. 21) each time a connection is accepted.

NOTE: Since this routine is called by the user's initialization before the FTP Server can accept connections. See the demo example in **ftpssock.c**

See the Network API section starting on page 13 for more info.

RETURNS

Traditionally this returns a **listening** socket, or **SYS_SOCKETNULL** if error. Since this routine is called by the user's initialization before the FTP Server starts, the use of the return value is up to the user.

NAME

ftps_connection()

SYNTAX

```
int ftps_connection(WP_SOCKTYPE sock);
```

PARAMETERS

Accepts a Socket descriptor, usually (but not always) a 32 bit value. This descriptor will be used later for calls to **send** and **receive** on the connection.

DESCRIPTION

ftps_connection() is called whenever the TCP layer has accepted a connection on the FTP Server command connection passive open routine (see **t_tcplisten()**, pg. 20). The socket passed will stay open until the FTP Server or the client closes it. No further action is required by the caller.

The socket descriptor passed will be used in later calls to **sys_send**, **sys_recv**, etc. Thus it should remain valid in the TCP API until the socket close routine is called for this descriptor.

See the Network API section starting on page 13 for more info.

RETURNS

Returns **0** if OK, **-1** if error occurs.

NAME

t_tcpopen()

SYNTAX

WP_SOCKTYPE t_tcpopen(ip_addr host, u_short fport, u_short lport);

PARAMETERS

ip_addr host	/* 32 bit IP address of host to connect to */
u_short fport	/* 16 bit unsigned TCP port number on other host */
u_short lport	/* 16 bit unsigned TCP port number on our side */

DESCRIPTION

t_tcpopen() is the TCP active open routine. Provided so FTP Server can open an active TCP connection, such as an FTP data connection.

NOTE: All parameters are passed in *local* **endian**.

See the Network API section starting on page 13 for more info.

RETURNS

Returns connected socket if OK, **SYS_SOCKETNULL** on error.

3.3 System Interface Functions

NAME

fs_dodir()

SYNTAX

Int fs_dodir(ftpsvr * ftp, u_long ftpcmd);

PARAMETERS

ftpsvr * ftp - pointer to an FTP control structure, which contains the socket descriptor to write the directory text to.

ftpcmd - the 4 letters of the FTP command encoded into a 32 bit value. This will generally be either: 0x4c495354 (**LIST**) or 0x4e4c5354, (**NLST**). It can be used as a general guide for formatting the directory output for the **dir** or **ls** user command.

DESCRIPTION

fs_dodir() is called to do a **dir** (or **ls** for UNIX users) on the current FTP directory, write the resulting text out to **ftps->datasock**, a descriptor for a socket which is open before this is called. How you do the **dir** is local to this function. The data written to the socket should be plain text, with one filename per line. The FTP client will display the text to the user exactly as you format it here.

Lines should be separated by **CRLF** sequences. See the RFC for formatting details. Any temporary files or buffers created in this process should be cleaned up before you return.

RETURNS

Returns **0** if OK, else **-1** if error.

NAME

lslash()

SYNTAX

void lslash(char * path);

PARAMETERS

char * path - C string with path to convert.

DESCRIPTION

lslash() is needed to format universal (UNIX) slashes '**/**' into local type backslashes '****'. If the FTP server's file system already supports a UNIX like file system, this is a no-op - the code returns without doing anything. On DOS and some other systems, the UNIX like slashes assumed by most FTP clients must be converted to the local DOS backslashes. This is always done in place in the string passed.

The PC-DOS version of this is shown here:

```
lslash(char * path)
{
char * cp;

    for(cp = path; *cp; cp++)
            if(*cp == '/')      /* UNIX slash? */
                    *cp = '\\';     /* convert to DOS slash */
}
```

RETURNS

Returns no meaningful value.

NAME

fs_dir()

SYNTAX

```
bool        fs_dir(ftpsvr * ftp);
```

PARAMETERS

ftp - pointer to an FTP session structure with the desired path contained in the member **filename**.

DESCRIPTION

fs_dir() is called to verify drive/directory passed in **ftp->filename** exists. The entire FTP session structure is passed so that the user can write code to hide the existence of, or prohibit access, to certain directories with certain users.

RETURNS

Returns **TRUE** if directory is available, else **FALSE**.

NAME

fs_permit()

SYNTAX

```
bool        fs_permit(ftpsvr * ftp);
```

PARAMETERS

ftpsvr * ftp - pointer to an FTP session structure with the desired path and filename contained in the member **ftp->filename**. The user name and password are also set in **ftp->user.username** and **ftp->user.password**, and the desired FTP command is in the buffer **ftp->cmdbuf**.

DESCRIPTION

fs_permit() is called by the server to check if the logged in user has permission for a file operation. The implementer may wish, for example to prevent certain users from deleting certain files, or only allow "**anonymous**" users to read files but not write them.

RETURNS

Returns **TRUE** if operation is to be allowed, else **FALSE**.

NAME

fs_lookupuser()

SYNTAX

Int fs_lookupuser(ftpsvr * ftp, char * username);

PARAMETERS

ftpsvr * ftp - pointer to an FTP session structure. If the return value is **0** (success) the strings **ftp->user.username**, **ftp->user.password**, and **ftp->user.home** should be filled in.

char * username - C string with the name passed by the FTP client.

DESCRIPTION

fs_lookupuser() is called by the server when an FTP **USER** command is received. This routine should lookup the user named in **username** in the local users' database. If the name is OK, the routine should fill in the **username**, **password** and **home** directory in the **user** structure in **ftp**. If no password required, fill in a null string. Filled in data should be in an un-encrypted form. Be sure not to exceed the maximum string lengths you set in **ftpport.h**.

The demo package contains a port file for the user database in **\misclib\userpass.c**. This example may be helpful when coding for other systems.

RETURNS

Returns **0** if **user** found, else **-1** if user invalid.

Index