PORTABLE TELNET 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 InterNiche's Telnet Server software . 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's Telnet Server software to a new environment.
If the Telnet code was delivered as part of an InterNiche TCP/IP stack, there is little or nothing to do - the Telnet layers were compiled, linked and tested with the IP stack. This manual is intended primarily as an aid to programmers porting InterNiche's Telnet Server software.

1.1 Terms and Conventions

In this document, the term "stack", when used without other qualification, means the 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 "porting engineer" refers to the engineer who is porting the Telnet code. An "end user" refers to the person who ultimately ends up using the engineer'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()**.

Small samples of source code from C programs is displayed in these boxes:

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

1.2 What Is a Telnet?

1.2.1 Introduction

Telnet is a protocol based on a TCP (Transmission Control Protocol) connection. It is used to transmit data with interspersed Telnet control information. Telnet protocol specification can be found in RFC0854.

The purpose of Telnet protocol is to provide a fairly general, bi-directional, eight-bit byte oriented communications facility. Its primary goal is to allow a standard method of interfacing terminal devices and terminal-oriented processes to each other. It is envisioned that the protocol may also be used for terminal-terminal communications ("linking") and process-process communication (distributed computing).

1.2.2 General Considerations

The Telnet protocol is built upon three main ideas: first, the concept of "Network Virtual Terminal"; second, the principle of negotiated options; and third, a symmetric view of terminals and processes.

When a Telnet connection is first established, each end is assumed to originate and terminate at a "Network Virtual Terminal", or NVT. An NVT is an imaginary device which provides a standard, network-wide, intermediate representation of a canonical terminal. This eliminates the need for "server" and "user" hosts to keep information about the characteristics of each other's terminals and terminal handling conventions.

The principle of negotiated options takes cognizance of the fact that many hosts will wish to provide additional services over and above those available within an NVT. Independent of, but structured within the Telnet protocol are various "options" that will be sanctioned and may be used to allow a user and server to agree to use a more elaborate set of conventions for their Telnet connection.

The symmetry of the negotiation syntax can potentially lead to non-terminating acknowledgment loops. Telnet standard has certain rules to avoid such loops.

1.3 What Is a Telnet Server?

A Telnet server is:"A device that implements Telnet protocol to provide certain services." For example InterNiche's Telnet Server provides the services of its menu interface. We are already familiar with UNIX machines using a Telnet Server (also called as Telnet daemon) to provide the user-level access.

A Telnet Client can make a connection with the Telnet Server to use its services. For example, a Windows 95 Telnet Client could be used to access the menu interface of a device running the InterNiche's Telnet Server.

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 to 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 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 stack 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 Telnet has two kinds of glue routines: the first kind is used to interface to the TCP/IP layer, and the second kind to manage the Telnet databases (user information, etc.).

1.5 Requirements

Before beginning a port, the porting engineer should ensure that the necessary resources are available in the target environment. Here is a brief summary of services InterNiche Telnet 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) (optional)
- Memory as described in Memory Requirements
- And of course, a TCP/IP stack

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 map file. The figures given below reflect the code size (they don't include size of data segment). This program is compiled with Microsoft C 8.0, using default optimization options, with debugging turned off. (Figures are subject to change without notice and are current as of 11/21/00).

Bytes	Description	Related Options
7432	Basic Telnet Server	**TELNET_SVR**
92	Support for description of error messages	**TEL_SHOW_MSG**
928	Support for menus	**TEL_MENU**
544	Support for user authentication	**TEL_USERAUTH**
8952	**Telnet with all features enabled**	

1.5.2 Operating System Requirements

The Telnet Server also requires a few basic services from the Operating System. These are listed here:

clock tick - **tel_check()** needs to be called every time tick to process input from ongoing Telnet sessions.

memory access - Telnet obtains dynamic memory by calls to the primitives **TEL_ALLOC()** and **TEL_FREE()**. These can be mapped directly to the standard **calloc()** and **free()** library calls or they can be mapped to a "partition" based system with very little effort.

1.6 Telnet Source Directories List

When distributed without InterNiche IP, the sources for Telnet are typically sent in a DOS "zip" file, **telsrc.zip**. This file should be unzipped with **pkunzip** (or compatible utility) in such a way as to preserve the underlying directory structure. It contains the sources for the Telnet Server code.

On DOS, the command to unpack the code is:

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

pkunzip should create the following sub-directories:

telnet	InterNiche's Telnet Server Source code	Creates a directory named **\telnet** in your source tree and copies all files there
misclib	InterNiche's sample code for menu interface, **NVRAM**, generic I/O, user authentication, etc.	Creates a directory named **\misclib** in your source tree and copies all files there

You can also request a Windows Telnet demo application from InterNiche Sales or Support. This includes the sources and makefiles for a Windows 95/NT application which serves as both a Telnet demonstration and test tool.

2. STEP BY STEP PORTING GUIDE

2.1 Overview of Porting the Telnet Server

This section describes the steps needed to port the InterNiche's Telnet 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. Provide hooks for the Telnet Server entry point routines (**tel_init**, **tel_cleanup**, and **tel_check**)
3. Create your version of **telport.h** and **telport.c** and compile.
4. Provide your own welcome message and user authentication.
5. Implement your own **tel_exec_cmd** to process the received command.
6. Provide the options that you want to support. This includes the negotiation and sub-negotiation commands.

2.1.1 Coding Conventions

The following conventions are followed in the Telnet source code:

- Boolean variables have the values TRUE or FALSE. Explicit matching should be done in expressions (for example **if (opt->l_value == TRUE)**).
- Functions return a value of **SUCCESS** or error number, thus to do error checking the result of a function call should be explicitly compared with **SUCCESS** (for example, **if (tel_parse(fhost) != SUCCESS)**).

2.2 Source Files List

Before beginning step one, you should be aware of which files in the InterNiche's Telnet 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 heavily modified for different target systems. The following is a list of Telnet source files which should NOT need to be modified in the course of a normal port. If you feel you need to modify one of these files 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.

The portable Telnet source files. These should not be modified.

telnet.c
telparse.c
telmenu.c
telerr.c
telnet.h

The port-dependent files are:

telport.c
telport.h

2.3 Telnet Entry Points

There are three routines which are entry points to the Telnet implementation.

tel_init()	- Should be called after the TCP/IP layer has been initialized.
tel_check()	- Should be called on every time-tick.
tel_cleanup()	- Should be called when the system is being shut down.

If **TELNET_SRV** is defined, then all the Telnet sources get included in the build. So to include the Telnet Server, **TELNET_SRV** should be defined in the **main** header file (or in an appropriate header file).

2.4 Description of Porting Issues - telport.h and telport.c

The description in this section affect the files **telport.h** and **telport.c**.

These files contain most of the port dependent definitions in the stack. CPU architectures (big vs. little endian), compiler idiosyncrasies, and optional features (for example, **TEL_SHOW_MSG**) are controlled in this file. A single mistake in these files (such as getting big & 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.

2.4.1 Standard Macros and Definitions

The InterNiche Telnet source expects **TRUE**, **FALSE**, and **NULL** to be defined within the scope of **telport.h.** The best way to do this is usually to include the standard C library file **stdio.h** inside **telport.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
```

Telnet also uses **SUCCESS** and **FAILURE**. If these are not defined elsewhere in your system, they can be defined as follows:

```
#define SUCCESS 0
#define FAILURE 1
```

2.4.2 Memory Allocation

The Telnet 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 TEL_ALLOC(size) calloc(1,size)
#define TEL_FREE(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 - the **TEL_ALLOC()** can be mapped to a call to allocate the next largest partition size.

2.4.3 CPU Architecture

Four common macros are used from Berkeley UNIX for doing byte order conversions between the representation used in a particular CPU and the CPU independent "network" order. These are **htons()**, **htonl()**, **ntohs()** and **ntohl()**. They may be either macros or functions. They accept 16 & 32 bit quantities as shown and convert them between network format ("big-endian") and the local CPU's format. Most "big-endian" processors, such as Motorola 68K, PowerPC and ARM 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 standard InterNiche stack source code distribution which works for Intel processors implements **htons()** and **ntohs()** as macros, whereas **htonl()** and **ntohl()** are implemented a function calls to the assembly language function **lswap()**, the implementation of which is contained in the file **cksum1.asm**.

```
#define htonl(long_var)    lswap(long_var)
#define ntohl(long_var)    lswap(long_var)
#define htons(short_var)  (((u_short)(s) >> 8) | ((u_short)(s) << 8))
#define ntohs(short_var)  htons(short_var)
```

Depending on your C compiler, it may be more efficient to define inline C macros or inline assembly language implementations of these macros.

```
#define   LITTLE_ENDIAN        1234
#define   BIG_ENDIAN    4321
#define   BYTE_ORDER  LITTLE_ENDIAN
```

In addition to the byte order conversion functions described above, it is necessary to set the value of the defined constant **BYTE_ORDER** to either **LITTLE_ENDIAN** or **BIG_ENDIAN** in order to indicate the byte ordering of the target system processor.

```
#define ALIGN_TYPE 2   /* 16 bit alignment */
```

Some processors will access memory more efficiently if the addresses of the addressed data are evenly divisible by 2 or 4. If the target system processor is of this variety, set the defined constant **ALIGN_TYPE** to either 2 or 4, respectively. **ALIGN_TYPE** effects the memory alignment of allocated packet buffers.

2.4.4 Debugging Aids

dtrap() is a macro called by the Telnet 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 experiment with the exact syntax to get it to compile. The stack code will generally continue executing after a **dtrap()**, but the **dtrap()** usually indicates 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. 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. The **info_printf()** is for printing informational messages, like arrival of a packet, change of metric for a route, etc.

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 info_printf printf   /* same parms as printf, */
#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 as follows:

```
#define info_printf          /* define to nothing */
#define dprintf              /* define to nothing */
```

The last debugging tool in **telport.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 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 */
```

2.4.5 Features and Options

Following is a description of all the **#define** options available for Telnet. The most important being **TEL_INICHE_IP**, which decides whether InterNiche's TCP/IP stack is used or not. If **TEL_INICHE_IP** is disabled, then the porting issues mentioned in this section need to resolved.

TELNET_SRV - Controls the inclusion of all Telnet sources in the build. It should be defined at an appropriate place (for example, in the main header file).

TEL_INICHE_IP - Use InterNiche's TCP/IP stack.

TEL_SHOW_MSG - Show description for error messages and Telnet negotiation options.

TEL_SESS_TIMEOUT - If defined the Telnet Server will close a Telnet session if there has not been any activity for **TEL_IDLE_TIME** seconds.

TEL_MENU - Enable the menu commands. The default implementation provides use of menu commands on InterNiche's command prompt. It is done in **tel_menu_init()**. If menus are to be used with some other architecture, then customization can be done in **tel_menu_init()**.

TEL_USERAUTH - Support for user authentication (login, password).

2.4.6 Transport (TCP) Layer

The supplied Telnet includes code to interface with InterNiche's standard sockets or Microsoft WinSock. You need to map the routines listed below if you have another TCP/IP stack.

sys_closesocket()
sys_send()
sys_recv()
sys_error()
sys_accept()
sys_socket()
sys_bind()
sys_listen()

For example, mappings for InterNiche's TCP/IP stack are as follows

```
#define sys_closesocket         t_socketclose
#define sys_send                t_send
#define sys_recv                t_recv
#define sys_errno               t_errno
#define sys_accept(X,Y,Z)       t_accept(X,Y)
#define sys_bind(X,Y,Z)         t_bind(X,Y)
#define sys_socket(X,Y,Z)       t_socket(X,Y,Z)
#define sys_listen              t_listen
#define SYS_EWOULDBLOCK         EWOULDBLOCK
```

For WinSock, we have the following mappings

```
int sys_send(SOCKET s, char FAR * buf, int len, int flags);
int sys_closesocket(SOCKET s);
#define sys_errno(X)        WSAGetLastError()
#define sys_accep           accept
#define sys_recv            recv
#define sys_bind            bind
#define sys_socket          socket
#define sys_listen          listen
#define SYS_EWOULDBLOCK   WSAEWOULDBLOCK
```

SYS_EWOULDBLOCK is the error code returned by functions to inform that they are blocked (for example **sys_recv()** would return **SYS_EWOULDBLOCK** to inform that it is waiting for some input).

In **tel_init()**, the **listen** socket is set to non-blocking mode. **setsockopt()** or equivalent call must be made to do so by the porting engineer.

2.4.7 Timers and Multitasking

The following functions use the Operating System specific calls for time-ticks. Again, they work for InterNiche's TCP/IP stack and WinSock. They should be modified if you are using any other TCP/IP stack.

tel_start_timer()
tel_check_timeout()

2.5 Telnet Static Data Requirements - NVRAM Parameters

As of this release, the Telnet Server does not have any such requirements.

2.6 Interface Provided by Telnet Server

When a Telnet session gets established, the Telnet Server should provide an interface through which the client can communicate. Telnet Server with InterNiche TCP/IP extends the console menu commands to Telnet. So the Telnet Client will get a prompt and he can type in and execute certain commands.

First look at how the Telnet Server works with InterNiche TCP/IP. This helps illustrate what should be done when porting it to other systems. Consider the following scenario:

1. When a new Telnet session is opened, the Telnet Server sends a welcome message and a prompt.
2. It processes characters from the client until a command has been entered (that is the when the user presses the <Enter> key).
3. It then calls **tel_exec_cmd()** to execute the command. For InterNiche's TCP/IP, **tel_exec_cmd()** maps to **do_command()**.
4. **do_command()** compares the input with its set of commands. If a match occurs, it calls the corresponding function. Otherwise it notifies that the command is not understood.

At the heart of this mechanism is the **GenericIO** structure. One such structure is used for each Telnet session. Here is how the information flows:

1. Telnet Client completes typing a command.
2. Telnet Server, when processing the corresponding session, finds that a complete command has been entered. So it calls **do_command()** with a pointer to the **GenericIO** structure for this session.
3. **do_command()** calls the appropriate function (example: **tel_show_stats**) with the **GenericIO** pointer
4. Using the **GenericIO** structure, **tel_show_stats()** sends all its output to the Telnet Client.

To provide an alternate menu interface via the Telnet Server, the following changes would be required:

1. **tel_exec_cmd()** should be mapped to a function which processes menu commands.
2. All the functions hooked to menu should use the **GenericIO** structure in combination with **ns_printf()** to output to the Telnet Client.

2.7 Setting Telnet Options

When a new Telnet connection is established, the Telnet Server negotiates for certain options with the Telnet Client. InterNiche's Telnet Server comes with few of them like **echo** and **suppress Go Ahead**. To support a new option, the following needs to be done.

1. For example if you want to enter support for **status** option: Add an extra member to the structure **TelOptionList** (end of list). Namely **struct TelnetOption status**.
2. Add the default values for this option to global array **tel_opt_list**.

The following things will now happen automatically.

3. When a new Telnet session is established, it starts with default values of all the options. It negotiates the options and the values are set appropriately.
4. If during a Telnet session, the Telnet Client renegotiates an option, then the values of that option are properly updated.

Using the above, you can implement the processing related to this option.

2.8 Doing Telnet Sub-Negotiation (Expanded Negotiation)

If some particular option requires a richer negotiation structure, it can do sub-negotiation. InterNiche's Telnet Server provides an entry point for such needs. So if sub-negotiation is desired for an option, then the changes can be done in **tel_proc_subcmd()**.

2.9 Telnet User Authentication

InterNiche's Telnet Server provides the option of user authentication. It works as follows:

1. **TEL_ADD_USER()** call is made in **tel_init()** to prepare a list of allowed users.
2. When a new Telnet session is initiated, the user is asked for **login** and **password**.
3. After the user has entered the **login** and **password, TEL_CHECK_PERMIT()** is used to verify them. If they are correct, then a normal Telnet session is started. If they are incorrect, then the user is asked to enter the **login** and **password** again.
4. Telnet Server allows **TEL_MAX_LOGIN_TRIES** (5 by default) tries. After **TEL_MAX_LOGIN_TRIES**, the Telnet connection is closed.

For InterNiche's stack, **TEL_ADD_USER()** and **TEL_CHECK_PERMIT()** map to **add_user()** and **check_permit()** functions (implemented in the **userpass.c** file in **\misclib** directory). If some other implementation for user authentication is used, then functions similar to **TEL_ADD_USER()** and **TEL_CHECK_PERMIT()** should be provided.

2.10 Input Routine for GenericIO Structure

Telnet uses the **GenericIO** structure for input-output. The input routine used for this is **tel_tcp_recv()**, which blocks until it receives a character from the Telnet session. Implementation using InterNiche's TCP/IP stack calls a function **tk_yield()** to do other routine processing, so that processing for other sessions (Telnet, FTP, etc.) doesn't get affected. To port this, an alternative implementation is required which is specific to the target OS.

2.11 Testing

Once your **telport.h** file is set up and your glue layers are coded, compiled, and linked, you are ready to test your Telnet Server. Windows 95 and most UNIX systems come with a Telnet Client. **telnet <*ip addr of server*>** should get you started.

2.12 Telnet User Menu

The Telnet Server comes with portable C code to implement a few simple diagnostic commands on a command line interface. The commands can be invaluable both during debugging of the server and to the end user during configuration and runtime. If you do not implement these menu commands as provided, we strongly suggest that some alternative method (i.e. a GUI) be provided to the end user for accessing the same data.

The menu commands are summarized below:

tshow - Shows OPTION values for all Telnet sessions
tstats - Shows the statistics for all Telnet sessions
logout - Logout from the Telnet session
exit - Logout from the Telnet session

3. GENERIC INPUT-OUTPUT MECHANISM USED BY TELNET

3.1 How Does It Work?

InterNiche's Telnet Server uses the **GenericIO** structure to provide a generic input-output mechanism. Using this structure, general purpose routines can be used for input and output. For example, **ns_printf()** can be used to send data to an output device, **con_page()** can be used to display data one page at a time.
So, we have a calling routine, which populates the **GenericIO** structure and passes it as a reference(pointer) to the called routine. An example of a calling routine is a particular Telnet session.

3.2 What Does the Calling Routine Do?

Let us assume that the calling routine is **X()** and the called routine is **Y().** Now **X()** calls **Y()**. **Y()** expects to know the input (command line), and **Y()** will have certain output. Hence **X()** will pass a pointer to **GenericIO** structure to **Y()** which contains the following information.

1. Pointer to input buffer
2. Pointer to output function (used to send output to some device/network)
3. An identifier representing a socket, file or anything else
4. Pointer to input function (to input a character)

3.3 What Does the Called Routine Do?

1. If input processing is required, it picks the input buffer from the structure and processes it
2. For output, it calls **ns_printf()** with a pointer to the **GenericIO** structure
3. While displaying output, if prompting from the input device is required, it calls the input function in **GenericIO**

3.4 What Does ns_printf Do?

1. Allocates a buffer to hold the output string
2. Forms the output string using **sprintf()** (or equivalent)
3. Uses the function in **GenericIO** structure to output this string
4. Frees the memory allocated to hold the string

3.5 GenericIO Structure

```
struct GenericIO
{
     char * inbuf;                    /* Pointer to command line */
     int (* out)(long id,char *outbuf,int len);
/* Function to send the output string */
     long id ;                        /* Identifier for the IO device. *
/* For TCP connection, it would * represent a SOCKET */
     char (*getch)(long id);
/* Get a character input from the I/O device *
/* This is needed to show scrollable items */
};
typedef struct GenericIO * GEN_IO ;
```

4. PORTING RELATED FUNCTIONS

The functions described in this section must be provided by the porting engineer as part of his porting the Telnet Server. The Windows demo package **WTel** can be referenced for examples. In you are using the InterNiche's IP stack, all these functions are included.
In the demo packages these functions are either mapped directly to system calls via MACROS in **telport.h,** or they are implemented in **telport.c**

4.1 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()
info_printf()

SYNTAX

```
void dprintf(char *, ...);
void info_printf(char *,…);
```

DESCRIPTION

These routines are functionally the same as **printf**. They are called by the stack code to inform the programmer or end user of system status. **dprintf()** prints error warnings during runtime and **info_printf()** is used to display informational messages.

For example, **dprintf()** would be used to display errors and **info_printf()** to display information about processing that happens in the background (i.e. arrival of a packet, change of a Telnet option, etc.).

4.2 Telnet Server Entry Points

NAME

tel_init()

SYNTAX

```
int tel_init (void);
```

DESCRIPTION

Initialize the Telnet Server.

1. Opens a **listen** socket on Telnet port
2. Builds the user, password database
3. Reads values from non-volatile RAM (optional)

RETURNS

Returns **SUCCESS (0)**, else returns a non-zero error code.

NAME

tel_check()

SYNTAX

void tel_check (void);

DESCRIPTION

Does routine Telnet processing. It should be called on every time tick. It does the following:

1. If there is a request for a new connection, bring one up
2. For each of the ongoing Telnet connections, check if any new data has arrived. If yes, process the data
3. Close a connection if it has remained idle for a long time

RETURNS

Nothing

NAME

tel_cleanup()

SYNTAX

void tel_cleanup (void);

DESCRIPTION

Performs cleanup for Telnet. That is:

- Closes all open Sockets
- Frees all allocated memory

RETURNS

Nothing.

4.3 Telnet User Authentication

NAME

TEL_ADD_USER()

SYNTAX

int TEL_ADD_USER(char *username, char *password, void *permissions);

DESCRIPTION

Adds information about a new user to the database.

RETURNS

TRUE if the entry was accepted, otherwise **FALSE**.

NAME

TEL_CHECK_PERMIT()

SYNTAX

```
int TEL_CHECK_PERMIT(char *username, char *password);
```

DESCRIPTION

Authenticates the user. Checks if the username and password are a valid combination. Also verifies the permissions.

RETURNS

TRUE if the entry was validated, otherwise **FALSE**.

4.4 Telnet Timing Routines

NAME

tel_start_timer()
tel_check_timeout()

SYNTAX

void tel_start_timer (unsigned long *timer);
int tel_check_timeout(unsigned long timer, int interval);

DESCRIPTION

Timers are used as follows:

1. A new timer is defined(**u_long sess_timer**)
2. **tel_start_timer(&sess_timer)** is called when the timer is to be started
3. **tel_check_timeout(sess_timer,interval_in_secs)** is called to find out if a timeout has occurred

RETURNS

Nothing.

4.5 Functions Supporting Use of GenericIO Structure

NAME

ns_printf()

SYNTAX

int ns_printf(GEN_IO pio,char * format, ...);

DESCRIPTION

A generic **printf()** routine. It uses information from the **GenericIO** structure to send the output to the appropriate device. The **GenericIO** structure contains a pointer to the function to be used for sending output to the device (e.g. file, or Telnet connection) and an **ID** identifying the specific device. For example, for printing to a file, the function would point to **fprintf** and the **ID** would be handle to file.
So **ns_printf()** does the following:

1. Allocates memory for storing the output string
2. Uses **vsprintf()** (or equivalent) to form the **output** string
3. Uses **GenericIO**'s **output** function to send the data
4. Frees the memory that was allocated for the output string

If **pio** is **NULL**, then output is sent to console.

RETURNS

Number of bytes that were output, or a negative number if error occurred.

NAME

con_page()

SYNTAX

```
int con_page(GEN_IO pio, int lines);
```

DESCRIPTION

Implements a page mechanism. The second argument is the current line number. This function checks if **LINES_PER_PAGE** has been printed. If yes, then wait till you get a character from the input device (**GEN_IO** has a function to get a character from the input device). If the user has pressed **Esc** key or any error has occurred, return **1**, otherwise return **0**.

RETURNS

Returns **1** to stop the display, **0** to continue.

NAME

tel_tcp_send()

SYNTAX

int tel_tcp_send(long s, char *buf,int len);

DESCRIPTION

Sends the contents of **buf** to the Telnet session represented by **s**. This function can be used with the **GenericIO** structure.
This function also provides the translation from **LF** (**\n**) to **CRLF** (**\r\n**). As Telnet needs the **CRLF** sequence, all occurrences of **LF** are replaced by CRLF.

RETURNS

Returns number of bytes sent.

NAME

tel_tcp_recv()

SYNTAX

int tel_tcp_recv(long s, char *buf,int len);

DESCRIPTION

Accepts input from a Telnet session represented by **s**. This function can be used with the **GenericIO** structure.
This function blocks the processing for the particular Telnet session till some input is received. It returns the first character of received input and discards other characters.

RETURNS

The first byte that was received. Return **0** if any error occurred.

APPENDIX A: USING THE WINDOWS DEMO - WTEL

The **WTel** Windows application runs the Telnet Server on a Windows 95 machine and has been built using Microsoft Visual C++ 4.2. It runs the Telnet Server on WinSock and provides MS-DOS prompt interface and contains all the Telnet sources. It can be used to test Telnet or as a demonstration of how the InterNiche Telnet Server works.

To quickly use the **WTel** application,

1. Open the **WTel** project workspace in Microsoft Visual C++ 4.2 (or compatible) environment.
2. Compile and build the executable.
3. Run the executable.
4. From another machine, Telnet into this machine.

WTel is available on request from InterNiche Sales or Support.

Index