InterNiche Menuing System API Technical Reference

51 E Campbell Ave
Suite 160
Campbell, CA. 95008

Copyright © 1998-2005

InterNiche Technologies Inc.
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

Table of Contents

1. INTRODUCTION

The InterNiche Menuing System provides an extensible, portable command line interface ("CLI") for embedded systems. It is highly flexible, modular, and written an ANSI C. It comes pre-integrated with most InterNiche source code products. This document is provided for those users who want to add their own menus and commands to the Menuing System, or even port it to an entirely different environment than those where it's used by InterNiche.

The Menuing System is designed for maximum independence from hardware, assuming only that an ASCII keyboard and character display is available. Since maintains 32 bit handles for IO devices it can be used in a variety of environments. It has been used on serial terminals (such as VT100), telnet sessions, HTTP sessions, and physical hardware (such as a raw PC keyboard and screen). It can handle multiple concurrent users, and can be ported in a few minutes to almost any system which supports **getchar()** and **sprintf()**.

2. SOFTWARE DESIGN

2.1 Input & Output

A keyboard input handler is provided which works via the common C library calls **kbhit()** and **getch()**. Menu commands from other sources, such as telnet sessions and web page CGI scripts can be passed directly to the command parser **do_command()**. **do_command()** directs input & output based on the “Generic IO device” (**GEN_IO**), a small structure which contains an input buffer and output routine. Commands which generate text to the output device can do so via calls to **ns_printf()**, which provides a functionality similar to **fprintf()** except that the output device defaults to the default IO device (the console) if the passed device handle is NULL. Example implementations of all these routines are provided, including **sprintf()** and **printf()** routines for use in systems whose C libraries lack them.

To use the Menuing System from a single simple console like keyboard & display hardware or serial terminal, only **kbhit()**, **getch()**, and **printf()** need to be provided. To invoke the Menuing System from other modules (such as telnet servers and web page CGI engines), a **GEN_IO** structure is created and passed to **do_command()**. **do_command()** will process the text command in the **GEN_IO** input buffer and send the resulting text output to the output routine pointed to by the structure.

2.2 Menus

Commands are added at the Menuing System by creating simple structures with pointers to command text (what you want to call the command), a callback routine (the routine to execute when the command is invokes), and some help text that describes the command. One or more arrays of these structures are created by the programmer, and pointers to the arrays are passed to **install_menu()**. These arrays of commands are referred to as menus.

The first entry in the array MUST be an entry whose command text is used to determine the name of the menu (for help system purposes), and the last entry MUST be NULL.

Here is an example of a small menu declaration:

```
struct menu_op net186_menu[]  =
{
   "net186",       stooges,          "net186 menu",          /* menu ID */
   "uart",         uart_stat0,       "display data uart stats",
   "usetting",     u_setting,        "display uart control struct",
   "uinit",        u_reinit,         "(re)initize UART",
   "baud",         u_setbaud,        "get/set modem UART's baud rate",
   "telnum",       set_number,       "show/set telephone dial info",
   "user",         set_username,     "show/set dial-in user name",
   "pass",         set_password,     "show/set dial-in password",
   "heaps",        mh_stats,         "heap (memory) usage statistics",
   NULL,  /* end of menu */
};
```

Once added via **install_menu()**, the array will be searched for a match each time the end user enters a command. Commands can be abbreviated, but the user must supply enough text so that the command is unambiguous. Parameters to commands may be added after the command text itself, separated by spaces.

Once the menu entry for the typed command has been found int the menus, the callback routine is called by the Menuing System. It is passed the **GEN_IO** structure which contains the invoking command line and the output routine. For commands invoked via the keyboard, a **GEN_IO** structure is supplied by the Menuing System code.

There is no hierarchy to the menu commands – all commands are available to the user at all times. This also means that duplicate and context sensitive commands are not allowed. When a programmer adds a menu care must be taken that all the commands are unique and are not duplicates of the beginning of another command. This means adding a menu with the command “**dir**” to a system that already has the command “**direct**” will result in “**dir**” being unreachable, since it’s a duplicate of the first three letters of “**direct**”. “direct” may still be invoked by typing its full name, or any sequence of its name’s characters longer than “**dir**”.

The Help system is invoked by typing “**?**” or “**help**”. These two strings are considered reserved commands by the Menuing System and their treatment inside the system is identical. Just typing “**help**” (or “**?**”) by itself will invoke the system’s internal “master” menu – the first menu installed in the system.

In response to a “**help**” command, the main menu entries are displayed one per line, first the command string and then the help text. At the bottom on the main menu, the names of the various other menus are displayed in a list separated by vertical bars. The user can display a Help screen for the other menus by typing the name of the menu after “**help**”. For example, a menu named **diagnostics** could be displayed by typing “**help diagnostics**”.

2.3 Format of Menu Entries

Each menu is just an array of struct **menu_op** elements. The struct **menu_op** is defined as follows:

```
struct menu_op
{
   char * opt;                /* the option name */
   int (*func)(void * pio);   /* callback routine */
   char * desc;               /* description of the option */
};
```

The **opt** (option) and **desc** (description) fields are just simple C strings. **opt** is the field which is matched to the user typed command, and **desc** is the one line explanation printed by the help system.

The callback routine that implements the actual command must conform to the filling prototype:

```
int (*func)(void * pio);        /* callback routine */
```

The returned integer should be **0** for success or a negative number for outright failure. These returned values are currently unused but may be used in a future release.

The **pio** parameter is a pointer is a **GEN_IO** pointer. It is cast as a **void *** so that application files and implement menu commands without including the **in_utils.h** file, as long as they use no command line parameters and use **ns_printf()** for output. This allows most menu commands to be written using **pio** as a simple IO descriptor and not having to be aware of it's internals.

For command routines that need to read the command line or input device (**pio->getch**), The structure pointer to by **pio** is as follows:

```
struct GenericIO
{
   char *   inbuf;      /* Pointer to command line  */

   /* Function to send the output string  */
   int (* out)(long id, char * outbuf, int len);

   /* Identifier for the IO device,, e.g. TCP SOCKET */
   long id;

   /* Get a character input from the I/O device This is needed to
    * show scrollable items */
   int (*getch)(long id);
};

typedef struct GenericIO * GEN_IO ;
```

Generally the main use of knowing the format of the **GEN_IO** is to access the command line which invoked the command to extract arguments. The following line of code will extract the first argument in the command line:

```
char * cp =  nextarg( ((GEN_IO)pio)->inbuf );
```

Note: If no argument exists, **cp** will POINT to a Null, '\0', character.
IT WILL NOT BE NULL.

Another interesting use of **GEN_IO** internals is shown in **con_page()**, described in the API Calls section.

2.4 Source Files

The following files contain the source code for the Menuing System:

menus.12c - Core code for managing menus
menu.h - Definitions for menu

In addition, these files contain definitions and routines which may be useful to the Menuing System, including a working implementation of **ns_printf()**:

in_utils.c - InterNiche utility routines required by menus
in_utils.h - Definitions for **GEN_IO**, et. al.

These files are all considered “portable files” by the InterNiche system. This means you should NOT change them when they are part of a larger InterNiche software system. They MAY be safely edited when used in other systems, however this is not usually necessary.

Numerous examples of how to construct and code menu system code abound in the InterNiche sources and can be located by grepping for the keyword “**menu**”. Perhaps the largest example is the file, **misclib\nrmenus.c**.

3. PORT PROVIDED CALLS

When ported to a new environment, the Menuing System needs a few basic calls to perform character IO. Providing these calls is all that's required to port to any environment. Quite often the calls already exist in some form on the target system.

NAME

kbhit()

SYNTAX

int kbhit(void);

DESCRIPTION

Determines if a character is waiting to by returned via **getch()**. This is used by the system to make sure calls from **getch()** do not block the calling thread.

It's vitally important to the menu system's internal logic that this call does not block. On MS-DOS and Microsoft Windows compilers this function is provided as part of the standard C library.

RETURNS

Returns **TRUE** (non-zero) if a character is waiting to by returned via **getch()** and **FALSE** (**0**) if no character is waiting.

NAME

getch()

SYNTAX

int getch(void);

DESCRIPTION

This returns the next unread character typed at the keyboard. Each keyboard character is returned only once.

On standard C libraries this routine will block if no new keyboard character is ready, however since the menu system always polls for received characters with **kbhit()** this should never happen in this context.

Generally the **getch()** primitive in the C compiler can be used by the menu system.

RETURNS

The ASCII value of the last keyboard character.

NAME

ns_printf()

SYNTAX

void ns_printf(void * iodev, char * format, …);

DESCRIPTION

This is functionally the same as the standard C library **fprintf()**, except that the first parameter does not have to be a file descriptor. It is treated as a more generic IO descriptor.

If the IO descriptor is NULL, the output device is the console, other wise the use of this parameter is implementation dependant.

In the InterNiche provided **ns_printf()** implementation, the IO parameter is either NULL (for standard console output), or a pointer to a **GEN_IO** structure. The **GEN_IO** structure contains a pointer to an output routine which has a syntax similar to the standard ANSI **write()** call. This call can then be used to by the **ns_printf()** code to send the formatted output to the correct IO stream, such as a socket or file.

This routine Generally does not need to be re-written - the provided example works on all InterNiche ports.

RETURNS

No meaningful value is returned.

4. API CALLS

The calls listed in this section are those that are implemented in the InterNiche menu system files. They are designed to be called from inside or outside the Menuing System.

NAME

do_command()

SYNTAX

int do_command(void * pio);

DESCRIPTION

do_command() is called when we have a complete command string and wish to have the menu system search for the command and execute it. It is provided as a uniform way for software modules other that the keyboard processor to pass command string into the Menuing System. In InterNiche network systems it has been used by telnet servers and Web servers to make the menu commands available via the network. The command text should be a null-terminated string pointed to by **pio->inbuf**. **do_command()** looks up the command in the menus and executes it. It prints error messages as appropriate.

The pio parameter passed must be a pointer to a filled in **GEN_IO** structure, which if defined in **misclib\in_utils.h**. An example static construction for such a structure is shown here:

```
char cbuf[CBUFLEN];                /* command line buffer. */
extern int std_out(long s, char * buf, int len);
extern int std_in(long s);
extern long IO_Id;                 /* will be set to socket or file */

/* Generic IO structure that do_command() will be called with */
struct GenericIO  std_io = {cbuf, std_out, IO_Id, std_in };
```

Of course the structure may also be allocated and filled in dynamically. **do_command()** will not free any of the members (like **cbuf**). However, it will pass the **pio** pointer to the menu command callback routine which may alter it.

For further information see the Input & Output section starting on page 5.

RETURNS

Returns **0** if the command passed was found in menus (or otherwise understood), returns **-1** if not.

NAME

kbdio()

SYNTAX

void kbdio(void);

DESCRIPTION

Called by the system whenever a console character is ready. Calls **kbhit()** to test, so it can be used for polling. If an **Enter**, ASCII return code (value) 13 or 10, is received, it calls **do_command()** with the currently buffered input characters.

This may be used by system code to poll the keyboard at regular intervals for received characters, or only called when the system knows a keystroke has been entered. It contains code so that it will NOT reenter a command parser, thus ensuring that all commands are serialized.

This does not block itself, however the underlying commands that are called may block.

RETURNS

No meaningful value is returned.

NAME

install_menu()

SYNTAX

int install_menu(struct menu_op * newmenu);

DESCRIPTION

This is called by application code to add new menu to master list. The passed parameter is a pointer to an array of menu items. A discussion of how these arrays are constructed and an illustration is in the Menus section on page 5.

RETURNS

Returns **0** if OK, **-1** if no more spare menu slots are available in the Menuing System.

NAME

con_page()

SYNTAX

int con_page(void * pio, int lines);

DESCRIPTION

This routine can be used to implement a simple paging mechanism so that charts or data dumps which are too large for a single screen size can be "paused" in mid-display until the user hits a character.

When a predetermined number of lines (currently 20 lines) from such a large data dump have been displayed, this routine blocks via the InterNiche **tk_yield()** macro until some input arrives on the **pio->getch()** routine.

If the input is the **ESC**, (escape) key a 1 is returned, allowing the calling routine to abort the continued display.

This should not be used on systems that do not support thread suspension via a macro that conforms to the InterNiche **TK_** macro, **tk_yield().**

Characters in **pio->inbuf** are NOT considered input by this routine. The continuation characters MUST come from the **pio->getch()** routine.

RETURNS

Returns **1** if we got a Break character, currently **ESC**(escape) key, **0** to keep printing.

NAME

nextarg()

SYNTAX

char * nextarg(char * argp);

DESCRIPTION

nextarg() returns a pointer to next **arg** in string passed. Arguments (parameters) are printable ASCII sequences of **char**s delimited by spaces. If string ends without more **arg**s, then a pointer to the NULL terminating the string is returned.

This is useful for extracting multiple arguments separated by spaces in **pio->inbuf** buffers passed to **do_command()** the menu command callback routines.

RETURNS

Returns a pointer to next **arg** in string or a pointer to NULL if no more **arg**s. Note that this does not return a NULL when at the end of the **arg** string.

Index