NIRSPEC

UCLA Astrophysics Program **U.C. Berkeley** **W.M.Keck Observatory**

Tim Liu

February 20, 1997

NIRSPEC Software Programming Note 02.00
Command Line User Interface

1 Introduction

This document describes the implementation of the **NIRSPEC** command line interface (**CLI**). The code structure and some programming issues are discussed in details. In addition, the issues of program building and execution are addressed. All the **CLI** function routines are listed in the end. The reader should refer to NSDN0701 for the **CLI** design outline before reading this programming note.

2 Overview

NSDN0701 provides an overview of the system requirements and structure for the **CLI**. To recap, the key features of this interface program are:

- **Tcl** is used as the command interpreter
- most of the **NIRSPEC** keywords are defined as valid Tcl commands
- built as a stand-alone program, with a socket interface to the **GUI**
- provides non-**Tcl** functions such as script verification and command recall

2.1 Programming environment

The **CLI** is programmed in ANSI C. The "Indian Hill" coding standards are adopted for programming style, function and variable naming convention, and in-code documentation with some Keck SCC amendments. **SCCS** is used for source code control and management. The source code modules, makefile, compiled object files, and executable are located in the directory **/kroot/kui/xnirspec** on the NIRSPEC development computer **crab.astro.ucla.edu** under the user **nirspec**.

2.2 Source modules

All the high-level function routines of the **CLI** program are in the source module **cli.c**. Because the **NIRSPEC GUI** is interfaced with the **CLI** via a UNIX socket, the socket-based communication interface routines used by the **GUI** program are provided by the source module

cli_server.c which is covered by NSPN0100 (Programming note on Graphical User Interface) and therefore will be not discussed in this document. The low-level socket routines used by both the **CLI** and the **GUI** are located in another source module **socket.c**. In addition, **NIRSPEC** commands are defined by a keyword table in the header file **nirspec.h** which is included by most of the **NIRSPEC** source file. Therefore, all the **CLI** code are found in these three source files which are listed below:

cli.c	**- CLI** program
socket.c	**-** low-level socket routines
nirspec.h	**- NIRSPEC** header file that defines keywords, etc.

3 CLI Program

This section describes the high-level source module **cli.c**. The two major routines **main()** and **Cmd_send()**are discussed, while the other routines in the source file are self-explanatory.

3.1 main()

The main program has the following structure:

```
/* Include header files        */
......

/* Declare static variables    */
......

/* Declare private function prototypes */
......

main( int argc, char *argv[] )
{
    ......

    /* Disable control-C             */
    ......

    /* Set up error logging          */
    ......

    /* Get socket name              */
    ......

    /* Open CLI socket server        */
    ......
```

```
    /* Create interpreter            */
    ......

    /* Register commands             */
    ......

    /* Event loop                    */
    while ( !quit ) {
        /* Get command line input    */
        ......

        /* Evaluate command string   */
        ......

            /* Script check          */
            ......

        /* Print evaluation results  */
        ......
    }
}
```

Descriptions are as follows:

1. There are several special header files in addition to the standard UNIX include files listed in the beginning of the code. **<readline/readline.h>** and **<readline/history.h>** are used by the GNU **readline** library routines to implement the command recall function. **<tcl.h>** is the standard **Tcl** include file. The header file **"nirspec.h"** contains the **NIRSPEC** keyword definitions which are used by the **CLI** program. Below is a fragment of the keyword table structure in **nirspec.h**:

```
typedef struct {
    char    *keyword;          /* NIRSPEC keyword name              */
    KTL_DATATYPE datatype;     /* keyword data type                 */
    char    *initval;          /* initial value                     */
    double  minval;            /* mininum value                     */
    double  maxval;            /* maximum value                     */
    int     cli;               /* valid Tcl command if TRUE         */
    int     argnum;            /* # CLI command args; 0 = any       */
    int     cid;               /* transputer command id             */
    int     broadcast;         /* broadcast flag (TRUE or FALSE)    */
    int     rwflag;            /* readable/writable flag            */
} KEYWORD_TABLE;

static KEYWORD_TABLE KeywordTable[] = {
    "telescop", KTL_STRING, "Keck II",                  0,     0,
            TRUE,  0, 0,       TRUE,  RDABLE | WRABLE,
    "observer", KTL_STRING, "UCLA IR Lab team",         0,     0,
            TRUE,  0, 0,       TRUE,  RDABLE | WRABLE,
```

If the flag **cli** is **TRUE**, the corresponding keyword will be defined as a **Tcl** command. The number of arguments, separated by white spaces, followed by a **Tcl** command is determined by the integer **argnum**. For a command argument of a character string, set **argnum** to **0**.

2. The two static integer variables **server_fd** and **client_fd** are socket file descriptors used in socket communication. The integer **ScriptCheck** is a flag. A **TRUE** value indicates the script verification mode is activated.

3. The private function prototype declaration block lists the function routines used in this source module. By our convention, public function prototypes are normally declared in a header file.

4. Control-C is disabled to prevent accidental killing of the program.

5. The **CLI** program uses **syslog()** to log error messages to both the system console and the log file **/var/log/nirspec.log**. This is set by the configuration file **/etc/syslog.conf**. **openlog()** is called in the beginning of the main program to open the log file.

6. The **CLI** and the **GUI** exchange information via a socket which resides in the memory-based directory **/tmp**. The name of the socket is supplied from the command line argument in **main()**. Note that because multiple copies of the user interface program are allowed to run in a computer under the **NIRSPEC** client/server architecture and more than one socket may be created, a fixed socket name is apparently not sufficient. The allocation of sockets is managed by the **GUI** because it is the **GUI** that launches the **CLI** program and passes the socket name to it.

7. Create the **Tcl** command interpreter with the function **Tcl_CreateInterp()**. The new interpreter contains all of the built-in **Tcl** commands. Then register user-defined new commands. The registration process loops through all the keywords in the **KeywordTable** structure and register those flagged in the **cli** field, as shown below:

```
for ( i = 0; i < NUM_KEYWORDS; i++ )
    if ( KeywordTable[i].cli )
        Cmd_register( interp, KeywordTable[i].keyword );
```

Several special new commands are also defined:

```
Cmd_register( interp, "chkscript" );
Cmd_register( interp, "help" );
Cmd_register( interp, "read" );
Cmd_register( interp, "waitfor" );
```

The function **Cmd_register()** invokes the **Tcl** routine **Tcl_CreateCommand()** which contains the function **Cmd_proc()** as a function argument. **Cmd_proc()** calls **Cmd_send()** to perform the command check and conversion, and then send the constructed new command string to the **GUI** which in turn send it to the **NIRSPEC** server for execution. The routine **Cmd_send()** will be discussed in some details later.

8. The heart of the **main()** is a loop which takes a typed-in string from the command prompt, parses it, and prints the evaluation result. The first part of the loop code is command entry. To provide the command recall function, the GNU **readline** library routines are used to handle the command input as shown below:

```
if ( ( cline  = strdup( readline( prompt ) ) ) != NULL ) {
    s = stripwhite( cline );
    if ( s[0] != '\0' )
        add_history( cline );
    strcpy( cmd_str, cline );
    free( cline );
}
```

readline() takes the input command string **cline**, the function **stripwhite()** strips white spaces from the start and end of the string, and **add_history()** puts the command string onto the command history stack.

The next block of the code deals with command parsing. To prevent an accidental exit from **Tcl**, the **Tcl** command **exit** is disabled by not going through the command evaluation. If the input command **cmd** is not **chkscript**, the program will simply call **Tcl_Eval()** to evaluate the command string **cmd_str** as shown below:

```
if ( strcasecmp( cmd, "chkscript" ) != 0 )
    code = Tcl_Eval( interp, cmd_str );
```

However, if the input command is **chkscript**, the flag **ScriptCheck** will be set to **TRUE** to indicate the script check mode. The script file is then checked for existence. Because a script execution is invoked by the procedure name defined in the script file which may not be the same as the script file name, the following code fragment scans through a script file and extracts the procedure name **proc_name**:

```
proc_name_found = FALSE;
while ( fgets( line, 80, fd ) != NULL ) {
    if ( line[0] == '#' )
        continue;
    i = sscanf( line, "%s %s", proc_cmd, proc_name );
    if ( i == EOF || i == 0 )
        continue;
    if ( i == 2 && strcasecmp(proc_cmd, "proc" ) == 0 ) {
```

```
        proc_name_found = TRUE;
        break;
    }
}
```

Note that a script must be “sourced” before execution and this is done by the following code:

```
sprintf( cmd_str3, "source %s", val );
code = Tcl_Eval( interp, cmd_str3 );
```

where **`val`** is the script file name. Then the script evaluation is carried out such that

```
if ( code == TCL_OK )
    code = Tcl_Eval( interp, proc_name );
```

As will be discussed in the next section, the **`NIRSPEC`** commands in a script are only evaluated, but not sent to the server for execution. Therefore, no real actions are taken when one runs **`chkscript`** to verify a script for syntax check.

A message generated by **`Tcl`** or the user code is in the pointer **`*interp->result`** and printed out by the following code:

```
if ( *interp->result != 0 )
    fprintf( stdout, "%s\n", interp->result );
```

3.2 Cmd_send()

The function of **`Cmd_send()`** is to convert a **`Tcl`**-registered **`NIRSPEC`** command into a keyword, check the number of arguments and the value range, construct a special command string, and then send it to the **`GUI`** for further processing. The major code blocks are explained in this section.

1. The first part of the routine is to process the special non-keyword commands as listed below:

```
if ( strcasecmp( argv[0], "help" ) == 0 ) {
    help();
    return TCL_OK;
}
else if ( strcasecmp( argv[0], "waitfor" ) == 0 ) {
    if ( argc != 3 ) {
        strcpy( result, "Error: wrong # args" );
        return TCL_ERROR;
    }
    else {
        wait = TRUE;
        keyword = strdup( argv[1] );
```

```
    }
}
else if ( strcasecmp( argv[0], "read" ) == 0 ) {
    if ( argc != 2 ) {
        strcpy( result, "Error: wrong # args" );
        return TCL_ERROR;
    }
    else {
        read = TRUE;
        keyword = strdup( argv[1] );
    }
}
else
    keyword = strdup( argv[0] );
```

As can be seen, the command **help** will call the function **help()** to display all the valid **NIRSPEC** commands and then return. The command **waitfor** requires two arguments, e.g., "**waitfor go 0**". It sets the **wait** flag and takes the **keyword** string from the command arguments. Similarly, the special command **read** will set the **read** flag and obtain **keyword**. Except for these special commands, the **keyword** string is taken from the command name.

2. The next step is to get the keyword index using the **lookup()** routine

```
if ( ( i = lookup( keyword ) ) < 0 ) {
    strcpy( result, "Error: invalid command/keyword" );
    return TCL_ERROR;
}
```

and check the number of keyword arguments and the keyword value range. However, this check is required only if the field **argnum** in the **KeywordTable** structure is greater than **0** (usually means the argument is a string) and if the command is neither **read** or **waitfor**, as shown by the following code fragment:

```
if ( ( KeywordTable[i].argnum > 0 ) && !read && !wait ) {
    if ( argc != (KeywordTable[i].argnum + 1) ) {
        strcpy( result, "Error: wrong # args" );
        return TCL_ERROR;
    }

    if ( KeywordTable[i].datatype != KTL_STRING ) {
        value = atof( argv[1] );
        if ( value < KeywordTable[i].minval ||
            value > KeywordTable[i].maxval ) {
            strcpy( result, "Error: value is out of range" );
            return TCL_ERROR;
        }
    }
}
```

The **Cmd_send()** routine returns when an error occurs.

3. If the script check mode is activated (**ScriptCheck** is **TRUE**), the function will simply return here:

```
if ( ScriptCheck )
    return TCL_OK;
```

4. The next code block is to construct a special command string that will be sent to the **GUI**:

```
if ( wait ) {
    sprintf( cmd, "wait %s", argv[1] );
    for ( i = 2; argv[i] != NULL; i++ ) {
        strcat( cmd, " " );
        strcat( cmd, argv[i] );
    }
}
else if ( read )
    sprintf( cmd, "read %s", argv[1] );
else {
    sprintf( cmd, "write %s", argv[0] );
    for ( i = 1; argv[i] != NULL; i++ ) {
        strcat( cmd, " " );
        strcat( cmd, argv[i] );
    }
}
```

The command string has a syntax such that "**[read|wait|write] {keyword} {value}**". For example:

```
read coadds
wait go 0
write itime 10
```

5. Finally, **Cmd_send()** sends the constructed command string to the client side of the socket with the following code:

```
if ( send( client_fd, cmd, strlen(cmd)+1, 0 ) == -1 ) {
    perror( "send()" );
    strcpy( result, "Error: failed to send the command out" );
    return TCL_ERROR;
}
else {
    if ( recv( client_fd, reply, sizeof(reply), 0 ) != -1 )
        strcpy( result, reply );

    return TCL_OK;
```

```
}
```

If the string is sent successfully, the routine will wait for a reply before it returns.

4 Low Level Socket Routines

The low level generic socket function routines used in **`cli.c`** and **`cli_server.c`** are contained in the source file **`socket.c`**. These routines are straightforward in programming, and therefore only a brief discussion is given here.

1. **`socket_create()`** creates a socket name. The location of a socket is determined by the environment variable **`NIRSPEC_SOCKET_DIR`** in the **`NIRSPEC`** client software initialization file **`NirspecClientInit`**. Because multiple copies of the **`NIRSPEC`** client program may be running in a computer, a socket is created dynamically with a sequential number as shown below:

```
for ( i = 1; i < 100; i++ ) {
    sprintf( socket_name, "%s/%s%d", dir, prefix, i);
    if ( access( socket_name, F_OK) != 0 )
        break;
}
```

A function call like **`socket_create( "nirspec_cli_socket", socket_name )`** may create the socket **`nirspec_cli_socket5`** if the sequence number **`1`** through **`4`** are already used for other existing sockets.

2. **`socket_serverOpen()`** has a time-out argument to set a time limit in establishing a socket connection. This time-out function is implemented as follows:

```
if ( ( pid = fork() ) == 0 ) {
    ppid = getppid();
    sleep( timeout );
    kill( ppid, SIGKILL );
}
else {
    *client_fd = accept( *server_fd, client_sock_address_ptr, &client_len);
    kill( pid, SIGKILL );
    return 0;
}
```

The routine forks a child process which acts as a timer. If the given time value expires before the timer is stopped by the parent process, the child process will kill its parent and subsequently stop the socket opening.

5 Building the Program

The **NIRSPEC** user interface programs including **CLI** and **GUI** are built with the make file **makefile** in the source code directory **/kroot/kui/xnirspec**. Listed below is a part of the make file that builds the **CLI** program executable **cnirspec**:

```
INCLUDE = /usr/local/dv/include
CC      = cc
CFLAGS  = -g -I$(INCLUDE) -I$(KROOT)/rel/default/include
LIBS2   = -ltcl7.5 -lsocket -lm -ldl -lnsl -lreadline -ltermcap

cnirspec: cli.o socket.o
     $(CC) -o cnirspec cli.o socket.o $(LIBS2)
```

6 Running the Program

Although the **CLI** is a stand-alone program, it is actually launched by the **GUI** because time coordination is required between the two user interface programs to set up the socket channel. When the **GUI** program starts, it invokes the **GUI-CLI** socket interface routine **CLI_serverOpen()** to create a socket and run the **CLI** program in an **xterm** window. The fragment of the code that starts the **CLI** is shown below:

```
sprintf( cmd, "xterm -132 -geometry 69x3+0-7 -fg white -bg black \
         -title \"XNIRSPEC 1.1 - Command Line  (%s@%s)\" \
         -cr white -ms gray -fn 8x13 -fb 8x13bold -sb -sk -ls \
         -e cnirspec %s &", user, host, SocketName );
system( cmd );
```

The first statement creates a system command that sets up an **xterm** window and specifies **cnirspec** to be run in the window. This shell command is then invoked from a **system()** call.

7 List of Function Routines

Functions in the source module **cli.c**:

main()	- main program
Cmd_register()	- register a new command
Cmd_proc()	- command procedure for a new command
Cmd_send()	- check, convert and send a command string
help()	- list commands
lookup()	- look up **NIRSPEC** keyword table index
stripwhite()	- strip white spaces from the start and end of a string

Functions in the source module **socket.c**:

`socket_create()` - create a socket
`socket_delete()` - delete a socket
`socket_clientOpen()` - open socket client
`socket_clientClose()` - close socket client
`socket_serverOpen()` - open socket server
`socket_serverClose()` - close socket server
`socket_read()` - read a string from socket
`socket_write()` - write a string to socket