NIRSPEC

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

Tim Liu

March 12, 1997

NIRSPEC Software Programming Note 04.00
Echelle Format Simulator External Interface

1 Introduction

This programming document describes the implementation of the Echelle Format Simulator (**EFS**) interface to the **NIRSPEC** server. As discussed in the design note NSDN1000 (Interface Between Echelle Format Simulator and **NIRSPEC** Server), the graphical user interface frond-end **EFS** will be implemented as a client program under the **NIRSPEC** client-server architecture. In order for this IDL widget based program to communicates with the server software which is coded in C, an inter-process communication (IPC) mechanism combining a UNIX socket and the IDL CALL_EXTERNAL function has been developed in NSDN1000. The reader should refer to NSPN1000 for details of the design.

2 Overview

The **EFS**-server interface should be capable of handling a high volume of traffic, some of which may be mission-critical like aborting an exposure. Therefore, rather than routing messages through the **GUI**-server link, a direct communication channel is set up to ensure a fast response time, though the former technique is simpler. For the same reason, in contrast to other low traffic and less time-critical IPCs employed in the **NIRSPEC** software such as the **GUI**-**QL** link, the **ESF**-server interface implements a dedicated and stand-alone process in order to manage the bi-directional message flow more efficiently.

The **EFS**-server communication routines are contained in three source modules: **efs_gateway.c**, **efs_server.c**, and **efs_client.c**. In addition, the low-level socket routines used in these source files are from **socket.c** which has been described in NSPN0200 (Programming Note on Command Line User Interface). These source files provide a gateway to the non-IDL server software for **EFS**. The following is brief descriptions of these modules:

efs_gateway.c	- **EFS** gateway main program and related routines
efs_server.c	- socket server routines used by the gateway program
efs_client.c	- socket client routines called by IDL CALL_EXTERNAL function
socket.c	- low-level socket routines

All the source files are located in the **NIRSPEC** client software development directory **/kroot/kui/xnirspec**.

3 Program Description

3.1 efs_gateway.c

When compiled, this module will run as a stand-alone process to handle the communication between the server and **EFS**. This gateway program contains the following routines:

```
void main( int argc, char *argv[] )                 - main program
void EFS_createInterest( KTL_HANDLE *ktl_handle)    - set up keyword broadcast
void EFS_callback( char *keyword,
                   void *user_data,
                   KTL_POLYMORPH *call_data,
                   KTL_CONTEXT *context )           - callback for keyword broadcast
void EFS_parse( int fd, char *cmd )                 - parse command string from EFS
int  lookup( char *keyword )                        - look up KTL keyword table index
```

The source file **efs_gateway.c** includes **"ktl.h"** and **"nirspec.h"** because several **KTL** routines and the **NIRSPEC** keyword table are used inside the program. In addition, the macros **EXPRESS_INTEREST()** and **KTL_DISPATCH()** are defined in the module for **KTL** function calls. These routines are described below:

1. The main function **main()** first checks command line arguments. For the moment, only the simulation switch "**-s**" can be supplied. Like other main functions in the **NIRSPEC** software, **main()** disables **contrl-C** to prevent accidental killing of the program. To communicate with the instrument server, the program makes a connection to the **NIRSPEC** keyword library by calling **ktl_open()**. Callbacks to respond to keyword changes from broadcasting are set up using **EFS_createInterest()**. The program then opens a socket channel to **EFS** with a 20 seconds time-out which allows the socket client enough time to open when **EFS** is launched.

The core of **main()** is the event loop to process **KTL RPC** events and **EFS** socket events. Because the gateway program must be able to handle the two different file I/O sources, an asynchronous I/O multiplexing scheme is implemented for the event loop using the UNIX **select()** function call. **select()** examines an I/O file descriptor sets to see if any of the file descriptors are ready for reading, writing, or have an exceptional condition. A fd set consisting of the **KTL** fd and the **EFS** socket fd is created in the beginning of the loop as follows:

```
FD_ZERO( &readfds );
ktl_ioctl( khand, KTL_FDSET, &readfds );
FD_SET( efs_fd, &readfds );
```

The macro **FD_ZERO()** initializes a file descriptor set to the null set. Note that because the **KTL** call **ktl_ioctl(,KTL_FDSET,,)** automatically clears a fd set, it must be placed before the macro **FD_SET(efs_fd, &readfds)** which includes **efs_fd** in the read fd set **readfds**.

The next code segment in the event loop is to block the process indefinitely until an **EFS** or **KTL** event arrives:

```
if ( (select(maxfds, &readfds, NULL, NULL, NULL) == -1) && (errno != EINTR) ) {
    perror( "select() failed." );
}
else {
    /*
     *  Get input from EFS
     */
    if ( FD_ISSET( efs_fd, &readfds ) ) {
        if ( EFS_serverIO( 0, cmd ) != -1 )
            EFS_parse( efs_fd, cmd );
    }
    /*
     *  Invoke KTL event handler
     */
    else
        KTL_DISPATCH( khand );
}
```

select() returns if either of the two fds is ready for reading. The program calls the macro **FD_ISSET()** to determine which fd is ready, and then invokes either **EFS_serverIO()** and **EFS_parse()** or **KTL_DISPATCH()** to perform the request operation.

When **main()** exits, the socket channel to **EFS** and the **RPC** connection to the **NIRSPEC** server are closed with **EFS_serverClose()** and **ktl_close()**.

2. When the gateway program receives a keyword which is sent from the **NIRSPEC** server via broadcast, a user-defined callback function will be invoked by **KTL_DISPATCH()** to send this keyword to **EFS** through the socket link. Whether the program should respond to a **NIRSPEC** keyword broadcast from the server is defined by **EFS_createInterest()** using the defined macro **EXPRESS_INTEREST**:

```
for ( i = 0; i < NUM_KEYWORDS; i++ ) {
    EXPRESS_INTEREST(  KeywordTable[i].keyword, EFS_callback );
}
```

where **EFS_callback()** is the callback routine. By definition, any keyword broadcast will invoke a callback in the gateway program that will forward this keyword to **EFS**.

The callback function first determines a keyword index using **lookup()** which looks up the **NIRSPEC** keyword table and then constructs a keyword string as follows:

```
switch ( KeywordTable[i].datatype ) {
    case KTL_INT:
    case KTL_BOOLEAN:
        sprintf( str, "%s %d", keyword, call_data->i );
        break;
    case KTL_DOUBLE:
        sprintf( str, "%s %f", keyword, call_data->d );
        break;
    case KTL_STRING:
        sprintf( str, "%s %s", keyword, call_data->s );
        break;
}
```

The constructed string **str** is sent to **EFS** by:

```
EFS_serverIO( str );
```

3. On the other hand, when the gateway program receives a command string from **EFS**, it calls **EFS_parse()** to convert the string into a keyword/value pair and then send it to the **NIRSPEC** server. **EFS_parse()** first breaks the string into tokens separated by spaces using **strtok()**. The first token is the keyword:

```
keyword = strdup( strtok( cmd, " " ) );
```

The keyword value **data** comes from the second token (or the rest of the command string if the keyword has a string type):

```
switch ( KeywordTable[i].datatype ) {
    case KTL_INT:
    case KTL_BOOLEAN:
        strcpy( value, strtok( NULL, " " ));
        if ( value != NULL )
            data.i = atoi( value );
        break;
    case KTL_DOUBLE:
        strcpy( value, strtok( NULL, " " ));
        if ( value != NULL )
            data.d = atof( value );
        break;
    case KTL_STRING:
        str = strtok( NULL, "" );
        if ( str != NULL )
            strcpy( value, str );
```

```
        else
            *value = NULL;
        data.s = strdup( value );
        break;
}
```

Finally, the keyword and its value are sent to the server via the **RPC** link using **ktl_write()**. A status message is replied to **EFS** using the socket I/O call **EFS_serverIO()**.

3.2 efs_server.c

This source file contains routines to provide server-side socket communications with **EFS**. These routines are similar to those in **ql_server.c** which has been described in NSPN0300 (Programming Note on Quick Look External Interface). The reader should consult with the design note for descriptions. In the future, **efs_server.c** and **ql_server.c** will be merged into a single source module.

3.3 efs_client.c

There're also many similarities between the socket client routines in **efs_client.c** and those in **ql_client.c**. Therefore, no separate description is given here. The reader should refer to NSPN0300 for discussion. Again, **efs_client.c** and **ql_client.c** will be combined into a single file.

4 Program Compiling

All the source code in the **NIRSPEC** client software directory **/kroot/kui/xnirspec** is compiled using the make file **makefile**. The **EFS** gateway program executable **efs_gateway** is built from the three source files **efs_gateway.c**, **efs_server.c**, and **socket.c** as shown by the following lines in **makefile**:

```
CC    = cc
CFLAGS  = -g -I$(INCLUDE) -I$(KROOT)/rel/default/include

NAMES2  = efs_gateway efs_server socket
SOURCE2 = $(NAMES2:%=%.c)
OBJECT2 = $(NAMES2:%=%.o)
LIBS3   = -L$(KROOT)/rel/default/lib -lktl -lktlker -lkcl -ldl -lsocket \
          -lucb -lm

TARGET  = xnirspec cnirspec ql_client.so efs_gateway efs_client.so
all: $(TARGET)

# Build EFS gateway program
efs_gateway: $(OBJECT2)
```

```
	$(CC) -o efs_gateway $(OBJECT2) $(LIBS3)
```

The socket client routines in **efs_client.c** are built as a shareable object library named **efs_client.so** that can be invoked by the CALL_EXTERNAL call:

```
# Build EFS socket client sharable object library
efs_client.so: efs_client.o
	ld -G -o $@ efs_client.o socket.o
```

5 Program Execution

The **NIRSPEC** user interface client software starts by executing the shell script file **xnirspec.sh** in **/kroot/kui/xnirspec**. The stand-alone **EFS** gateway program **efs_gateway** is launched by the scrip as follows:

```
if ( !($noefs) ) then
    if ( !($simulate) ) then
        exec ./efs_gateway &
    else
        exec ./efs_gateway -s &
    endif
endif
```

where **noefs** and **simulate** are two flags passed from the **xnirspec.sh** command line. For example, the command entry "**xnirspec.sh -noefs**" will start the **NIRSPEC** client program without running **EFS**. Similarly, the switch "**-s**" indicates the simulation mode is activated.

The socket server routines in **efs_server.c** are called by the **EFS** gateway program. For example, **EFS** establishes the socket connection when it starts:

```
if ( ( efs_fd = EFS_serverOpen( 20 ) ) < 0 )
    ERROR( ("Aborted: failed to open socket connection to EFS.\n"));
```

The client routines contained in the shareable object **efs_client.so** are called from IDL programs using the CALL_EXTERNAL function. For example, the following IDL code opens a socket client by calling **EFSCom_open()**:

```
inp = strarr(2)
inp(0) = ' '
inp(1) = ' '
status = call_external('/kroot/kui/xnirspec/efs_client.so','EFSCom_open', $
                       inp, n_elements(inp), /f_value)
```

The string array **inp** contains parameters to be passed to the called function **EFSCom_open()**. The CALL_EXTERNAL function call returns the value **status**. A zero value indicates a success

as defined in **EFSCom_open()**. The following IDL statements call the routine **EFSCom_io()** to read the socket channel:

```
inp = strarr(2)
inp(0) = '0'
inp(1) = ' '
msg = call_external('/kroot/kui/xnirspec/efs_client.so','EFSCom_io', $
                    inp, n_elements(inp), /s_value)
```

If **inp(0)** = **'1'**, a socket write will be performed by **EFSCom_io()**.