NIRSPEC

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

James Larkin based on 6.00 by Tim Liu **March 3, 1999**

NIRSPEC Software Programming Note 06.01
Keyword Library

1 Introduction

This document describes the implementation of the NIRSPEC keyword library. The reader should refer to NSDN0600 for design outlines and a preliminary list of NIRSPEC keywords.

2 Overview

The NIRSPEC keyword library is an interface layer between the KTL applications such as the GUI and CLI and the underlying instrument server. The connectivity is provided by the RPC message system. The keyword library provides the following functions:

- connect to RPC server
- control over keyword library
- read a keyword from server (one-shot)
- write a keyword to server
- broadcast a keyword from server (continuous read)
- disconnect from server

The keyword library routines are in the source module **`nirspec_keyword.c`**. The RPC interface routines called in **`nirspec_keyword.c`** are contained in **`nrpc_face.c`**. In addition, the following special header files are used by the keyword library source file:

`ktl.h`	– generic KTL definitions
`ktl_keyword.h`	– keyword style specific KTL definitions
`nirspec.h`	– NIRSPEC specific definitions
`nrpc.h`	– RPC protocol definitions
`error.h`	– error code and message definitions

Except for **`ktl.h`** and **`ktl_keyword.h`** which are provided by the **`/kroot/ktl`** structure, all the other source modules are located in the server directory **`/kroot/kss/nirspec/keyword`**.

The NIRSPEC keyword library has been developed on the basis of keyword libraries for existing Keck systems.

3 Program Description

3.1 Header file “nirspec.h”

The header file **"nirspec.h"** contains NIRSPEC keyword definitions. The keywords are declared in the data structure array **KeywordTable[]** as shown below:

```
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 CLI 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           */
    int         access;                 /* Keyword classification */
    char        units[80];              /* Units field */
    char        help[80];               /* single line help */
    char        alias[80];              /* command alias */
} KEYWORD_TABLE;

static KEYWORD_TABLE KeywordTable[] = {
    "telescop", KTL_STRING, "Keck II",                  0,     0,
                TRUE,  0, 0,
                TRUE,  RDABLE | WRABLE, 0, “”,
                “Set telescope name”,
                “Set telescope name”,
    "observer", KTL_STRING, "UCLA IR Lab team",         0,     0,
                TRUE,  0, 0,
                TRUE,  RDABLE | WRABLE, 0, “”,
                “Set observer’s name”,
                “Set observer’s name”,
    "outdir",   KTL_STRING, "/kroot/data/spec/",        0,     0,
                TRUE,  1, 0,
                TRUE,  RDABLE | WRABLE, 0, “”,
                “Set SPEC data directory”,
                “Set SPEC data directory”,
......
};
#define NUM_KEYWORDS ( sizeof(KeywordTable) / sizeof(KEYWORD_TABLE) )
```

The different fields in the structure are explained by the comments lines in **KEYWORD_TABLE**. To add a keyword to the table, create a new entry by filling out the fields in the structure and recompile the server program. You may also need to recompile the client programs so that the CLI and EFS programs that require knowledge of keywords can be updated as well.

Another important definition in the header file is the RPC service handle **HANDLE** which is used in both **nirspec_keyword.c** and **nrpc_face.c**. **HANDLE** is declared by the following data structure:

```
tyepdef struct handle {
    char        *server_host;          /* RPC server host name             */
    char        *client_host;          /* RPC client host name             */
    char        *client_user;          /* RPC client user login            */
    CLIENT      *client;               /* RPC client handle                */
    SVCXPRT     *transport;            /* broadcast service transport      */
    int         prognum;               /* broadcast program number         */
    fd_set      fdset;                 /* file descriptor set              */
} HANDLE;
```

3.2 Routines in nirspec_keyword.c

This section only discusses those important function routines in the keyword library module.

int keyword_open(char *service, char *style, int flag, HANDLE **handle):

This routine opens a connection to the RPC message system. It first calls **rpc_clientOpen()** in the RPC interface routine module **nrpc_face.c** to create a RPC client handle as shown below:

```
if ( ( status = rpc_clientOpen( (*handle)->server_host,
                                &(*handle)->client ) ) < 0 ) {
    free( (*handle)->server_host );
    free( *handle );
    return status;
}
```

keyword_open() will return if the call to **rpc_clientOpen()** returns a negative status code which indicates a failure in creating the RPC client handle.

The routine then obtains the host domain name from **get_hostname()** in **nrpc_face.c** and the user login from the environment variable **USER** as seen below:

```
if ( get_hostname( client_host ) < 0 ) {
    status = CLNT_ERR_NO_HOSTNAME;
```

```
        syslog( LOG_ERR, ClntErrMsg[-status - CLNT_ERR_CODE0] );
        return status;
    }
    (*handle)->client_host = strdup( client_host );

    if ( (client_user = (char *)getenv( "USER" ) ) == NULL )
        client_user = strdup( "unknown" );
    (*handle)->client_user = strdup( client_user );
```

keyword_open() also checks the access permission of the RPC client by calling **rpc_clientAcess()** in **nrpc_face.c** which finds the access information from the RPC server:

```
if ( ( status = rpc_clientAccess( *handle ) ) < 0 )
    return status;
```

Finally, this opening routine sets up RPC broadcast function by calling the RPC interface routine **rpc_broadcastSetup()** in **nrpc_face.c**:

```
if ( ( status = rpc_broadcastSetup( &(*handle) ) ) < 0 )
    return status;
```

rpc_broadcastSetup() creates a service transport handle, registers the handle with the portmapper, and creates a client handle on the RPC server side.

int keyword_ioctl(HANDLE *handle, int flags, int arg1, int arg2, ...):

This routine controls various aspects of the interactions of the keyword library with the message system. The integer **flags** defines an operation to perform and **arg1**, **arg2**, **...**, are flag-dependent arguments. The only operation used by the NIRSPEC keyword library is to return the mask of file descriptors. The flag for this operation is **KTL_FDSET** and the function returns the RPC fd set.

int keyword_read(HANDLE *handle, int flags, char *object_name, KTL_POLYMORPH *call_data):

This routine reads a keyword. This is done by calling the function **rpc_read()** in **nrpc_face.c** as shown below:

```
retval = rpc_read( handle->client, object_name, call_data );
```

where **object_name** is the keyword to be read and **call_data** is the structure containing the data to be received. The interface function **rpc_read()** calls the server remote procedure **rpc_read_1()** in **nrpc_svc_proc.c** to perform the keyword read function.

int keyword_write(HANDLE *handle, int flags, char *object_name, KTL_POLYMORPH *call_data):

This function writes a keyword by calling the function routine **rpc_write()** in **nrpc_face.c**:

```
retval = rpc_write( handle->client, object_name, call_data );
```

where **object_name** is the keyword and **call_data** is the structure containing the data to be written. **rpc_write()** calls the remote procedure **rpc_write_1()** on the RPC server to perform the keyword write function.

int keyword_close(HANDLE *handle):

This keyword function disconnects the keyword library from the RPC message system. It first calls **rpc_broadcastClose()** in **nrpc_face.c** to close RPC broadcast and then destroys the RPC client handle by using the RPC function **clnt_destroy()**.

What **rpc_broadcastClose()** does is to first call the server remote procedure **rpc_closebroadcast_1()** in **nrpc_svc_proc.c** to do the job and then destroy the broadcast server handle. This is illustrated from the following code:

```
/*
 *  Call remote procedure on the server
 */
status_res = rpc_closebroadcast_1( &arg, handle->client );
if ( status_res->status < 0 )
    syslog( LOG_ERR, status_res->errmsg );

/*
 *  Destroy the broadcast server handle
 */
if ( handle->transport != NULL ) {
    ( void ) pmap_unset( handle->prognum, RPCB_VERS );
    ( void ) svc_destroy( handle->transport );
}
```

int keyword_event(HANDLE *handle, int *nevent, KTL_EVENT_DESCR **pdescr):

This routine returns descriptions of events which are keyword broadcasts from the RPC server and invokes callback functions. The argument **nevent** indicates the number of events occurred and **pdescr** is the event descriptor, a data structure containing various information on an event. **keyword_event()** receives a RPC broadcast client request and calls the RPC broadcast dispatch routine as shown below:

```
while ( !done ) {
    switch ( select( maxfd, &handle->fdset, NULL, NULL, &timeout ) ) {
      case -1:
```

```
            if ( errno == EINTR )
                continue;
            return -1;
            break;
          case 0:
            return 1;
            break;
          default:
            svc_getreqset( &handle->fdset );
            done = TRUE;
            break;
        }
    }
```

The dispatch routine **rpc_broadcast_prog()** is in **nrpc_face.c**. This routine decodes the function argument passed by the client, fills in the event descriptor and adds the new entry to the event list.

4 Library Building

The keyword library routines in **nirspec_keyword.c** are built as a shareable object library named **libnirspec_keyword.so.0.0** by **makefile** in the server directory **/kroot/kss/nirspec**. Below is part of the make file that builds the keyword library:

```
APP    = nrpc
CC     = cc -DNIRSPEC_DEBUG
CFLAGS = -g -I$(KROOT)/rel/default
LIBS   = -lnsl -lc -lm -lucb \
         /kroot/rel/default/lib/libktl.so.0.0 \
         /kroot/rel/default/lib/libktlker.so.0.0

TARGET = libnirspec_keyword.so.0.0 nirspec_server clean
all: $(TARGET)

# Build keyword sharable library
libnirspec_keyword.so.0.0: nirspec_keyword.o $(APP)_face.o $(APP)_clnt.o
$(APP)_xdr.o
        ld -G -o $@ nirspec_keyword.o $(APP)_face.o $(APP)_clnt.o \
        $(APP)_xdr.o $(LIBS)
```

The keyword library built above is for the real instrument server. A separate shareable library is also required for the simulation server. This object library is called **libnirspecsim_keyword.so.0.0** and built by the simulation server make file **makefile_sim** in the same directory. In fact, **libnirspecsim_keyword.so.0.0** and **libnirspec_keyword.so.0.0** are identical except for different names.