Tim Liu modified by James Larkin **March 21, 1999**

NIRSPEC Software Programming Note 07.01
RPC Server

1 Introduction

This programming note describes the implementation of the RPC server. The reader can consult the document NSDN1100 for the design of the NIRSPEC client-server architecture.

2 Overview

The NIRSPEC server code is located in the directory **`/kroot/kss/nirspec/keyword`**. The source code includes header files, a RPC protocol definition file, and C source modules, as listed below:

`nirspec.h`	- NIRSPEC server definitions
`nrpc.h`	- RPC protocol definitions
`error.h`	- error code and message definitions
`nrpc.x`	- RPC protocol
`nrpc_face`	- KTL-RPC interface routines
`nrpc_server.c`	- RPC server program
`nrpc_svc_proc.c`	- RPC remote procedures
`nrpc_svc.c`	- RPC server stub (not used)
`nrpc_clnt.c`	- RPC client stub
`nrpc_xdr.c`	- RPC data format exchange routines
`fileio.c`	- data I/O routines
`motor.c`	- step motor routines

In addition, the generic KTL definition header file "**`ktl.h`**" is used by server routines.

3 Program Description

3.1 RPC protocol definition `nrpc.x`

The RPC protocol **nrpc.x** is compiled by the RPC utility program **rpcgen** to generate the RPC protocol definition header file **nrpc.h**, the RPC server stub **nrpc_svc.c**, the client stub **nrpc_clnt.c**, and data format exchange file **nrpc_xdr.c**. The protocol definition file is coded in RPCL, the protocol description language. It defines a data type, the union **rpc_param**, which corresponds to the KTL data type **KTL_POLYMORPH**, as shown below:

```
union rpc_param switch( param_type type ) {
case RPC_INT:           int      i;
case RPC_FLOAT:         float    f;
case RPC_DOUBLE:        double   d;
case RPC_STRING:        string   s<>;
case RPC_INT_ARRAY:     int      ia<>;
case RPC_FLOAT_ARRAY:   float    fa<>;
case RPC_DOUBLE_ARRAY:  double   da<>;
};
```

nrpc.x also defines the argument type **rpc_arg** and the two result types of the remote procedures **rpc_status_res** and **rpc_val_res**. The argument type is a structure consisting of a keyword and its value. The return structure **rpc_status_res** contains a status code and a error message, while **rpc_val_res** combines **rpc_arg** and **rpc_status_res**.

The program definition part of **nrpc.x** is listed below:

```
program RPC_PROG {
    version RPC_VERS {
        rpc_status_res RPC_ACCESS(rpc_arg)         = 1;
        rpc_status_res RPC_SETBROADCAST(rpc_arg)   = 2;
        rpc_status_res RPC_CLOSEBROADCAST(rpc_arg) = 3;
        rpc_val_res    RPC_READ(rpc_arg)           = 4;
        rpc_status_res RPC_WRITE(rpc_arg)          = 5;
    } = 1;
} = PROGNUM;

#ifdef RPC_HDR
%#ifdef SIMULATION
%#define PROGNUM                  0x20000500
%#define RPCB_PROG      ((u_long) 0x20002000)
%#define RPCB_VERS      ((u_long) 1)
%#define RPC_BROADCAST  ((u_long) 1)
%#else
%#define PROGNUM                  0x20001000
%#define RPCB_PROG      ((u_long) 0x20005000)
%#define RPCB_VERS      ((u_long) 1)
%#define RPC_BROADCAST  ((u_long) 1)
%#endif
#endif
```

Because the simulation server and the real instrument server can run at the same time, their program definitions are different.

3.2 Header files

3.2.1 `nirspec.h`

The header file "**`nirspec.h`**" defines transputer command id as shown below:

```
#define CID_GO_SPEC                     1
#define CID_ABORT_SPEC                  2
#define CID_FRAME_READY_SPEC            3
......
```

The transputer command id's are mapped to NIRSPEC keywords by the keyword definition structure **`KeywordTable`**.

The header file contains NIRSPEC keyword definitions. The keywords are defined 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    *help;                  /* single line help                 */
} KEYWORD_TABLE;

static KEYWORD_TABLE KeywordTable[] = {
    "telescop", KTL_STRING, "Keck II",              0,     0,
            TRUE,  0, 0,
            TRUE,  RDABLE | WRABLE, 0,
            "set telescope name",
    "observer", KTL_STRING, "UCLA IR Lab team",     0,     0,
            TRUE,  0, 0,
            TRUE,  RDABLE | WRABLE, 0,
            "set observer's name",
    "outdir",     KTL_STRING, "/kroot/data/spec/",  0,     0,
            TRUE,  1, 0,
            TRUE,  RDABLE | WRABLE, 0,
            "set SPEC data directory",
```

```
    "rootname",   KTL_STRING, "test",                 0,     0,
            TRUE,  1, 0,
            TRUE,  RDABLE | WRABLE, 0,
            "set SPEC image file root name",
    "filenum",    KTL_INT,  "1",                      0,  9999,
            TRUE,  1, 0,
            TRUE,  RDABLE | WRABLE, 0,
            "set SPEC image file running number",
    ......
};
#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 that require knowledge of keywords.

The header file also defines instrument component configuration tables such as:

```
static char *sampmode_table[3] = {
    "Single",
    "CDS",
    "MCDS",
};
#define NUM_SAMPMODES ( sizeof(sampmode_table) / sizeof(sampmode_table[0]) )

static char *filter_table[12] = {
    "Blank",
    "z",
    "J",
    "H",
    "K'",
    "K",
    "JH",
    "HK",
    "3.0",
    "3.5",
    "4.0",
    "4.5",
};
#define NUM_POS_FILT ( sizeof(filter_table) / sizeof(filter_table[0]) )
```

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:

```
typedef 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;
```

In addition, two other important structure templates, the observing parameter table **OBS_PARAM** and the Fits header table **FITS_TABLE**, are defined as listed below:

```
typedef struct {
   char   outdir[80];               /* file name                      */
   char   filename[21];             /* file name                      */
   char   object[21];               /* object name                    */
   char   comment[21];              /* comment                        */
   char   filter[6];                /* filter                         */
   double itime;                    /* integration time               */
   int    coadds;                   /* number of coadds               */
   int    sampmode;                 /* sampling mode                  */
} OBS_PARAM;

typedef struct {
    char        *keyword;           /* Fits keyword                   */
    KTL_DATATYPE datatype;          /* keyword data type              */
    KTL_POLYMORPH value;            /* keyword value                  */
    char    *format;                /* keyword value format           */
    char        *comment;           /* comment                        */
} FITS_TABLE;
```

3.2.2 error.h:

The header file "**error.h**" defines error codes and error messages as shown by the following code segment:

```
/*
 *  Server error code
 */
#define SERV_ERR_CODE0                                  0
#define SERV_ERR_NO_SERVERDIR_VAR         -(SERV_ERR_CODE0 + 1)
#define SERV_ERR_NO_TMPFSDIR_VAR          -(SERV_ERR_CODE0 + 2)
#define SERV_ERR_NO_CFGFILE_VAR           -(SERV_ERR_CODE0 + 3)
......

/*
 *  Server error messages
 */
static char *ServErrMsg[] = {
    "",
    "Error: the environment variable NIRSPEC_SERVER_DIR not defined.\n",
```

```
    "Error: the environment variable NIRSPEC_TMPFS_DIR not defined.\n",
    "Error: the environment variable NIRSPEC_CONFIG_FILE not defined.\n",
    ......
};
```

3.3 Routines in `nrpc_face.c`:

Routines in **`nrpc_face.c`** provide an interface between the KTL layer and the RPC server. Those important ones are briefly described below:

`int rpc_clientOpen( cgar *serfver, CLIENT **client ):`

This routine creates the RPC client handle **`client`** using the RPC function call **`clnt_create()`** such that

```
*client = clnt_create( server, RPC_PROG, RPC_VERS, "tcp" );
```

`int rpc_clientAccess( HANDLE *handle ):`

This routine checks RPC client access permission from the remote procedure **`rpc_access_1()`** in **`nrpc_svc_proc.c`**.

`int rpc_broadcastSetup( HANDLE **handle ):`

This routine sets up RPC broadcast. It first creates a service transport handle from the RPC function **`svctcp_create()`**. Before registering the handle with the portmapper using **`svc_register()`**, it checks for an available port by incrementing the base port defined in the RPC protocol file **`nrpc.x`**. A maximum of 100 ports are available. The routine then creates the client handle on the server side by calling **`rpc_setbroadcast()`** in the same source module.

`int rpc_setbroadcast( HANDLE *handle ):`

This routine sets up monitor RPC ("follow-up" RPC or "reverse" RPC) by calling the remote procedure **`rpc_setbroadcast_1()`**.

`rpc_broadcast_prog( svc_req *rqstp, SVCXPRT *transp ):`

This is the RPC broadcast dispatch routine.

`int rpc_broadcastClose( HANDLE *handle ):`

This routine calls the remote procedure **`rpc_closebroadcast_1()`** in **`nrpc_svc_proc.c`** to close RPC broadcast.

`int rpc_read( CLIENT *client, char *keyword,`
`KTL_POLYMORPH *data ):`

This function routine reads a keyword from the RPC server via the RPC remote procedure **rpc_read_1()** in **nrpc_svc_proc.c**.

int rpc_write(CLIENT *client, char *keyword,
KTL_POLYMORPH *data):

This function calls the remote procedure **rpc_write_1()** to write a keyword to the RPC server.

3.4 Routines in nrpc_svc_proc.c:

nrpc_svc_proc.c contains RPC server remote procedure functions. Again, those important routines are briefly described below:

status_res *rpc_access_1(rpc_arg *arg):

This routine checks the access permission of the client. It opens the authorization file **Authorize** and compares the client's user and host names with those listed in the authorization file. If they match, a zero value is returned by the function. Otherwise, a negative value is returned.

rpc_status_res *rpc_setbroadcast_1(rpc_arg *arg):

This routine sets up RPC broadcast. It calls the RPC function **clnt_create()** to create a client handle on the server side. The client's login information is maintained by a linked list.

rpc_status_res *rpc_closebroadcast_1(rpc_arg *arg):

This routine closes RPC broadcast by finding the client handle from the client list and then closing it.

rpc_val_res *rpc_read_1(rpc_arg *arg):

This remote procedure reads a keyword value from the keyword buffer **kvalue[]**.

rpc_status_res *rpc_write_1(rpc_arg *arg):

This routine writes a keyword to the server. If the keyword has a corresponding non-zero transputer id, it will be sent over the TSP link. Otherwise, it will be processed locally. The keyword will be broadcast if the broadcast flag is set.

int rpc_broadcast_1(rpc_arg *arg):

This routine performs RPC broadcast by making a one-way call to all "follow-up" RPC servers.

3.5 Routines in nrpc_server.c:

`main():`

This is the main function of the RPC server program. It has the following function flow:

- Check arguments
- Disable ctrl-C
- Set up error logging
- Create server handles for the transports "netpath"
- Open TSP link
- Initialize keywords from instrument configuration auto-backup file or default values
- Initialize some transputer cid values
- Initialize all motors
- Open file I/O
- Read Fits header files and initialize keyword tables
- Make connection to the DCS keyword library
- Open instrument engineering data log file
- Create a pipe for communications between parent and child
- Create a timeout pipe
- Fork to create a timeout clock
- Poll TSP read link
- Asynchronous I/O loop
    - Create a fd set having the RPC socket, the timer, and TSP link pipe fds
    - Block until an event arrives
    - Read TSP link pipe
    - Timeout to write engineering data log
    - Process RPC client request

Some explanation on the main function is given below:

1. The main function accepts two command line arguments. They are "**`-nodcs`**" and "**`-r`**".

2. Ctrl-C is disabled to prevent incidental killing of the program execution.

3. The message logging is performed by the Unix **`syslog()`** function.

4. The RPC server handle is created using the RPC function call **`svc_create()`**.

5. The transputer bootable file is defined by the environment variable **`NIRSPEC_BOOT_FILE`** and is loaded by the routine **`TSPCom_init()`** in **`tspcom.c`**.

6. If the option "**`-r`**" is selected, the server initializes the keyword buffer from the instrument configuration auto-backup file by calling **`config_backup()`**. Otherwise, the keyword buffer

is filled up with values defined in **KeywordTable[]** in the header file **nirspec.h** by calling the static function **config_default()**.

7. Some of the transputer cid's must be initialized when the server starts. That's done by the routine **cid_init()**.

8. All the step motors should be initialized when the server starts. The routine **STP_initAllMotors()** in **motor.c** performs this function.

9. The image frame buffers are opened by the function **FileIO_open()** in **fileio.c**.

10. The routine **Fits_init()** reads FITS header information files defined by the environment variables **NIRSPEC_HEADERINFO_SPEC** and **NIRSPEC_HEADERINFO_SCAM** and initializes FITS keyword tables.

11. The server logs instrument keywords in an engineering data log file defined by the environment variable **NIRSPEC_INSTR_LOG** every 5 minutes. This is done by the function **instrlog_open()**.

12

```
/*
 *  Create an unnamed pipe for uni-directional communications
 *  between parent and child
 */
pipe( pipe_fd );

/*
 *  Create a timeout pipe
 */
pipe( pipe_fd2 );

signal( SIGUSR1, ( void(*)() ) USR1Handler );

/*
 *  Fork
 */
if ( ( pid2 = fork() ) == 0 ) {
    /*
     *  Fork to create a timeout clock
     */
    if ( ( pid3 = fork() ) == 0 ) {
        while( 1 ) {
            sleep( timeout_val );
```

```
                /*
                 *  Write to timeout pipe
                 */
                timeout_id = 0;
                write( pipe_fd2[1], &timeout_id, sizeof( int ) );
            }
        }
        /*
         *  Poll TSP read link
         */
        else {
            while( 1 ) {
                /* data arrived */
                if ( TSPCom_getMessage( &cid, &param ) >= 0 ) {
                    /*
                     *  Write to pipe
                     */
                    write( pipe_fd[1], &cid, sizeof( int ) );
                    write( pipe_fd[1], &param, sizeof( long ) );

                    /*
                     *  Wait for signal from parent to resume polling
                    */
                    pause();
                }
                else
                    sleep( 1 );
            }
        }
    }
    /*
     *  Asynchronous I/O loop
     */
    else {
        if ( no_dcs ) {
            while ( 1 ) {
                FD_ZERO( &readfds );
                readfds = svc_fdset;
                FD_SET( pipe_fd[0], &readfds );
                FD_SET( pipe_fd2[0], &readfds );

                if ( ((i = select( maxfds, &readfds, NULL, NULL, NULL )) == -
1)
                     && ( errno != EINTR ) )
                    perror ( "select() failed" );
                else {
                    /*
                     *  Read tsp link pipe
                     */
                    if ( FD_ISSET( pipe_fd[0], &readfds ) ) {
                        read( pipe_fd[0], &cid, sizeof( int ) );
                        read( pipe_fd[0], &param, sizeof( long ) );
```

```
                    /*
                     *  Process message from tsp link
                     */
                    tsplink_msg_process( cid, param );

                    /*
                     *  Send signal to child to resume
                     */
                    kill( pid2, SIGUSR1 );
                }
                /*
                 *  Timeout to write engineering data log
                 */
                else if ( FD_ISSET( pipe_fd2[0], &readfds ) ) {
                    read( pipe_fd2[0], &timeout_id, sizeof( int ) );

                    if ( !Simulate )
                      instrlog_update();
                }
                /*
                 *  Process client request
                 */
                else
                    svc_getreqset( &readfds );
            }
        }
    }
    else {
        while ( 1 ) {
            FD_ZERO( &readfds );
            readfds = svc_fdset;
            FD_SET( pipe_fd[0], &readfds );
            FD_SET( pipe_fd2[0], &readfds );
            ktl_ioctl( dcs_khand, KTL_FDSET, &tempfds );
            j = 0;
            for ( i = 0; i < maxfds; i++ ) {
                if ( FD_ISSET( i, &tempfds ) ) {
                    dcs_fd[j++] = i;
                    FD_SET( i, &readfds );
                }
            }

            if ( ((i = select( maxfds, &readfds, NULL, NULL, NULL )) == -
1)
                 && ( errno != EINTR ) )
                perror ( "select() failed" );
            else {
                /*
                 *  Read tsp link pipe
                 */
                if ( FD_ISSET( pipe_fd[0], &readfds ) ) {
```

```
                    read( pipe_fd[0], &cid, sizeof( int ) );
                    read( pipe_fd[0], &param, sizeof( long ) );

                    /*
                     *  Process message from tsp link
                     */
                    tsplink_msg_process( cid, param );

                    /*
                     *  Send signal to child to resume
                     */
                    kill( pid2, SIGUSR1 );
                }
                /*
                 *  Timeout to write engineering data log
                 */
                else if ( FD_ISSET( pipe_fd2[0], &readfds ) ) {
                    read( pipe_fd2[0], &timeout_id, sizeof( int ) );

                    if ( !Simulate )
                  instrlog_update();
                }
                /*
                 *  Invoke KTL event handler
                 */
                else if ( FD_ISSET( dcs_fd[0], &readfds ) ||
                          FD_ISSET( dcs_fd[1], &readfds ) ) {
                    KTL_DISPATCH( dcs_khand );
                }
                /*
                 *  Process client request
                 */
                else
                    svc_getreqset( &readfds );
            }
        }
    }
}
```

`static void rpc_prog_1(rqstp, transp):`

This is the server dispatch routine.

`int config_default( void ):`

This routine initializes keyword values from the keyword table **`KeywordTable[]`** as defined in **`nirspec.h`**.

`int config_backup( void ):`

This routine initializes keyword values from the configuration auto-backup file defined by the environment variable **NIRSPEC_CONFIG_FILE**.

int config_update(int i):

This function routine updates the configuration auto-backup file every time a keyword is updated by the server.

int instrlog_open(void):

This routine opens the instrument engineering data log file defined by the environment variable **NIRSPEC_INSTR_LOG**.

int instrlog_update(void):

This function updates the instrument engineering data log file.

static void tsplink_msg_process(int cid, long param):

This routine processes messages from the TSP link.