# NIRSPEC

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

**George Brims**

**Revised January 4, 1999**

**NIRSPEC Software Programming Note 08.01**
**Host-Transputer Communication**

## 1 Introduction

In the NIRSPEC system, the control and acquisition functions are carried out by a network of transputers. The host computer is connected to this network through its external SCSI port, using the Transtech MatchBox to translate between the SCSI bus and the transputers' serial protocol. Our underlying server program interfaces to the MatchBox device via Transtech-supplied routines. In this programming note we describe the implementation of the various levels of communication routines for the host-transputer link.

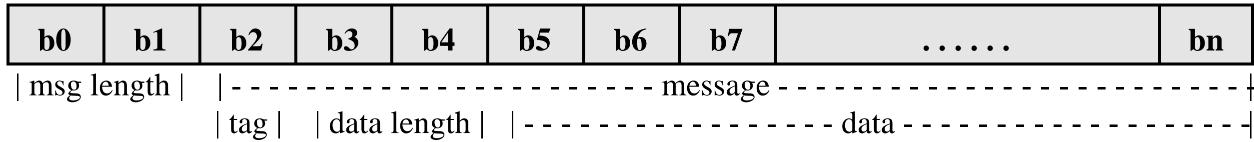
## 2 Message protocols

If the host and the transputers are to talk to each other they need to use a defined set of protocols, implemented in both the host and transputer software. On the host side, communication tasks are carried out by calling host-transputer interface routines. These routines handle message flow to and from the MatchBox and hide the details of the communication process from the high-level application code. Similar functions are provided by Occam processes on the transputer side. By working together, these communications routines provide an interface between the host control program and the underlying Occam processes.

Our starting point was a server program called `iserver`, supplied with the MatchBox, but it didn't meet all of our needs. We wanted to be able to layer a graphical user interface on top of our control code, and also communicate with IDL applications such as our Quicklook facility. Unfortunately `iserver` runs in the exact opposite manner. The occam program uses a library of host-interface routines to take charge of your host computer by sending requests to `iserver`. The host is used as an ANSI terminal and a filestore for the occam code. The screen I/O is extremely simplistic, so we can forget about a GUI. Because of this limitation we developed our own server program, based on the low-level host-transputer interface routines supplied by Transtech, and some of the higher-level examples they also provide. This server connects to the various graphical interface programs.

Although we abandoned `iserver`, we based our communications protocols on the style `iserver` uses. Each transaction across the link is a packet of bytes. The protocol must be flexible enough to deal with both short command messages and bulk data. An easy way to do this is to have the message start with a number indicating how long it's going to be. This is the `iserver` standard method, and meshes very easily with the transputer environment. (The occam language has a format for declaring a message protocol with a number at the beginning, followed by an array of size set by

that number.) For a discussion of occam message protocols, see NSPN22). We have one simple format, interpreted in two different ways for messages and data.




The above table illustrates the overall format. Since our maximum packet size is only 4096 bytes, the `iserver` protocol only needs two bytes to represent any possible message size. So **b0** and **b1** give the packet size, **n**, and bytes **b2** to **bn** comprise the message. Within the message, **b2** is a byte tag, **b3** and **b4** give us a 2-byte length, and **b5** to **bn** are a byte array of data of the given length.

For the message format, the tag byte **b2** is used as the command identifier, and the data is an ASCII string representation of a 32-bit integer[1]. (Making it a 32-bit integer is just a convention; since the number comes through as a string we could pass other data types, but this would complicate things further along). The root transputer (the one nearest the host) converts the string to an integer. The message is then passed on to the other transputers as the byte followed by the integer. Similarly, messages in byte-and-integer form are converted by the root transputer to byte plus 2-byte length plus string before transmission to the host. I am not sure why the 32-bit integer is sent as a string instead of as four bytes, but it is probably a historical artifact used to avoid big-endian/little-endian problems using transputers with a variety of host types. There is one limitation here, in that we can't send negative integers from the host to the transputers (but we can get them back to the host).

The data format is only used in one direction, transputer to host. The tag byte doesn't need to carry any information but is kept for consistency with the other format, so it's sent as a zero byte. The data length is the length in bytes of the data which makes up the rest of the packet. We have used 5 bytes in total for header information, leaving 4091 bytes for the data. We always send data as 32-bit integers, so we can only fit in a maximum of 1022 pixels in a data packet. This means each large packet (65536 pixels per acquisition transputer in the spectrometer channel for instance) is sent as a series of smaller packets of 1022 pixels, followed by a final shorter one.

Since we have this restriction of 1022 pixel values, the largest packet we can actually send is 4093 bytes, so the host program always sends each message in this size of block.

[1] The command identifier is referred to everywhere throughout the code and documentation as the “cid” (pronounced “see-eye-dee”). The integer is usually referred to as the parameter, and its variable name is usually “param”. Each cid corresponds to a KTL keyword; the effect of modifying a keyword that has a corresponding cid is that the cid is sent with the new keyword value as the parameter.

## 3 Transputer implementation

The root transputer program has two buffer process communicating with the host, one handling messages in each direction (see NSPN17 and NSPN21 for more details of the root program's operation and structure)

### 3.1 From-host buffer

The buffer reading messages from the host has the simpler task since it only receives messages in one form, and from one source, the link from the host. It extracts the byte command identifier from **b2**, gets the data length from **b3** and **b4**, then extracts the elements of the data string from the rest of the packet. It calls a conversion routine to turn that string into an integer. The occam protocol **trans2trans** is then used to send the byte cid and the integer parameter together around the transputer network.

### 3.2 Reply buffer

The process sending replies back from the transputers to the host is a little more complicated. For one thing it handles messages coming from several different parts of the transputer network. These are passed on to the host on a first come first served basis, except that once a message arrives from either camera section with a “frame ready” cid, the process enters a loop where it passes on all the sections of that data frame before dealing with any other messages.

The other complication is that messages could be of the simple form (byte plus integer) or the data packet form (size integer plus array of pixel values). There are two routines which package up the two types of message.

#### 3.2.1 msg.to.host routine

This routine converts the integer parameter to a string, with the string conversion routine returning a length for the byte string. It puts the byte cid in the first element of a buffer array, converts the string length to 2 bytes and puts them in next two elements, then follows that with the converted string. The write to the host link is of the form

```
to.host ! size::array
```

where `size` (a 16-bit integer) is the length of the message part of the packet, i.e. the string length plus 3. Writing in this occam form (called the counted array format) will send this length as the head of the packet, which will then be (string length plus 5) elements long.

### 3.2.2 data.to.host routine

This routine has to deal with more data but doesn't have to do any string conversion, since we leave the data in raw form (a necessary endian swap takes place in the host since it's the faster processor). The data packet which arrives from the camera sections is generally much larger than our 4096-byte packet, so it has to be split into multiple sub-packets for transfer to the host. In general this routine is called on to split a packet into the smallest possible number of sub-packets by making each carry the maximum complement of 1022 data values, with an odd-sized packet to finish.

The values for each sub-packet are copied into a buffer array, beginning with a zero byte (it's there to match the rest of the protocol but doesn't need to carry any information). The next two bytes are used to carry the number of bytes to follow, then the data values are copied into the buffer. Again, the write to the host link is of the form

```
to.host ! size::array
```

where `size` (a 16-bit integer) is the length of the message part of the packet, i.e. the string length plus 3. Writing in this occam form (called the counted array format) will send this length as the head of the packet, which will then be (string length plus 5) elements long.

## 4 Host side implementation

The host software uses low-level routines supplied by Transtech with the MatchBox, as a library called TSPLIB. The code module in the host is `tspcom.c`. The functions in `tspcom.c` can be divided into three levels in terms of how they interface to each other. Application programs call high-level routines which send data packets to, or receive data packets from, a host link via intermediate level routines. The details of establishing a host link, reading data from the link, and writing data to the link are handled by the low-level TSPLIB routines which are invoked by the intermediate level functions.

The host-transputer interface code is a part of the server software and the source code is therefore located in the server development directory `/kroot/kss/nirspec/keyword`. The C code comprises the header file `tspcom.h` and the source file `tspcom.c`. The vendor-provided low-level TSPLIB (Transtech SCSI Processor Library) C functions that handle the SCSI interface are contained in the object library file `/opt/transtech/lib/libtsp.a`.

All the high-level interface routines were developed in house. The intermediate level routines with the prefix “`OPS_`” were adapted from the vendor-supplied program `linkops.c` located in `/opt/transtech/source/linkops` with heavy modifications, while those with the prefix “`LOPS_`” are derived from `linkios.c` in the same directory, also with many changes. The rest of the intermediate-level routines are based on the source module `tsplink.c`, which is located in directory `/opt/transtech/source/linkios`, with minor modifications.

The three levels of function routines are listed below:

**High-level routines:**

| Routine | Description |
|---|---|
| `TSPCom_boot()` | - download a bootable occam file over the host link |
| `TSPCom_close()` | - close the host link |
| `TSPCom_getFrame()` | - get a data frame from the host link |
| `TSPCom_getMessage()` | - get a message from the host link |
| `TSPCom_getPacket()` | - get a data packet from the host link |
| `TSPCom_init()` | - initialize the host link and download the bootable file |
| `TSPCom_reset()` | - reset the host link |
| `TSPCom_sendCommand()` | - send a command id and parameter over the host link |
| `TSPCom_sendMessage()` | - send a message over the host link |
| `TSPCom_sendPacket()` | - send a packet over the host link |
| `TSPCom_trace()` | - set the host link trace flag |

**Intermediate level routines:**

| Routine | Description |
|---|---|
| `OPS_BootWrite()` | - write a bootable occam file to the link |
| `OPS_Close()` | - close the host link |
| `OPS_CommsAsynchronous()` | - set comm mode (asynchronous) |
| `OPS_CommsSynchronous()` | - set comm mode (synchronous) |
| `OPS_ErrorDetect()` | - check error detection |
| `OPS_GetRequest()` | - read from the host link |
| `OPS_Open()` | - open the host link |
| `OPS_Reset()` | - reset the host link |
| `OPS_SendReply()` | - write to the host link |
| `LOPS_BootWrite()` | - write a bootable occam file to the link |
| `LOPS_Close()` | - close the host link |
| `LOPS_CommsAsynchronous()` | - set comm mode (asynchronous) |
| `LOPS_CommsSynchronous()` | - set comm mode (synchronous) |
| `LOPS_ErrorDetect()` | - check error detection |
| `LOPS_GetRequest()` | - read from the host link |
| `LOPS_Open()` | - open the host link |
| `LOPS_Reset()` | - reset the host link |
| `LOPS_SendReply()` | - write to the host link |
| `AnalyseLink()` | - analyze the link connection |
| `CloseLink()` | - close the link connection |
| `OpenLink()` | - open the link connection |
| `ReadLink()` | - read from the link connection |
| `ResetLink()` | - reset the link connection |
| `SetProtocol()` | - set up communication protocol |
| `TestError()` | - test error status of the link |
| `WriteLink()` | - write to the link connection |

**Low-level routines:**

| | |
|---|---|
| `tsp_analyse()` | - reset a host link connection with analyze |
| `tsp_close()` | - close a host link connection |
| `tsp_error()` | - return host link connection error status |
| `tsp_open()` | - open a host link connection |
| `tsp_protocol()` | - set a host link read protocol |
| `tsp_read()` | - read data from a host link |
| `tsp_reset()` | - reset a host link connection |
| `tsp_write()` | - write data to a host link |

In the following sections we describe the high-level host-transputer communication routines only, since they are customer-built. The programming details of the intermediate level functions can be found from their original source programs. For the low-level TSPLIB routines, the reader should consult Chapter 8 in the "MatchBox User Manual" which discusses the TSP library.

As discussed in section 2 above, a message packet size of 4093 bytes is used by the link interface. Both short command messages and long pixel data are transferred with the same packet size. The 4088 bytes of message/data body in the packet can hold 1022 pixels of data (4 bytes per pixel) and therefore `tspcom.c` assigns an initial value of 1022 to the data packet size variable `PacketSize`. Although `PacketSize` can be changed in the program, it actually remains fixed in `tspcom.c` because this value allows a maximum data transfer throughput. The program uses `*Tbuf`, a pointer to the message packet, to hold a packet of message (both header and body). `*Tbuf` is dynamically allocated with different packet sizes.

### 4.1 TSPCom_boot

**`int TSPCom_boot( char *boot_file )`**

`TSPCom_boot()` first opens the boot file `boot_file`, fills the boot data buffer, of size `BOOT_BUFFER_LENGTH` (2048 bytes) with the boot data, and then calls `OPS_BootWrite()` to write the boot data to the host link. The routine then calls `OPS_CommsAsynchronous()` to set an asynchronous communication mode for the link. This routine is called only by `TSPCom_init()` in `tspcom.c`.

When the simulation flag `Simulate` is `TRUE`, this routine returns immediately without doing anything (this is true for most of the other high-level interface routines as well).

### 4.2 TSPCom_close

**`int TSPCom_close( void )`**

This routine frees the allocated memory space for the packet transmission buffer `*Tbuf` and calls `PS_Close()` to close the link.

### 4.3 TSPCom_getFrame

**int TSPCom_getFrame( int chan, int frame_num )**

This function receives a pixel frame from the link. The argument chan indicates the origin of the incoming data: 0 is for the spectrometer and 1 for the slit-view camera. The other function argument frame_num designates the sub-frame number for a $1024^2$ spectrometer image. Because of the throughput limitation of the MatchBox, it takes over 4 seconds for the transputer to transfer a spectrometer frame to the host. During the data transfer period, the host-transputer link is solely dedicated to the data reading process and no other communication task can be performed. This causes a problem for the DCS to send time-critical correction parameters to the **NIRSPEC** image rotator continuously (at 1Hz) without an interruption. To get around this problem, a $1024^2$ spectrometer frame is divided into 16 sub-frames for transfer and thus the link can be taken over by other communication tasks between the subframe transmissions. Note that a $256^2$ slit-view camera frame is transferred as a single frame. When the simulation flag Simulate is TRUE, the routine generates a simulated frame.

The program first identifies whether the incoming subframe is the first_frame or the last_frame as shown below:

```
if ( chan == 0 ) {
   if ( frame_num == 1  || Simulate ) first_frame = TRUE;
   if ( frame_num == 16 || Simulate ) last_frame = TRUE;
}
else {
   first_frame = TRUE;
   last_frame = TRUE;
}
```

For the first subframe, the program allocates a memory space for frame to hold one row of pixel data, removes the old frame buffer Frame in /tmp and opens a new one:

```
if ( first_frame ) {
    frame = (long *)malloc(PacketSize*sizeof(int));
    unlink( Frame[chan] );
    fp_frame = fopen(Frame[chan], "wb");
}
```

Since the data packet size PacketSize is 1022 pixels (long integers) or 4088 bytes, a single subframe ($256^2$ ) needs to be transferred in 65 data packets (NUM_PACKETS) . The following code reads each packet of data into Tbuf, packs every four pixel bytes (inbuf) into the long integer data, and then puts it into the single row data buffer frame which is subsequently written into the image frame buffer Frame:

```
for ( i = 0; i < NUM_PACKETS; ++i ) {
    status = TSPCom_getPacket();
    if ( status < 0 )
        return status;

    inbuf = &Tbuf[5];
    for ( j = 0; j < PacketSize; ++j ) {
        data  =              *inbuf++;
        data +=      256 * (*inbuf++);
        data +=    65536 * (*inbuf++);
        data += 16777216 * (*inbuf++);
        *(frame+j) = data;
    }

    index = i * PacketSize * frame_num;
    fseek( fp_frame, 4*index, SEEK_SET );
    if ( i < (NUM_PACKETS-1) )
        fwrite( frame, 4, (int)PacketSize, fp_frame );
    else {    /* only need to write 4 pixels from last packet */
        fwrite( frame, 4, 128, fp_frame );
    }
}
```

Note that the last packet in a subframe contains 128 pixels only (65536 = 64*1022 + 128).

In the simulation mode, the routine will use the random generator `rand()` to produce a noise frame, instead of reading the host link.

If the subframe is the last one in the series, the routine closes the image frame buffer `Frame` and releases allocated memory space for `frame`:

```
if ( last_frame ) {
    fclose( fp_frame );
    free( (void *)frame );
}
```

### 4.4 TSPCom_getMessage

**`int TSPCom_getMessage( int *cid, long *param )`**

This function calls `TSPCom_getPacket()` to receive a data packet from the link. If `TSPCom_getPacket()` returns a value less than 0, which indicates no data available in the link, `TSPCom_getMessage()` simply returns. Otherwise, it unpacks the message from the transmission buffer `Tbuf` and extracts the command id `cid` and the parameter value `param`.

### 4.5 TSPCom_getPacket

**int TSPCom_getPacket( void )**

This routine receives a data packet from the link by calling the intermediate-level function OPS_GetRequest(). The data packet is stored in the buffer Tbuf. If no data is available in the link, OPS_GetRequest() returns STATUS_NODATA (declared as -99 in tspcom.h). Otherwise, it returns 0.

### TSPCom_init

**int TSPCom_init( char *boot_file )**

For speed, an incoming pixel frame from the transputer is stored temporarily in a frame buffer in the memory-based file system /tmp where it is converted into a FITS file and written to the hard disk. Two frame buffers, /tmp/FRAME for the spectrometer image and /tmp/FRAME2 for the slit-view camera image, are defined. This routine then calls OPS_Open(), OPS_CommsSynchronous() and OPS_Reset() to open a host link, set a synchronous communication mode for the link and reset the link, respectively. Note that a call to OPS_Open() will return the link id ConnId which identifies a specific link throughout a connection session. If these operations are successful, the program will then invoke TSPCom_boot() to down-load the boot file onto the link.

When a message is sent to the link, it's first packed up in the message transmission buffer Tbuf per communication protocol. The host-transputer protocol allows a variable packet size, although a fixed size (4088 bytes of data plus 5 bytes of header) is actually used in the NIRSPEC software. For efficiency, the size of the transmission buffer should be able to change "on the fly" as the data packet size changes. Therefore, the transmission buffer Tbuf is allocated dynamically as shown below:

```
Tbuf = (unsigned char *)malloc(PacketSize*sizeof(long)+5);
```

Note that the data packet size PacketSize is 1022 pixels and each pixel is encoded by a long integer (4 bytes). In addition, Tbuf allocates 5 bytes for header information.

If there's no error, TSPCom_init() returns a positive link id ConnId.which is passed by a function argument in OPS_Open(). Otherwise, it will return -1.

### 4.6 TSPCom_reset

**int TSPCom_reset( char *boot_file )**

This routine resets a link by first closing the link and then calling TSPCom_init() to re-initialize the link. An application program can use this routine to reset a host-transputer link which hangs up during a session, without killing the whole program.

### 4.7 TSPCom_sendCommand

**`int TSPCom_sendCommand( int cid, long param )`**

The host software communicates with the transputer system via keyword-style commands. Each command consists of an integer command id called `cid` and a long integer parameter `param`. This routine calls `TSPCom_sendMessage()` to perform the data sending task.

### 4.8 TSPCom_sendMessage

**`int TSPCom_sendMessage( int *cid, long *param )`**

This routine packs up a command id and its parameter into the transmission buffer `Tbuf` per the communications protocol and calls `TSPCom_sendPacket()` to send the message packet over the link.

### 4.9 TSPCom_sendPacket

**`int TSPCom_sendPacket( void )`**

This routine calls `OPS_SendReply()` to send the message packet `Tbuf` to the link. The buffer `Tbuf` can be filled up by `TSPCom_sendMessage()`.

### 4.10 TSPCom_trace

**`void TSPCom_trace( int mode )`**

This routine sets the link trace flag `TSPTrace` for debugging purpose. When `TSPTrace` (set by `mode`) is `TRUE`, the link debug trace is turned on and all the message flow over the link will be displayed.

## 5 Program building

The source module `tspcom.c` is compiled by `makefile` in the server directory and built into the server executable `nirspec_server*` as shown below:

```
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 RPC server program
nirspec_server:$(APP)_server.o $(APP)_svc_proc.o $(APP)_xdr.o fileio.o tspcom.o
      $(CC) -o nirspec_server $(APP)_server.o $(APP)_svc_proc.o $(APP)_xdr.o \
      fileio.o tspcom.o utils.o $(LIBS) /opt/transtech/lib/libtsp.a
```

Note that `libtsp.a` contains the low-level TSP library routines.