NIRSPEC

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

Tim Liu **January 30, 1996**

NIRSPEC Software Design Note 3.01
Host-Transputer Communications Protocol

1 Introduction

NIRSPEC uses transputers to perform functions of array clocking, data acquisition, and stepper motor control. In the system configuration, the host Sparcstation is connected to the transputer network via a Transtech product called Matchbox on the external SCSI port of the workstation. Because the user-interface program will run from the Sparcstation, the transputer system has to be slaved to the host computer, i.e. all I/O service requests are initiated from the host computer. Unfortunately, the INMOS host file server program (iserver) provided in the Occam toolset that we use for our transputer software development does just the opposite. Therefore, we need develop our own host-transputer communications protocol and interface routines. This document will address the issue of communications between the host and the transputer in the Matchbox, while the communications between the Matchbox and transputer subsystems will be discussed in a separate design note.

2 Protocol Design Considerations

Several issues are considered when designing the communications protocol. The starting place should be the iserver protocol implemented in the Occam toolset. By studying and modifying the iserver protocol we will come up with a protocol suitable to our applications.

First, communication between the host and the transputer takes place by passing a message that consists of a data stream of bytes. So, a byte is the smallest cell in a message. Second, communications in Occam processes are carried out by means of channels and a conventional channel protocol only permits data arrays of fixed sizes, i.e., messages must have the same length. This is not the case in such applications as transferring data from a sub-array readout. In order to allow data arrays of arbitrary size to be sent over an Occam channel, a variable length array protocol has to be declared, in which the channel carries a size value of the array (called counted array) first, then followed by the actual array components (Occam finds the array size and places it in the beginning of the data stream automatically). This means that in our protocol definition a data packet must begin first with the length of message to be sent out. Third, to simplify implementation in software we should use a single protocol for passing both short command messages and long data packets. This requires that we carefully choose the size of the message packet which can range from a few bytes, when passing a command, up to 4 Mbytes, when transferring a 32-bit 1024x1024 frame.

A small packet size will increase the overhead when transmitting frame data since it needs to be sent out in many packets. On the other hand, a very large packet size will also make the flow of short command messages inefficient. Therefore, an optimum packet size should be a compromise between message flow and data flow. Because most of the communications between the host and the transputer are to pass short commands, we will use a short packet size rather than a very long one in the protocol.

We now consider how to construct a message that carries all the information it needs to pass. In general, an I/O service request message initiated from the host computer has two components. The first component asks what kind of service request is and the second one carries a parameter value to be passed when the request is served by the transputer. For example, if we want the transputer system to start a 60 seconds integration, the command will be "integrate" and the parameter value is "60". When constructing this message we can use one byte to encode "integrate" and two bytes to represent "60". The command or tag byte is placed in the beginning of the message. When the transputer receives the message packet it decodes the tag byte and extracts the parameter value from the message, and then takes appropriate actions based on the command and parameter it has received. In this way any I/O service calls generated by the control program can be easily encoded in a simple message. Similarly, when the transputer transmits a data frame back to the host, a tag byte can be placed in the message packet that indicates the arrival of frame data.

3 Host-Transputer Communications Protocol

Based on the above considerations a simple protocol is adopted. The structure of the message packet is shown in Figure 1.

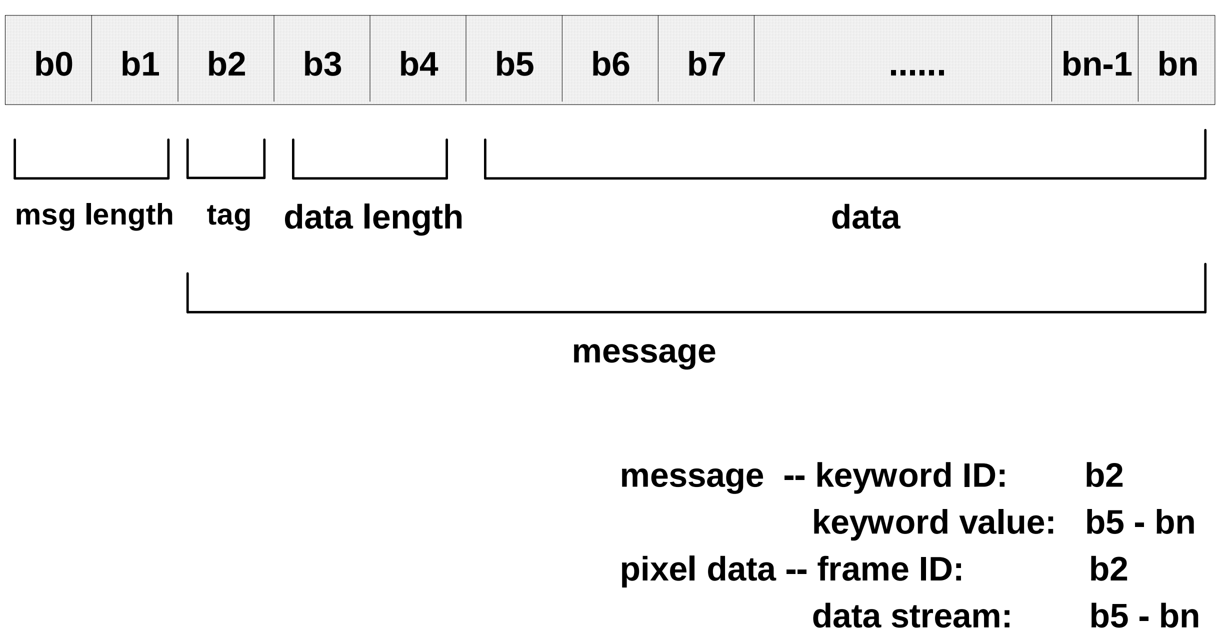

Figure 1 Message packet structure

The first two bytes b0 and b1 gives the length of the message which equals (b0 + 256*b1). Note that the message starts from b2 and ends at bn. b0 and b1 forms a signed 16-bit integer because integer types declared in Occam are all signed. So the maximum length of a message we can possibly have is 32767 bytes. The current implementation of the protocol sets a default message size of 4111 bytes (4 Kbytes of data + 5 bytes of header, see below). The packet size will be able to be changed from the user-interface program because of the requirements of some engineering functions. Therefore, one 32-bit image frame needs to be sent out in 1024 separate but continuous message packets. Unlike the iserver protocol, the message packet size here can have either even or odd number of bytes.

The third byte b2 is the tag byte which carries the command information. It is unsigned and can therefore take a value from 0 to 255. 256 different command tags should be enough for our needs.

The bytes b3 and b4 gives the length of data packet immediately followed them: b5, b6, b7, ..., bn. And this equals (b3 + 256*b4). As determined by the maximum message length, the maximum data packet length is 4 Kbytes. The minimum data packet length is zero. Although the data packet size seems redundant since it can be deduced from the given message length, we still keep it in our new protocol because it comes from the iserver protocol and it will make software implementation of communication routines convenient in some cases.

To summarize, a message consists of two parts according to our protocol: the message header which contains the message size, tag value, and data size and takes a total of 5 bytes, and the message body containing the actual message which can be a parameter value or a data stream.

4 Host-Transputer Communication Interface Routines

The defined protocol has been implemented in host and transputer software. On the host side, communication tasks are carried out by function calls from a host-transputer library. The function routines handle message/data flow to and from the transputer in the Matchbox, and hide the details of communication process from high-level application code. Similar functions are provided by Occam code on the transputer side. Thus these two sets of communications routines provide an interface between the host control program and the Occam processes.

4.1 Host communications routines

Communications functions on the host computer are implemented in C. These routines can be classified as two levels. Application programs call high level routines which send and receive messages/data to/from a message buffer via low level routines. The details of establishing a link, reading/writing data from/to the link are handled by the low level routines. We list these routines below, along with their functions:

High level routines:

TSPCom_init	- initialize host link and download bootable
TSPCom_sendCommand	- send a command to link
TSPCom_getMessage	- get a message from link
TSPCom_sendMessage	- send a message to link
TSPCom_getFrame	- get a frame from link
TLink_getPacket	- get a message packet from link
TLink_sendPacket	- send a message packet to link
TLink_boot	- download a bootable file to link
TLink_trace	- trace link message flow

Low level routines:

OPS_Open	- open host link
OPS_Close	- close host link
OPS_Reset	- reset host link
OPS_BootWrite	- write bootable to link
OPS_ErrorDetect	- check error detection
OPS_CommsAsynchronous	- set comm mode (asynchronous)
OPS_CommsSynchronous	- set comm mode (synchronous)
OPS_GetRequest	- read from host link
OPS_SendReply	- write to host link
OpenLink	- open link connection
CloseLink	- close link connection
ResetLink	- reset link connection
AnalyseLink	- analyze link connection
ReadLink	- read from link connection
WriteLink	- write to link connection
TestError	- test error status of link
SetProtocol	- set up protocol

4.2 Transputer communications routines

The host-transputer communication protocol is called TSP in Occam. It is declared as follows:

```
PROTOCOL TSP IS INT16::[]BYTE
```

INT16::[]BYTE is a counted array in Occam language that has a variable length. The value INT16 gives the size of the array. This is necessary for our application because the protocol should be flexible enough to be capable of carrying frame data of any size as discussed before. Since the transputer is slaved to the host computer, it must run a process all the time that polls the host link to check the arrival of a message. In Occam this can be implemented as follows:

```
fh ? in.buf.size.INT16 :: in.buf          -- poll the link from PC
SEQ
   cid := in.buf[0]                       -- read tag byte cid
   param := ...                  -- read paramter value
   CASE cid                               -- start a selection process
      cid.abort
         ... process
      cid.go
         ... process
      ... more process
```

The transputer program has two Occam procedures that send messages and frame data to the host computer:

```
msg.to.host()   - send a message back to host link
data.to.host()  - send frame data back to host link
```