NIRSPEC

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

George Brims

Revised November 30, 1998

NIRSPEC Software Programming Note 21.00

Transputer-transputer communications

1 Introduction

This note describes how the transputers in the NIRSPEC system communicate with each other, including details of the message passing protocols and the structure of each program. For an overall introduction to transputers and occam, see NSPN22. A separate note (NSPN08) has more detail on the host-transputer interface, although we will discuss some aspects of it here.

2 Hardware details

The transputers talk to each other over standard transputer serial links, each link being bi-directional and running at 20 Mbit/s. They are connected in a network to the host Sparcstation, through a device called a Matchbox, which translates between the SCSI protocol on the host side and the transputer serial protocol on the other. The Matchbox connects to the workstation's SCSI port through a pair of fiber-optic extenders, which bridge the 300 feet or so of cable run between the computer room and the Nasmyth platform.

The network layout is shown in Figure 1. The network has three branches connected to the "root" transputer. This transputer is housed in its own enclosure, along with two serial transputers which

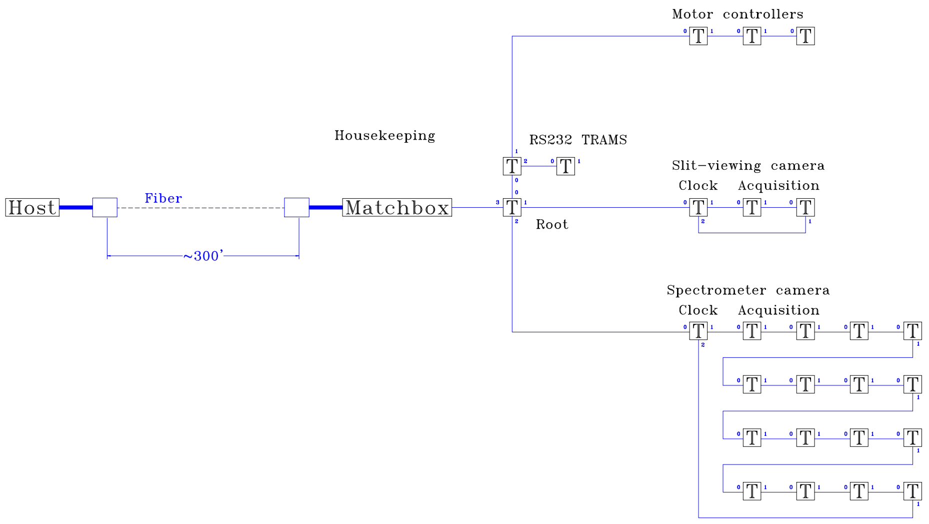

Figure 1: Transputer network layout

add RS232 capability to the system. The rest of the transputers are on VME format cards in a 6U high card cage.

The two camera sections each have a clock generator transputer, followed by 2 (slit-viewing camera) or 16 (spectrometer camera) data acquisition transputers. The last acquisition transputer in each chain is looped back to the clock generator, so each camera section is a message loop. The third branch has the two RS232 TRAM[1] modules (one on a side branch) and three motor control transputers.

3 Process types

All our occam programs have overall structures which combine a few basic types of process as outlined below.

3.1 One-to-many buffer

This process takes a message from a single input (usually coming from the direction of the host), and routes it onward to one of a number of other processes. To decide where to send the message it looks at the command id (cid) contained in the message. This process is pretty simple; it's just an endless WHILE loop with a message input statement, reading any message that arrives from one other processor. Once a message is received it is processed through a CASE statement on the value of the cid.

3.2 Many-to-one buffer

This process is the reverse of the one-to-many process, and a little more complex. Again it is an endless loop, but this time it has to handle replies heading back toward the host from several possible sources, and multiplex them onto a single output. At the top of the loop is an ALT construct. This is a facility of the occam language that allows a process to sit idle waiting for input from multiple sources, yet respond very quickly once an input is received. Once a message comes in, it is sent on in the direction of the host no matter what it is. In the case of a cid denoting that data is ready, the process will pass all blocks belonging to that individual frame before going back to the top of the loop to process another input.

3.3 One-to-one buffer

This is the simplest type of process, simply taking in messages on one channel and passing them straight on through another without even examining the contents. In some processors, especially the root processor, we use chains of these simple processes to form a First-In-First-Out (FIFO) buffer for messages. This makes the system less prone to problems when one process becomes busy, perhaps because of delays in talking to external hardware, and so can’t receive further instructions. If more command messages are sent to that process, then undelivered messages back up through our

[1] TRansputer Application Module - a transputer daughtercard standard format.

network until the host process can't send any more. A chain of these simple processes then effectively lengthens all the branches of the network so it can hold more pending messages.

3.4 "Do work" process

Obviously the transputers are doing more than just passing messages to and fro. Some of the processes are actually doing work, controlling the instrument or monitoring it. A "do work" process will take messages from the host or another transputer telling it what to do, and may or may not send back one or more replies as a result. Some "do work" processes are alone on their own transputer, while others are one of many on a particular processor. Structurally, a "do work" process is usually an endless loop headed by a message input. Once a host message is received, it is passed to a CASE statement, which looks at the command identifier (cid) contained in the message, and steers operation to the appropriate section of code. In the case of the acquisition transputers, each command message received is immediately passed on unaltered to the next transputer in line before being obeyed.

The simplest and most common operation in any process is simply to change the value of some internal variable to match the value supplied in the parameter field of the message. Typically, a KTL keyword belonging to the nirspec server is modified, and if it has a corresponding cid (as defined in **nirspec.h**), that cid is sent to the transputers, with the modified value as the parameter. Thus the keyword modify command in the host software results in a keyword value change in the occam software. Since the parameter value sent to the transputers is a 32-bit integer, floating-point keyword values must be scaled for transmission. For example integration times are converted to milliseconds.

4 Message passing philosophy

The philosophy of the NIRSPEC transputer message system is pretty simple. The driver software on the host sends and receives variable length packets up 4096 bytes in length. The first transputer in the chain, known as the root transputer, translates to and from the other formats used in the rest of the transputer network.

The primary type of message passing between transputers consists of a byte command id (the "cid"), followed by a 32-bit integer value (the "parameter"). These cids mostly correspond to KTL keywords in the host server software (although some are only passed between transputers). Most communication, such as all commands from the host, and the majority of replies, are in this format. The major exception is the format of larger messages containing image data going back to the host. The only other exception is communication between the programs serial1 and serial2, and their two copies of the driver supplied with the RS232 hardware. The driver was written to send and receive in the ANY protocol, explained in section 4.4.

The occam language requires that messages be strictly formatted, and that the formats be declared in a protocol definition. Each link (physical or virtual) is then declared as using one of the protocols, and every message sent is a list of items that corresponds to the protocol declaration. This is not as complicated as it sounds. The only exception to this formalism is the ANY protocol - a raw protocol that relies on complete agreement on message content between the sending and receiving programs at all times.

We have two different link protocols defined for communicating between the transputers, and another for communications to and from the host. The protocols are defined in the file **nirspec.inc**, as follows:

```
PROTOCOL TRANS2HOST IS INT16::[]BYTE :
PROTOCOL TRANSINT32 IS BYTE; INT32 :
PROTOCOL TRANS2TRANS
  CASE
    msg; BYTE; INT
    data; INT; INT::[]INT
:
```

4.1 TRANS2HOST protocol

The host-transputer protocol (TRANS2HOST) uses the counted array format of occam. A sixteen bit integer is followed by a byte array, the size of which is carried by the integer. Every message from the host, or reply or data packet going to the host, is converted between the other protocols and this one by the root transputer program. The host software sets a maximum block size of 4096 bytes for a transfer, so returning data packets (which can be up to 256 kbytes in size) have to be split up by the root code and sent to the host as a series of shorter pieces. There are more details of this arrangement in the document describing the root transputer code (NSPN17), and host-transputer communications (NSPN08).

4.2 TRANS2TRANS protocol

The TRANS2TRANS protocol used between the majority of the transputers is a dual protocol, since we have a couple of different requirements for message passing. We need a format for simple messages passing commands from the host and returning status or error messages on completion, and we need a format which will allow us to pass back large blocks of data from the camera sections. When we have a requirement to send different message formats over the same channel, occam allows us to list them with the CASE keyword as above. In order to distinguish the formats, they must each start with a name, in our case “msg” for messages and “data” for data packets. This name must then appear as the first item in the list of data elements when we do a read or write. We must also use the CASE construct when we read in a message over such a channel, so that we handle both eventualities

For the “msg” format we use the same protocol as the **iserver** program supplied as part of the occam toolset. [This program is meant to run on the host computer and provide a front end — terminal and filestore — to the occam program. This would limit us to a crude text-based user interface, so instead we use our own server so we can have our assortment of GUI clients.] The protocol is one byte followed by one (32-bit) integer. The byte is the command identifier, referred to everywhere as the “cid”. In most programs “cid” is usually the name of the variable, which is compared against a list of constants defined in the primary include file **nirspec.inc**. These all have names of the form **cid.xxxx**. The integer is the parameter of the command (variable name is usually “param”). This mirrors the keyword concept used in the host code. Not all server keywords have corresponding cid values, but for those that do it's a one-to-one mapping.

The “data” format uses a counted array of 32-bit integers to handle the longer data packets. First is an integer, usually identifying which transputer was the source of the data packet, then an integer holding a counter, followed by an integer array of that size.

4.3 TRANSINT32 protocol

The other protocol used between transputers is TRANSINT32. It is necessary because we have to pass some messages to or from or through the two RS232 transputers. These are 16-bit T225 processors. We can't declare the same “msg” protocol because it uses INT and the default size of an INT on those processors is 16 bits instead of 32. Instead we duplicate the “msg” style of the TRANS2TRANS protocol, but with the integers *declared* as 32 bits. The RS232 and motor control transputers don't produce blocks of data, so there was no need to duplicate the “data” format for them.

4.4 ANY protocol

The ANY protocol is the crudest protocol available, that is it isn’t really a protocol at all, more a declaration we’re planning to send all sorts of stuff over this link. The problem with this is it’s possible to write programs for two processes that will talk to each other, have a mismatch between what one sends and the other expects, and get no help from the development system in catching the error. If you’re using a defined protocol like the others above, the send list and receive list must both agree with the protocol for that link or the code won’t even *compile*, let alone run. With the any protocol the programs will compile and link, but one or both will stall when they try to make the mismatched transfer. The RS232 driver was written to use this protocol so that it could be sent commands, and send and receive single characters or strings. And yes, we did have this sort of problem!

5 Program structures

All the different programs in the NIRSPEC system have one thing in common; once they have initialized values of assorted variables and sent out some default values to hardware if necessary, they all enter endless loops where they sit and wait for messages from the host computer, or replies from the other transputers. When appropriate they either act on the host messages or pass them on to other transputers (and sometimes both). The programs for each transputer are described elsewhere, so here we will just outline the message handling part of each.

5.1 Root transputer

The root transputer has two main buffer processes, each connected to the host computer on one side and the rest of the transputer network on the other. There is a one-to-many process receiving messages from the host and routing them to the other transputers, and a many-to-one process receiving messages or data blocks from the other transputers and passing them back to the host. The receive-from-host buffer continually monitors the host link and takes in each message, unpacks it from the counted byte array format and re-packages it into the "msg" format of byte cid and integer parameter. It then examines the cid of the message and sends it on down the appropriate link. In a very few cases it does something with the message itself. In order for it to reply, it needs to send a message to the other buffer for transfer to the host. It's not allowed to have a sending channel to the host since the other parallel buffer already has one, so it passes the message to the other buffer over a virtual channel called "buffer2buffer".

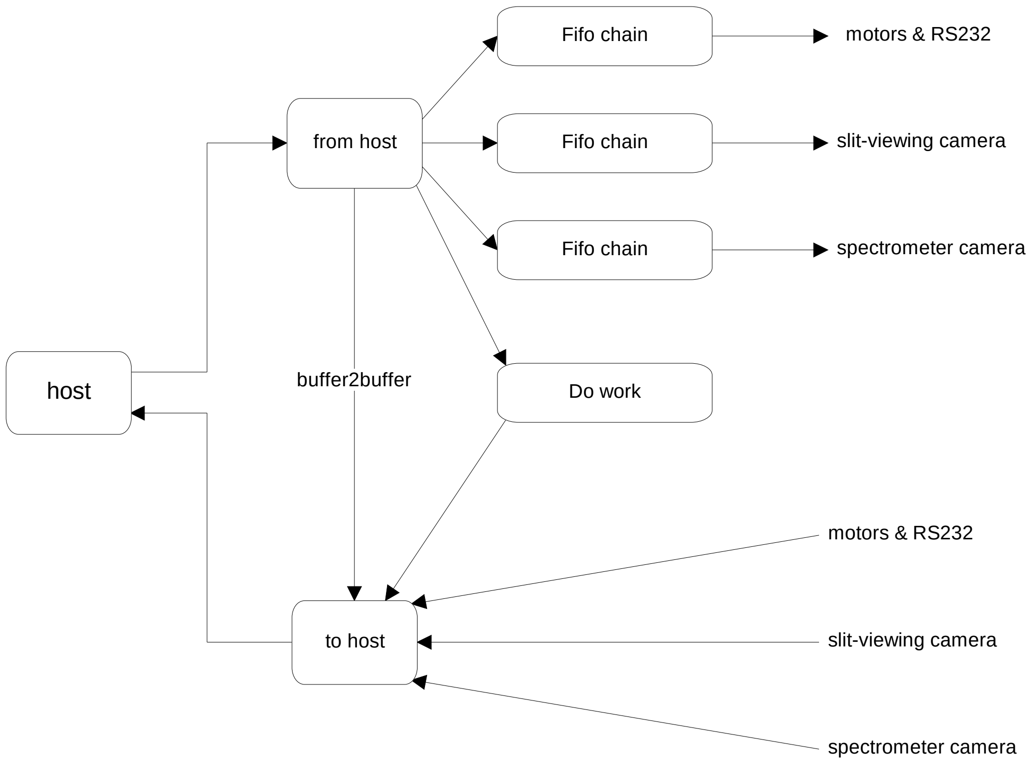

Figure 2: Root program structure

The other buffer process is a many-to-one process monitoring the three channels from the three physical branches of the network, and the virtual link from the other process. Whenever a message or data packet comes in, it re-packages it for transmission to the host. If the message contains a "data ready" cid from either camera section, it enters a loop where it takes in and passes on the appropriate number of data packets for that image, before going back to the top of the main loop and checking for the next message. This ensures that we don't accidentally interleave data packets from the two camera sections if they should both happen to finish at the same time.

In addition to its message routing functions, the root transputer also does a little work of its own, talking to some temperature sensors (Dallas Semiconductor DS1820) distributed around the electronics cabinets, and controlling and monitoring the various calibration unit lamps. This work is performed by a separate “do-work” process. Messages from the “from host” buffer go to this process over virtual link “2worker”, and replies go to the “to host” buffer of virtual link “fromworker”

There are also a whole set of one-to-one buffer processes forming a FIFO buffer for messages sent by the “from host” buffer to the motors/RS232, spectrometer and slit-viewing sections. These use a set of virtual links (actually declared as three 8-element *arrays* of links) called to.motor.fifos, to.spec.fifos, and to.scam.fifos. The last process in each chain then talks over a physical link to the other processor.

5.2 (RS232) transputers

The first RS232 transputer (known as serial1) has to pass messages both ways between the root and the motor transputers, as well as perform its own functions when the cid dictates. The overall structure is similar to the root transputer.

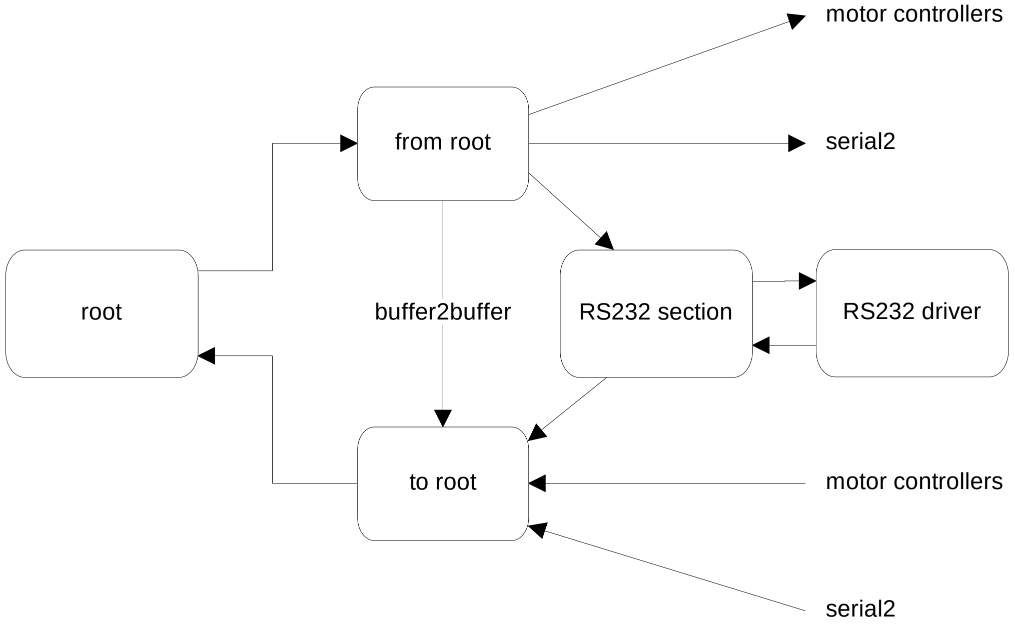

Figure 3: Serial1 structure

There is a one-to-many buffer (labeled here “from root”) handling messages from the root transputer and passing them on to the other serial transputer (serial2) or the motor controllers. The other buffer (labeled “to root”) is a many-to-one type, taking in replies from serial2 or the motor controllers, and passing them all back to the root. As in the root program there's a “buffer2buffer” virtual link so the “from root” process can reply to the root. To perform its own functions the program has a “do work” section, the RS232 section, also looping endlessly. This piece of code acts on any cid messages passed to it by the “from root” buffer, and sends its replies back via the “to root” buffer. In order to interface with the RS232 hardware, it communicates with a driver process (a piece of code supplied with the hardware). All four channels in and out of the RS232 section of the code are virtual links, since they run between channels on the same processor. The links to the driver are of type ANY, meaning there's no set protocol to the messages. In fact what is sent is always a stream of bytes, with the RS232 section and the RS232 driver being carefully written so each knows what to send or expect in reply.

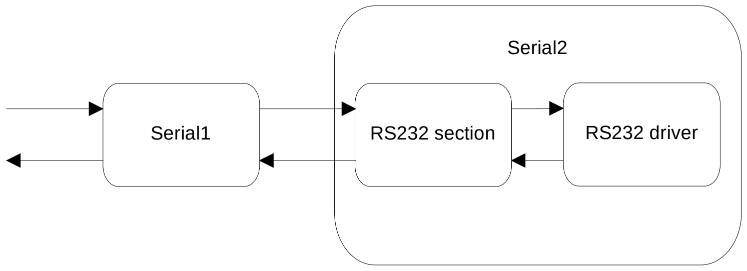

Figure 4: Serial2 structure

The second serial processor is simpler since it's on the end of a branch of the network, and is not expected to receive any messages other than commands it’s supposed to obey (if it gets any others it will just ignore them). There are just two parallel processes, the RS232 section which takes in messages passed on by processor serial1, and the RS232 driver section. The RS232 section loops indefinitely waiting for commands from the host, and carries out any requested function, using the same RS232 driver as in the other serial processor. The links to the driver are again virtual links of protocol type ANY.

5.3 Motor control transputers

There are three motor control transputers, each controlling a number of mechanisms via their digital I/O ports. They are connected on one branch of the network following the serial1 transputer. Their programs are all similar in structure and message handling philosophy, with each having a one-to-many buffer process receiving and distributing commands and a many-to-one process collecting and forwarding replies. The motor code is more fully described in NSPN20.

Each board has four I/O ports, with a parallel process assigned to each so that we can drive up to four mechanisms from each board. In addition each board has a process handling a watchdog timer built into the hardware. This guards against the situation where a motor program hangs up and leaves the “motor power on” signal on an I/O port high. This would leave a holding current on the stepper motor, resulting in overheating. If the watchdog process doesn't reset the hardware timer every now and then the output is automatically shut down to protect the motor.

5.3.1 Motor 1

Motor 1, the first motor transputer, controls 4 mechanisms: both filter wheels, the slit wheel, and the dust cover of the calibration unit. The command buffer passes commands to these processes, as well as to Motor 2 and the watchdog process. Replies are processed by the reply buffer, which sends them on to the command buffer and so back to the host. This is different from other processors like the root and serial1, where the reply process talks directly to the next processor in the host direction. The reason for this has to do with the operation of the motor quit command (see NSPN20), and we won't go into it here.

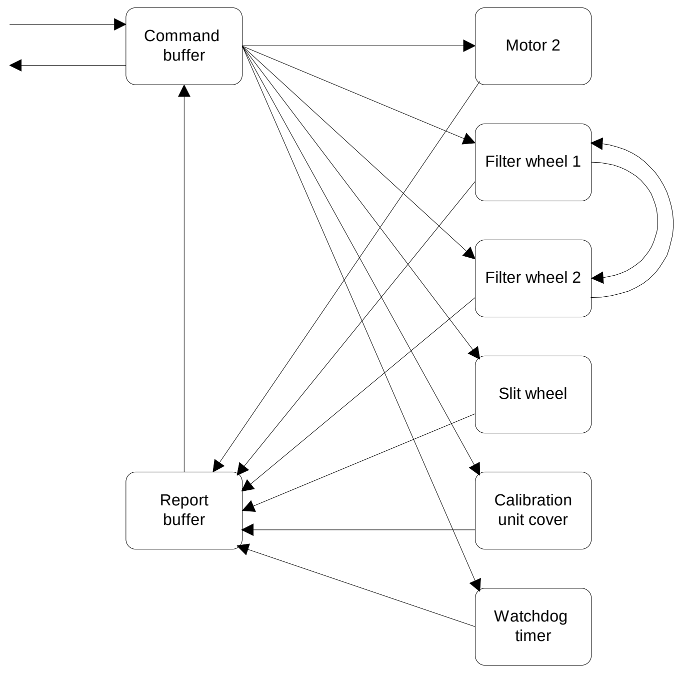

Figure 5: Motor 1 structure

The other unusual thing about this program is a pair of extra bidirectional links between the two filter wheel processes. These are used to coordinate operation of the two wheels, to ensure that they don't both go to open positions and flood the detectors with light.

5.3.2 Motor 2

Motor 2 has essentially the same structure, except that the four working processes this time drive the echelle mechanism, the cross-disperser, the calibration unit flip mirror and the calibration unit pinhole mechanism. The incoming command messages come from Motor 1, and some are passed to Motor 3. Again there is a watchdog timer process in addition to the working processes, but no extra links between the mechanism driving processes.

5.3.3 Motor 3

Motor 3 is similar to the other two motor processes, except there are fewer processes because it only drives one mechanism, the image rotator. Also, because it's at the end of the chain, it doesn't have to pass on any commands. There is still a watchdog process as in the other two motor transputers.

5.4 Clock generator transputers

The message passing structure of both clock generator transputers is simple, since the clock generator transputers perform only one function at a time. Each program is one single process which loops endlessly, monitoring the three possible message inputs through an ALT structure. Each clock generator is connected to the root processor, the first acquisition transputer, and the last acquisition processor. Any message or data packet from the acquisition processors is simply passed on to the root transputer, and the program returns to the ALT loop.

When a message arrives from the host via the root transputer, it is passed to a CASE statement where the appropriate action is taken. This is generally either a simple command changing a parameter value such as integration time or number of coadds, or a command to take a frame using the current parameter settings. During the operation of taking a frame, a number of messages will pass to and fro between the acquisition transputers and the clock generator, then data ready messages and data packets will go back to the root at the end of the process. Only after the data packets have been sent will the clock generator return to its primary ALT loop and listen again, although between coadds it checks briefly for abort messages sent from the host.

This process is all explained in more depth in the documents on the clock generators (NSPN18) and the data-taking process (NSPN33).

5.5 Data acquisition transputers

The acquisition transputers in each camera section all have the same message passing scheme. Like the clock generators, each is a single process which idles in an ALT construct waiting for messages. In this case each has only two possible sources for a message.

Command messages from the host come via the clock generator. Each command is passed on to the next acquisition processor before being checked by a CASE statement for the required action. This is generally either a simple command changing a parameter value such as sampling mode, or a command to take a frame using the current parameter settings. During the operation of taking a frame, a number of messages will pass to and fro between the acquisition transputers and the clock generator, then data ready messages and data packets will go back to the root at the end of the process. Only after the data packets have been sent will the acquisition transputers return to their primary ALT loops and listen again, although between coadds they check briefly for abort messages sent from the host.

Replies and data packets come from the next processor in the chain and are passed back in the direction of the clock generator.

In each camera section the last acquisition transputer is connected to a link of the clock generator. This does two things: it allows us to write the same code for every acquisition transputer, since the last one now has somewhere to pass on commands in the same way as all the others, and it lets the clock generator know that all the acquisition processors have “seen” a command.

6 Troubleshooting

Troubleshooting the network is mostly performed using utility programs supplied with the Matchbox, which can confirm that all the physical links are in place and all the transputers are healthy. However we found that wasn't adequate for all situations. Sometimes a problem occurs because some component of the network has become stalled or crashed. Because almost every transputer is involved in some way in passing on messages to others, a problem on one transputer can prevent another from doing its job. When that happens it's useful to have some kind of post mortem tool available to tell us what is hanging up, before we reboot the system.

The tool we came up with is a command id called cid.test. This cid is routed through the network according to the value of its parameter (usually the cid alone is sufficient to route a command). Each transputer is also programmed to respond with an echo of the command if it has a specific parameter value (adding 1000 to the parameter value, for unremembered historical reasons). So the root transputer echoes cid.test with parameter 1001 if the parameter is sent as 1, and passes it on otherwise. The SCAM clock generator responds to parameter 2, and the SCAM acquisition transputers to 3 and 4. Before we move on to the spectrometer section, we use parameter 5 to check the loop-back from acquisition transputer 2 to the clock generator is working. Parameter 5 will come right back to the host unaltered, having gone around that loop.

The spectrometer clock generator responds with parameter 1006 when it gets a 6, and the acquisition transputers respond to parameters of 7 to 22. The loopback is probed with parameter 23. The two serial transputers respond to 24 and 25, and the motor controllers to 26, 27 and 28. Other values are ignored.

To understand how this is useful consider the situation where serial 1 is having problems getting an answer out of a piece of external hardware. Once serial 1 has received a command to do something, the working component is now busy and will not be listening for another command. That means if we send it one, the “from root” buffer (see Figure 3) will now be hung up, since it will be trying to pass that command on to the RS232 section. One symptom of that problem will be that commands sent to the *motor* transputers will not get a response. If we want to find out why that is we can send cid.test to each transputer down the chain in turn to see how far the chain is clear. This simple technique has proved very useful in eliminating wasted time looking in the wrong place for a problem.

The other use for cid.test is that we can insert a send of cid.test into our code at any place where we want to send some debug information back to the host (kind of like putting a printf in a C program).