George Brims

Revised January 1, 1999

NIRSPEC Software Programming Note 17.00
Root transputer

1 Introduction

The root transputer is the bridge between the host computer and the transputer network. Its main function is to act as a message exchange, distributing commands from the host to the other transputers, and collecting replies and data and passing them back to the host. It also performs some "housekeeping" functions, monitoring the air temperature at various points inside the electronics cabinets where it is housed, and controlling and monitoring the arc lamps and quartz-halogen bulbs in the calibration unit. (Packaged with the root transputer in the "housekeeping box" are two RS232 interface TRAMs, which take care of reading cryogenic temperatures and controlling 110V power via a Pulizzi power strip. Their programs are described in NSPN34)

2 Program structure

Program structure is shown in Figure 1, and is also discussed in NSPN21, which describes the whole network, the message passing philosophy, and the structure of all the transputer programs. NSPN08 also discusses host to transputer communications.

Recall that an occam program can have multiple parallel streams of execution, or processes (see NSPN22 for a guide to programming in occam). Processes communicate over channels, which can be declared over real (processor to processor) or virtual (process to process in the same transputer) links.

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.

In order for the "from host" process to reply to the host, it needs to send its message via the "to host" buffer. The "from host" buffer is 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`.

The second buffer process ("to host") is a many-to-one process monitoring the three channels from the three physical branches of the network, and the virtual channel `buffer2buffer` from the other process. Whenever a message or data packet comes in, the process re-packages it for transmission

to the host. If the message contains a "data ready" `cid` from either camera section, the process enters a loop where it passes 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.

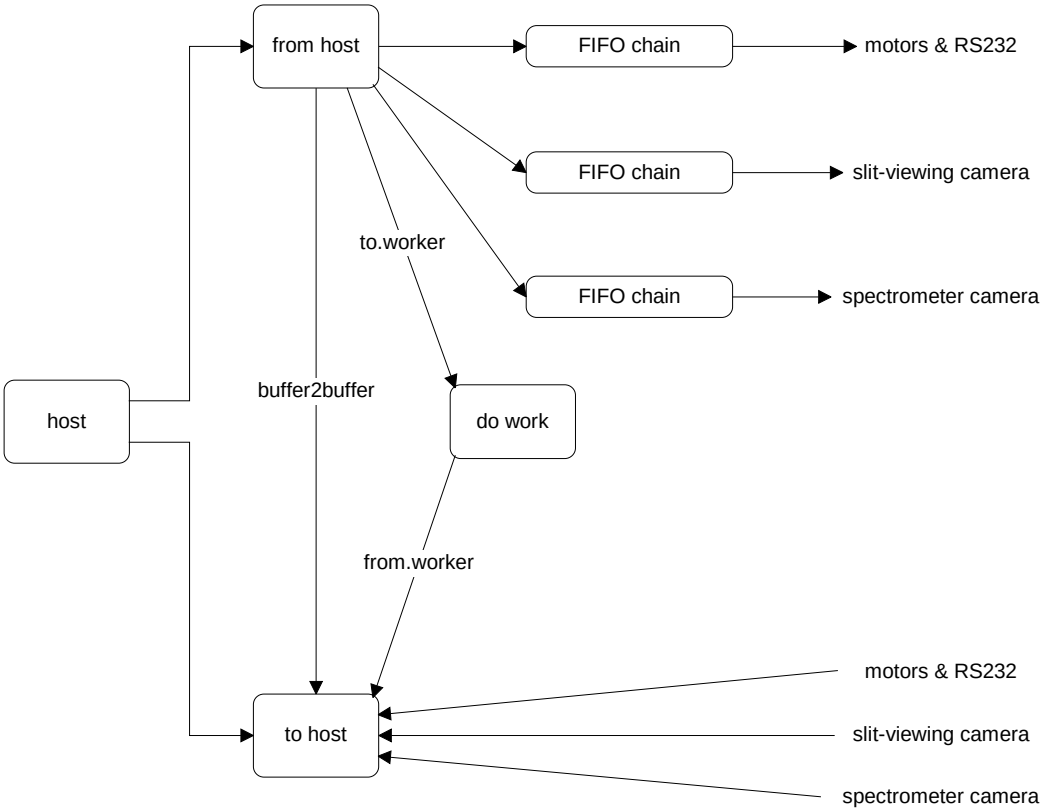

Figure 1: Root program structure

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 `to.worker`, and replies go to the "to host" buffer over virtual link `from.worker`.

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 the physical link to the other processor. These FIFO chains of processes simply give us somewhere for messages to stack up if one of the other transputers becomes busy or hangs up, and

it stops accepting messages. As more and more messages are sent to a busy processor from the host, they will eventually back up until the from.host buffer is also stalled, unable to pass on the last message. At that point the problem propagates to the host program, since its attempts to send (even things intended for other branches of the network) are now blocked.

3 Process details

3.1 From host buffer

The from host buffer is an infinite loop which continually monitors the host link for messages. Messages arrive in the "counted array" format — an integer followed by a block of bytes of size given by the integer. The format of the block is a byte command identifier (`cid`), followed by a two-byte integer denoting the length of the rest of the message, which is an ASCII string representation of an integer. The ASCII-encoded integer is the parameter (`param`) of the `cid`. The process extracts the `cid` and unscrambles the `param`,. It then uses a (very long) CASE statement to examine the `cid`. Every `cid` is unique to its destination, so it can then be sent onwards (with its parameter) over the appropriate link. Messages for any other transputer go to the first process in the FIFO chain for that branch of the network. A `cid` for the root transputer itself will go over the `to.worker` link to the worker process.

3.2 To host buffer

This process is a little more complicated, since it has to read from multiple inputs. It is an infinite loop which is headed by an ALT construct. The ALT construct lets a transputer process look at multiple possible sources of input (channels or timers), servicing whichever one has input first. In this case the four possible sources of input are the channels from the three branches of the physical network, plus the channel `from.worker` from its own worker process. Each input is taken in, then dealt with according to which kind of message it is and what `cid` it carries.

Whenever a message is received from another process, it can be of two forms. Either it is a simple message (`cid` plus `param`), or a data block. If it is a simple message and the `cid` isn't a "data ready" message from either camera section, it is passed to the routine `data.to.host`, which packs it into a packet for transmission to the host. The format in this direction is the same as input from the host, so the parameter is converted to a string of ASCII bytes.

If the message carries a "data ready" `cid`, the process enters a code section where it deals with the data blocks which will immediately come in over the same channel. The frame from the 256^2 slit-viewing camera will arrive as 2 blocks of 32768 pixel values (each 32 bits). The root transputer code reformats these two blocks into a single frame of 65536 pixels, then passes on the frame ready message. It then passes the 65536 pixel frame to the routine `data.to.host`, which breaks it up into smaller blocks and transmits them to the host. This is necessary because the limit for the physical link through the MatchBox to the host is 4096 bytes at a time.

In the case of the spectrometer camera, there will be 16 blocks, each of 65536 pixels. For each block, the process sends on the data ready message, with the `param` set to 1 through 16, followed by the block itself, split into smaller blocks by routine `data.to.host`. In this case the host code does the re-assembly of the data from the different acquisition transputers into a single rectangular frame. There isn't enough memory in the root transputer, or any transputer in the system, to hold a whole 1024^2 frame.

3.3 Worker process

The worker process only handles a few functions. It reads the temperatures in the electronics cabinets via Dallas Semiconductor DS1820 sensor chips, and controls and monitors the calibration unit lamps. The structure is an endless loop headed by an ALT statement. The two input to the ALT are a timer which trigger reading of the next DS1820 sensor, and the input channel `to.worker` which takes in the `cid` and `param` of the small number of commands which are routed to it.

3.3.1 Temperature sensor reading

The reading of the DS1820 sensors is performed at set intervals, and can be turned on or off from the host by sending `cid` value `cid.sensor.read`. If the parameter is zero, temperature reading is turned off by setting the flag `read.sensors` to FALSE. If it has a positive value the flag is set TRUE, and the interval in seconds between readings is set to the value of the parameter. Each temperature is returned as the sensor number times 1000, plus the returned sensor reading, which is in half degrees Celsius. So for example sensor 9, 25.5 degrees would come back as 9051.

There are two other `cid` values related to the sensors. The first is `cid.sensors.reset`, which resets all the sensors on the line. This is mostly used as a diagnostic. If this command is echoed to the host with parameter 1, at least one sensor is responding to the reset. If it's 0, there is probably a connection problem.

The other sensor command is `cid.sensors.getid`, which gets the unique ROM id of a new sensor chip. Each DS1820 has a 64-bit identifier programed into its ROM at the factory, and will only give back data if addressed with that id. This is not a frequent operation but the code is there if needed. Since the id is 64 bits, it can't be returned through our message convention in one go. Sending the request with param of 0 will return the lower 32 bits, and a param of 1 will return the high 32 bits. To use this command, the new sensor needs to be the only one connected to the root transputer.

3.3.2 Lamp control and monitoring

Control of the lamps is extremely simple. The DAQ17 transputer board used for the root transputer has 4 general-purpose I/O ports mapped to registers. The two different sets of lamps — neon, argon, xenon and krypton arc lamps and three quartz-halogen bulbs — are controlled via separate ports, by setting on or off the appropriate output bits. The sensor circuitry, which tells us that a lamp has

actually turned on or off when commanded, is read back via another register. There are two separate `cid` values, `cid.lamps.command` and `cid.lamps.status`. The bit pattern for controlling the lamps is sent to the former, and the response will be the status `cid` followed by the bit pattern read from the sensors. If you simply want to know what is on or off without commanding any changes, send `cid.lamps.status` with any parameter value and the sensor readings will come back.

3.4 FIFO buffer processes

Each FIFO buffer chain works in exactly the same way, so I will use the spectrometer camera chain as the example. We declare an array of 8 virtual channels with the declaration

```
[8]CHAN OF TRANS2TRANS to.spec.fifos :
```

When there is a message from the host to go to the spectrometer camera transputers, the from host process will write

```
to.spec.fifos[0] ! msg; cid; param
```

The first buffer process will get this message on channel `to.spec.fifos[0]` and pass it along on channel `to.spec.fifos[1]`. The whole code for this buffer process is

```
BYTE cid :
INT param :
SEQ
  WHILE TRUE
    SEQ
      to.spec.fifos[0] ? CASE
        msg; cid; param
          to.spec.fifos[1] ! msg; cid; param
```

All this process ever does is read whatever message arrives on the input channel and write it straight to the output channel. The next process in the chain similarly receives messages on channel `to.spec.fifos[1]` and sends them on over channel `to.spec.fifos[2]`.

The last FIFO process takes in messages on channel `to.spec.fifos[7]` and sends out on channel `to.spec`, which is a channel over the physical link to the spectrometer camera clock generator transputer.