George Brims

Revised January 14, 1999

NIRSPEC Software Programming Note 33.00

The data acquisition process

1 Introduction

The acquisition of data from the IR arrays in NIRSPEC is a complex process, involving interactions between transputer boards of two different types, the analog hardware, and the array itself. The two types of transputer boards and their programs are described much more fully elsewhere (NEAN04 & 05 for the hardware and NSPN18 & 19 for the programs), but this note describes the whole process so that the interactions and dependencies can be understood.

2 Hardware description

The essential features of the hardware are described in this section, though only as far as needed to understand the software.

2.1 Clock generator boards (DAQ17)

The T805 transputer on each DAQ17 board sees the outside world as a selection of registers where output data are loaded and input data read back.

Clock waveforms are generated by writing a series of 32-bit words to a parallel output-only port. There are two registers feeding data to the physical port, as well as various control registers.

Most of the clock pattern is sent out through a register which feeds a first-in-first-out (FIFO) buffer, 32 bits wide by 16384 deep. The transputer loads a sub-section of the waveform into the FIFO register and a repeat count into another register, then writes to a control register to command the hardware to send the data out. This scheme takes advantage of the repetitive nature of the patterns required by the IR arrays to simplify programming. The timing of the clock output is set completely by hardware, and can be much faster than the transputer itself can generate, as long as the waveform is composed and loaded into the FIFO buffer in advance.

As well as the repeated part, the arrays always require some clock pulses at the start of the waveform, and sometimes a sequence at the end. In this case output is sent word by word through the other register that feeds the same physical connectors. The start and end sequences are usually short and don't need to be very fast, so it doesn't matter that writing out this way is a bit slower (approx 320ns per time interval rather than 200ns through the FIFO buffer).

Most lines of the waveform go from the board's front panel connectors to the level shifter board, with two exceptions. Two lines are split off to a pair of BNC connectors and fed to the interface board in the analog crate. These are the `convert` signal for the A-D converters, and the `select` signal. There are two A-Ds on each analog board, but just one output connector. `select` (a level rather than a pulse) determines which A-D output is seen at the connector. Eight of the clock output lines are also copied to pins on the VME bus connector. We use one to send the `read.data` clock pulse to the acquisition boards, which tells them to read in the next set of pixel values from the A-Ds.

The DAQ17 boards also have four other (bi-directional) ports, most often used for driving motors and reading limit switches. When we use the boards for clocking arrays we use one of these ports to send data to serial D-A converters in the analog hardware (they set analog offset voltages for the pre-amps and control the detector bias for the Aladdin array), and another port to control the selectable gain and filter bandwidth on the analog boards (there are 4 choices for each, so we use 4 bits in total). These settings aren't changed during an integration so they don't enter into the discussion to follow.

2.2 Data acquisition boards (DAQ15)

The acquisition boards also use FIFO buffers mapped to registers, this time to take in data from the outside world. Each DAQ15 board has four T805 transputers. Each reads its data from a 512 deep FIFO. A pulse on the `read.data` line from the clock waveform triggers the reading of data into the FIFO. The FIFO's "half full" signal is fed to the transputer so as to trigger an interrupt. Each time there are 256 or more pixels in the FIFO, the transputer will read in 256 data words.

2.3 Pre-amp & A-D boards

These boards have two analog channels feeding two 16-bit A-D converters. When the A-Ds see the `convert` pulse from the clock generator they digitize the analog signal. The two digital data are sent through a multiplexer circuit to the front panel connector. The choice of which one is fed out at any given time is determined by whether the `select` signal from the clock generator is high or low.

2.4 Interface board

In order to isolate the digital and analog sections of the electronics and cut down on noise, the digital control signals (`convert`, `select` and the bits controlling gain and bandwidth) are passed through Burr-Brown ISO150 capacitive isolator chips on the interface board, which then feeds the signals along the analog system backplane to the pre-amp/A-D boards.

2.5 Bias board

The bias board generates the various voltages needed to power the IR array in each camera. It has serial D-A converters which set the pre-amp offset levels, and in the case of the 1024^2 Aladdin detector, another one to control the detector bias. These voltages are never changed during an integration.

2.6 Level shifter board

This board converts the output pulses from the clock generator to the appropriate voltage levels for the IR array detectors, which are quite different for the two types we use.

2.7 Interconnections

2.7.1 Transputer serial links

Each camera section of the instrument has one DAQ17 clock generator board. The slit-viewing camera uses two of the four transputers on a single DAQ15 acquisition board, while the spectrometer camera uses four DAQ15 boards for a total of 16 acquisition transputers. Each clock generator transputer has one serial link to the root transputer, which passes it commands from the host. The acquisition transputers are joined in a daisy-chain, each end of which has a link to the clock generator transputer.

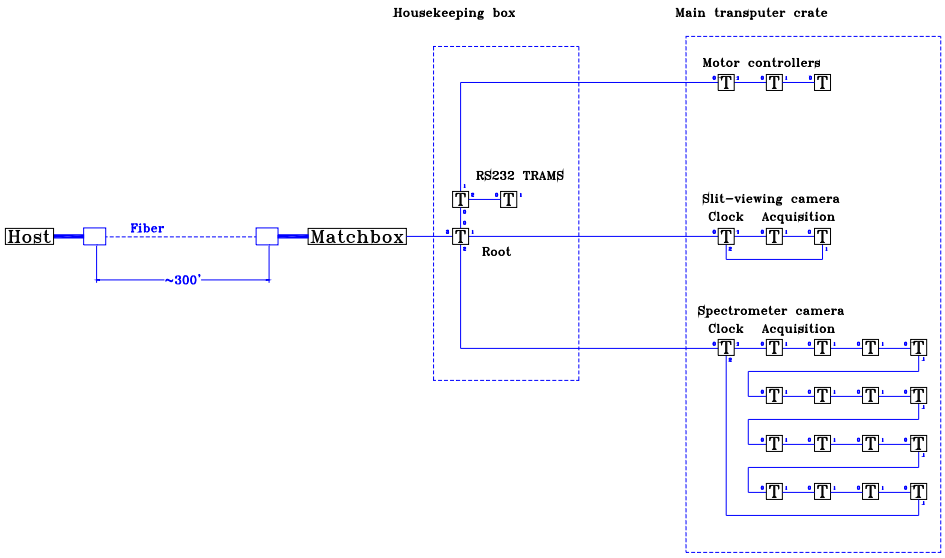

Figure 1: Transputer network

2.7.2 Clock signals

The clock signals go from the clock generator front panel to the level shifter, except for the `convert` and `select` lines, which go via the interface board to the pre-amp/A-D boards, and the `read.data` line which goes along the digital backplane to the acquisition boards. Level-shifted clock pulses go from each level shifter to the array.

2.7.3 Control signals

The serial data controlling the D-A converters for analog offset and detector bias go from one DAQ17 I/O port to the bias board, and the 4 bits controlling analog gain and bandwidth go from another DAQ17 I/O port to the interface board.

3 Acquisition sequence

3.1 Introduction

Now that I've given you some (probably too much) background on the hardware, it's time to describe what happens when we take a frame.

The most important interaction in the whole system is the one that gets data into the DAQ15 boards. The clock waveform includes pulses sent to the acquisition transputers on their `read.data` lines, feeding pixel values into the input FIFO buffers. The acquisition transputers read in data whenever there are enough pixel values in their FIFO buffers. The acquisition transputer has no idea when these pixel values will appear, but just reads until it has the right number of pixels in memory. In fact, the analog hardware doesn't even need to be switched on for this process to work.

I will describe what happens on the spectrometer camera, but the process is almost identical on the SCAM side (I will mention the few simple exceptions). Wherever I mention a variable name in the spectrometer side containing the character sequence `.spec`, the SCAM variable will be the same with `.scam` instead. (Note that the use of dots in variable names in occam has nothing to do with structures as they would in C. We simply use them as separators when stringing words or phrases together to make a meaningful variable name.)

A word about the structure of the two types of program, clock generator and acquisition: both are single process programs. There is no parallel processing going on within any one processor. The parallelism here is in the programs running simultaneously on the different transputers and interacting with each other via messages over the serial links and interactions with the hardware.

3.2 Setup

Before we take a frame there are a number of parameters which may need to be set (though the host software sends a set of default values when it first runs). These are integration time, sampling mode (single sampling, double correlated sampling or multiple correlated sampling), number of co-adds, and sample rate (how fast we take pixels). For multiple correlated sampling the transputers also need to know the required number of samples. Each parameter has a keyword in the host code, and is sent to the transputers as a command identifier (`cid`) with its associated parameter value (`param`). Messages consisting of `cid/param` pairs are sent to select the values for the next frame. In most cases this just results in a value being changed, and used once the integration starts. One special case is the sample rate. If the rate is changed the clock program immediately calls its routine `makewave` to re-generate the stored waveforms, since the output rate is changed by making the waveform longer or shorter.

3.3 Taking a frame

Taking a frame is triggered by the host computer sending a message with command identifier `go.spec` (or `go.scam`). When the clock generator gets this command, it sets a couple of values (`coadd.num` = 0 and `not.abort` = TRUE), then passes the command on to the first acquisition transputer, which passes it on down the chain of acquisition transputers to the last one.

The clock generator and acquisition transputers then both do a CASE on the value of `sampmode`, and go into the appropriate code. In the clock generator there is a 60-80 line section of inline code for each of the three modes, but in the acquisition transputers the code is huge so it's been split off into subroutines. Although there are 3 observing modes there are only two data-taking routines: one for single sampling (`ssample`) and one to handle both double correlated and multiple correlated sampling (`dcorrsample`). These two modes are the same except for the number of samples per read, 1 for DCS or more than 1 for MCDS.

In each type of sampling there is a period during each co-add when the clock generator is idle, waiting for the integration time to expire. During this period it is also open to receiving messages from the host. Only one command identifier will have any effect — the abort message — while anything else is read and discarded. If the abort message is received it is passed to the acquisition transputers, which then discard their data and abandon the observation too.

I will describe each data-taking mode separately. Each mode is a little more complex than the last so it's probably a good idea to read them in order.

3.4 Single sampling

On entering the code section for single sampling, the clock generator enters a WHILE loop, which it will execute once for each co-add, incrementing `coadd.num` each time. There are two ways for the program to exit this loop. The usual way is for it to complete the requested number of co-adds (`coadds.spec`) and end the observation. Occasionally it will exit because the flag `not.abort` has been set false by an abort message from the host.

In the acquisition transputers, routine `ssample` clears the data buffer `frame` and the FIFO hardware buffer, and sets the flag `coadding` to TRUE, before entering a WHILE loop on flag `coadding`. This loop will also normally be executed once per co-add. The two reasons for `coadding` becoming FALSE, forcing an exit from the loop are again that the requested number of co-adds is done, or the host has requested an abort. The loop starts with a message input from the clock generator. There are three possible commands; take another coadd, abort the frame, or complete the frame. The acquisition transputer has no advance knowledge of the number of coadds it will do, and in fact it doesn't even keep track of how many it has done.

3.4.1 Normal completion

Each time it starts its co-adding loop, the clock generator sends a message to the acquisition transputers, with command identifier `cid.daq.integ.start.spec`. This is the command that tells the acquisition transputers that another co-add is to start, so they should expect data.

The acquisition transputers are waiting at the top of their loop for any incoming message. When a message arrives, each one passes it on to the next, then parses the command with a CASE statement. If the command is `cid.daq.integ.start.spec`, it then goes into an enumerated loop where it sets up an interrupt on the half full flag of the FIFO input buffer. This interrupt will go off once data starts to arrive.

Once the `integ.start` command has passed from the last acquisition transputer to the clock generator, the clock generator now knows that all the acquisition transputers are ready for data, so it can start the co-add. It takes a time-stamp from the on-chip clock, and clears the detector array using routine `global.reset` (`reset` on the SCAM). It then waits for the integration time to expire. Since during the integration time the clock generator is sitting idle, it is here we allow messages from the host to interrupt the flow. We use the Occam ALT construct to handle this situation. The ALT allows the clock generator to sit idle and wait for input from either the time expiring or a message arriving. (We will talk about what happens when a message arrives in the next subsection).

Once the integration time expires, the clock generator calls routine `clockwave` to generate the readout clock pattern. This clocks through the array, selecting the pixels in sequence, and latches a stream of pixels into the acquisition transputers' FIFO input buffers. Finally it increments the counter `coadd.num`.

The acquisition transputers are waiting for these pixels to arrive. Every time the input buffers reach half full the acquisition transputers are interrupted (the line `event ? event.byte`), then read in data. After each block of 256 pixels is read, each transputer goes through a long sequence of lines to co-add these new values into the buffer, then goes back to the top of the enumerated loop. It will execute this loop 256 times (128 times for the SCAM), so each acquisition transputer buffers 65536 pixels (32768 on the SCAM). The acquisition transputers then go back to the top of their `WHILE coadding` loop and wait for another command.

This whole sequence will repeat until the clock generator has counted up the right number of co-adds (`coadd.num` equals `coadd.spec`). It then exits its co-adding loop. Once out of the loop it checks whether `not.abort` is true. If it is, meaning the coadd loop ended normally, it sends command `cid.daq.integ.end.spec` to the acquisition transputers and waits for it to be passed back.

On receiving this `integ.end` command, each acquisition transputer passes it on to the next, and so back to the clock generator. Once the clock generator gets this acknowledgment it has completed

all it will do for this frame, and goes back to its main program section where it idles waiting for messages from the host or messages or data from the acquisition transputers.

After passing on the `integ.end` command, each acquisition transputer sets `coadding` to FALSE, which will end the loop once it gets back to the WHILE at the top. First though, the data have to be sent to the host. Each acquisition transputer has a unique number, `tnum`, which is downloaded to it from the host at run time, so it knows where it is in the chain. The first acquisition transputer has a `tnum` of zero, and so on up to 15 (just 0 or 1 in the SCAM), The first one sends a message with command identifier `cid.frame.ready.spec` to the clock generator, which passes it back to the host computer to alert it that data are on the way. The first acquisition transputer then sends its data to the clock generator, which then passes it back to the host. It then enters a loop where it will take in the data blocks from the other 15 acquisition transputers and pass them along. It knows from the value of `tnum` how many times it will have to do this. The next one along has a `tnum` of 1, so it will send its own data, then pass along only 14 blocks from the others, and so on down the line to the last one, which sends its own data, but doesn't have to pass along any.

Having sent back their data, the acquisition transputers too can go back to their main program loop where they wait for further commands.

3.4.2 Abort

If the host sends an abort command during an integration, it will be received by the clock generator during the idle period of the integration between resetting the array and clocking out the data. If that happens, the clock generator has to do a couple of things. It can't just pass the message straight on to the acquisition transputers, since at this point they are waiting for incoming data, so first it clocks the array so that the acquisition transputers will complete the current co-add and go back to listening for messages. It doesn't matter that this is happening before the end of the allotted integration time, since the data will be trashed anyway. It then sets the flag `not.abort` to FALSE, so it will exit its own loop, then acknowledges the abort message with an echo to the host. Once it has exited the co-adding loop, it checks the value of `not.abort`. If it's false then it passes the message along to the acquisition transputers and waits for the acknowledgment that they have all received it.

When the acquisition transputers get the abort message, each passes the message along to the next, then sets flag `coadding` to false. This will cause it to drop out of its co-adding loop without ever sending the data back. The data are effectively discarded since the frame buffer is cleared at the start of the next observation.

Clock generator code for a single sampling frame

```
cid.go.spec                                  -- take an observation
  SEQ
    coadd.num := 0
    not.abort := TRUE
    to.daq ! msg; cid; param                 -- pass the cid.go command to the acquisition transputers
    CASE sampmode                            -- which sampling mode is to be used for this observation?

      single.samp                            -- single sample mode (reset, wait integration time, read)
        SEQ
          WHILE ((coadd.num < coadds.spec) AND not.abort)    -- loop through requested number of coadds
            SEQ
             to.daq ! msg; cid.daq.integ.start.spec; param  -- tells daqs to start this coadd
             loop.back ? CASE                    -- waiting for DAQ ready signal (echos integ.start)
               msg; cid; param
                 SKIP
             clock ? time.now                    -- store current time
             global.reset ()                     -- reset the array
             loadwave (read)                     -- load up fifo with read frame for end of itime
             ALT                                 -- now wait for end of integration *or* abort message

               clock ? AFTER time.now PLUS itime.spec.ticks -- itime has elapsed
                 SEQ
                   clockwave (read)              -- read out the array
                   coadd.num := coadd.num + 1    -- keep a count of coadds

               from.root ? CASE                  -- received command from root
                 msg; cid; param
                   SEQ
                     CASE cid                    -- check the command
                       cid.abort.spec            -- command is abort without saving data
                         SEQ
                           clockwave (read)      -- clock the array so acquisition boards don't hang up
                           coadd.num := coadd.num + 1
                           not.abort := FALSE    -- FALSE means we will fall out of the coadding loop
                           to.root ! msg; cid.abort.spec ; spec -- acknowledge
                       ELSE
                         SKIP
          IF
            not.abort
              SEQ
                to.daq ! msg; cid.daq.integ.end.spec  ; 0
                loop.back ? CASE                 -- receives the echo of the cid from the last DAQ
                  msg; cid; param
                    SKIP
            TRUE
              SEQ
                to.daq ! msg; cid.abort.spec ; 0    -- daqs will discard data when they see this
                loop.back ? CASE                 -- receives the echo of the cid from the last DAQ
                  msg; cid; param
                    SKIP
```

Acquisition code for a single sampling frame

```
PROC ssample (VAL INT tnum, CHAN OF TRANS2TRANS up, down, in, out)
  --
  --   s s a m p l e       We use this procedure to take data in single-sampling mode, with co-adding
  --                       if required.
  --
  BYTE    cid :
  INT     param :
  INT     index :
  INT     result :
  BOOL    coadding :
  BYTE    event.byte :
  INT     should.be :
  SEQ
    SEQ i = 0 FOR 65536
      frame[i] := 0                                     -- first clear the data array
    FifoRst    := 0                                     -- clear the FIFO incoming data buffer
    coadding := TRUE
    WHILE coadding                                      -- loop until coadding is set to FALSE
      SEQ
        up ? CASE                                       -- wait for a message from the clock...
          msg; cid; param
            SEQ
              down ! msg; cid; param                    -- ...and pass it on down the line
        CASE cid                                        -- check the message content
          cid.daq.integ.start.spec                      -- message is start of a coadd
            SEQ
              index := 0                                -- init pointer into data buffer
              SEQ i = 0 FOR 256                         -- take in our piece of the frame in 256 blocks
                SEQ
                  subframe IS [frame FROM index FOR 256] :   -- define block in frame array
                  SEQ
                    int.enable := half.full             -- we will interrupt on half full flag of FIFO
                    master := master.int.en             -- enable interrupts
                    event ? event.byte                  -- wait for interrupt event
                    result := int.poll /\ half.full     -- mask half full bit in interrupt result reg.
                    should.be := half.full
                    IF
                      result <> should.be               -- check interupt really was half full
                        CAUSEERROR()                    -- if not it's a fatal problem, bomb out
                      TRUE
                        SKIP                            -- otherwise we're fine, go on and read in data
                    int.enable := 0                     -- turn off interrupts
                    [dma FROM 0 FOR 256] := [FifoRdLo16 FROM 0 FOR 256]  -- DMA in the block of data
                    subframe[  0] := subframe[  0] + dma[  0] -- coadd most recent data values into buffer
                    subframe[  1] := subframe[  1] + dma[  1] -- we do it longhand like this for speed
                    subframe[  2] := subframe[  2] + dma[  2]
                    subframe[  3] := subframe[  3] + dma[  3]
                    :
                    :
                    subframe[253] := subframe[253] + dma[253]
                    subframe[254] := subframe[254] + dma[254]
                    subframe[255] := subframe[255] + dma[255]
                  index := index + 256
              FifoRst := 0                              -- empty the FIFO at the end of the frame

          cid.abort.spec                                -- command is to abort integration
            SEQ
              coadding := FALSE                         -- this will drop us out of "WHILE coadding" loop

          cid.daq.integ.end.spec                        -- command is end of integration
            SEQ
              coadding := FALSE                         -- this will drop us out of "WHILE coadding" loop
              IF
                tnum = 0                                -- first transputer only, send "frame ready" to root
                  SEQ
                    out ! msg; cid.frame.ready.spec; spec
                TRUE
                  SKIP
              out ! data; tnum; 65536::frame            -- send out frame data
              SEQ i = 0 FOR (15 - tnum)                 -- pass on data from all the downstream processors
                SEQ
                  in ? CASE
                    data; tid; frame.size::frame
                      out ! data; tid; frame.size::frame
:
```

3.5 Double correlated sampling

On entering the code section for double correlated sampling, the clock generator enters a WHILE loop, which it will execute once for each co-add, incrementing `coadd.num` each time. There are two ways for the program to exit this loop. The usual way is for it to complete the requested number of co-adds (`coadds.spec`) and end the observation. Occasionally it will exit because the flag `not.abort` has been set false by an abort message from the host.

In the acquisition transputers, routine `dcorrsample` is called with parameter `multi` equal to one (anything greater than one is multiple correlated sampling, described in the next section). The routine first clears the data buffer `frame` and the FIFO hardware buffer, and sets the flag `coadding` to TRUE, before entering a WHILE loop on flag `coadding`. This loop will also normally be executed once per co-add. The two reasons for `coadding` becoming FALSE, forcing an exit from the loop are again that the requested number of co-adds is done, or the host has requested an abort. The loop starts with a message input from the clock generator. There are three possible commands; take another coadd, abort the frame, or complete the frame. The acquisition transputer has no advance knowledge of the number of coadds it will do, and in fact it doesn't even keep track of how many it has done.

3.5.1 Normal completion

Each time it starts its co-adding loop, the clock generator sends a message to the acquisition transputers, with command identifier `cid.daq.integ.start.spec`. This is the command that tells the acquisition transputers that another co-add is to start, so they should expect data.

The acquisition transputers are waiting at the top of their `WHILE coadding` loop for any incoming message. When a message arrives, each one passes it on to the next, then parses the command with a CASE statement. If the command is `cid.daq.integ.start.spec`, it then goes into a loop which it will execute `multi` times, which in this case means just once, since this is DCS. Inside that loop is an enumerated loop where it sets up an interrupt on the half full flag of the FIFO input buffer. This interrupt will go off once data starts to arrive.

Once the `integ.start` command has passed from the last acquisition transputer to the clock generator, the clock generator now knows that all the acquisition transputers are ready for data, so it can start the co-add. It takes a time-stamp from the on-chip clock, and clears the detector array using routine `global.reset` (`reset` on the SCAM). Next, the clock generator calls routine `clockwave` to generate the readout clock pattern. This clocks through the array, selecting the pixels in sequence, and latches a stream of pixels into the acquisition transputers' FIFO input buffers.

The acquisition transputers are waiting for this first set of pixels to arrive. Since this set of samples is taken before the integration time, they are subtracted from the co-add buffer values. The set at the end of the integration will be added, so the resulting data will be the difference between the two sets. Every time the input FIFO buffers reach half full (256 pixel values) the hardware interrupts the

acquisition transputers (the line `event ? event.byte`), which then read in data. After each block of 256 pixels is read, each transputer goes through a long sequence of lines to subtract the values from the buffer, then goes back to the top of the enumerated loop. It will execute this loop 256 times (128 times for the SCAM), so each acquisition transputer buffers 65536 pixels (32768 on the SCAM). The acquisition transputers now wait for the next set of pixels.

The clock generator then waits for the integration time to expire. Since during the integration time the clock generator is sitting idle, it is here we allow messages from the host to interrupt the flow. We use the Occam ALT construct to handle this situation. The ALT allows the clock generator to sit idle and wait for input from either the time expiring or a message arriving. (We will talk about what happens when a message arrives in the next subsection).

Once the integration time expires, the clock generator calls routine `clockwave` to generate another readout clock pattern. This clocks through the array again, and latches a second stream of pixels into the acquisition transputers' FIFO input buffers. Finally it increments the counter `coadd.num`.

The acquisition transputers have entered another loop on the variable `multi` (which again they will do just once for DCS) and are waiting at the top of a second enumerated loop for this second set of pixels to arrive. Again the half full signal from the FIFO input buffers generates interrupts, and the transputers read in 256 pixels on each interrupt. After a block of 256 pixels is read, each transputer goes through a long sequence of lines to add these new values to those already in the buffer, then goes back to the top of the enumerated loop. As before it will execute this loop 256 times (128 times for the SCAM). The acquisition transputers then go back to the top of their `WHILE coadding` loop and wait for another command.

This whole sequence will repeat until the clock generator has counted up the right number of co-adds (`coadd.num` equals `coadd.spec`). It then exits its co-adding loop. Once out of the loop it checks whether `not.abort` is true. If it is, meaning the coadd loop ended normally, it sends command `cid.daq.integ.end.spec` to the acquisition transputers and waits for it to be passed back.

On receiving this `integ.end` command, each acquisition transputer passes it on to the next, and so back to the clock generator. Once the clock generator gets this acknowledgment it has completed all it will do for this frame, and goes back to its main program section where it idles waiting for messages from the host or messages or data from the acquisition transputers.

After passing on the `integ.end` command, each acquisition transputer sets `coadding` to FALSE, which will end the loop once it gets back to the WHILE at the top. First though, the data have to be sent to the host. Each acquisition transputer has a unique number, `tnum`, which is downloaded to it from the host at run time, so it knows where it is in the chain. The first acquisition transputer has a `tnum` of zero, and so on up to 15 (just 0 or 1 in the SCAM), The first one sends a message with command identifier `cid.frame.ready.spec` to the clock generator, which passes it back to the host computer to alert it that data are on the way. The first acquisition transputer then sends its data to the clock generator, which then passes it back to the host. It then enters a loop where it will take

in the data blocks from the other 15 acquisition transputers and pass them along. It knows from the value of `tnum` how many times it will have to do this. The next one along has a `tnum` of 1, so it will send its own data, then pass along only 14 blocks from the others, and so on down the line to the last one, which sends its own data, but doesn't have to pass along any.

Having sent back their data, the acquisition transputers too can go back to their main program loop where they wait for further commands.

3.5.2 Abort

If the host sends an abort command during an integration, it will be received by the clock generator during the idle period of the integration between the first and second times it clocks out the data. If that happens, the clock generator has to do a couple of things. It can't just pass the message straight on to the acquisition transputers, since at this point they are waiting for the second set of incoming data, so first it clocks the array so that the acquisition transputers will complete the current co-add and go back to listening for messages. It doesn't matter that this is happening before the end of the allotted integration time, since the data will be trashed anyway. It then sets the flag `not.abort` to FALSE, so it will exit its own loop, then acknowledges the abort message with an echo to the host. Once it has exited the co-adding loop, it checks the value of `not.abort`. If it's false then it passes the message along to the acquisition transputers and waits for the acknowledgment that they have all received it.

When the acquisition transputers get the abort message, each passes the message along to the next, then sets flag `coadding` to false. This will cause it to drop out of its co-adding loop without ever sending the data back. The data are effectively discarded since the frame buffer is cleared at the start of the next observation.

Clock generator code for a double correlated sampling frame

```
cid.go.spec                                  -- take an observation
  SEQ
    coadd.num := 0
    not.abort := TRUE
    to.daq ! msg; cid; param                 -- pass the cid.go command to the acquisition transputers
    CASE sampmode                            -- which sampling mode is to be used for this observation?

      single.samp                            -- single sample mode (reset, wait integration time, read)
          <-------
                   single sampling code
                                      -------->

      correlated.double.samp       -- correlated double sample mode. sequence is reset, read 1st frame, wait
        SEQ                        -- integration time, read 2nd frame. signal is 2nd frame minus 1st frame.
          WHILE ((coadd.num < coadds.spec ) AND not.abort)
            SEQ
              to.daq ! msg; cid.daq.integ.start.spec; param  -- tells daqs to start this coadd
              loop.back ? CASE                               -- wait for loopback of integ.start
                 msg; cid; param
                   SKIP
              loadwave (read)                                -- load FIFO with read frame
              global.reset ()
              clock ? time.now                               -- start integration timer
              clockwave (read)                               -- read first frame
              ALT                                            -- now wait for end of integration *or* abort message
                clock ? AFTER time.now PLUS itime.spec.ticks -- integration time has expired
                SEQ
                  clockwave (read)                           -- read second frame
                  coadd.num := coadd.num + 1                 -- keep track of coadd count

                  from.root ? CASE                             -- message from host
                    msg; cid; param
                      SEQ
                        CASE cid                               -- check the command
                          cid.abort.spec                       -- command is abort (discard data)
                            SEQ
                              clockwave (read)                 -- do read frame so daqs can finish their loop
                              not.abort := FALSE               -- FALSE means we will drop out of the coadding loop
                              to.root ! msg; cid.abort.spec ; spec  -- acknowledge message
                          ELSE
                            SKIP
                                                               -- end of WHILE loop

                                                               -- finish the observation
          IF
            not.abort                                          -- we didn't end because of abort message
              SEQ
                to.daq ! msg; cid.daq.integ.end.spec ; 0     -- tell the daqs we're done
                loop.back ? CASE                               -- wait for the echo of the integ.end from the last DAQ
                  msg; cid; param
                    SKIP
            TRUE                                               -- we *did* end on an abort
              SEQ
                to.daq ! msg; cid.abort.spec ; 0               -- tell the daqs we're done
                loop.back ? CASE                               -- wait for the echo of the cid.abort from the last DAQ
                  msg; cid; param
                    SKIP
```

Acquisition code for a double or multiple correlated sampling frame

```
PROC dcorrsample (VAL INT tnum, multi, CHAN OF TRANS2TRANS up, down, in, out)
  --
  --   d c o r r s a m p l e      We use this procedure to take data in double correlated sampling mode with co-adding.
  --                              Also does multiple correlated sampling when multi > 1.
  BYTE    cid :
  INT     index :
  INT     param :
  INT     result :
  BOOL    coadding :
  BYTE    event.byte :
  INT     should.be :
  SEQ
    SEQ i = 0 FOR 65536                                        --  First clear the data array
      frame[i] := 0
    FifoRst  := 0                                              -- clear the FIFO incoming data buffer
    coadding := TRUE
    WHILE coadding                                             -- loop until coadding is set to false
      SEQ
        up ? CASE                                              -- wait for a message from the host...
          msg; cid; param
            SEQ
              down ! msg; cid; param                           -- ...and pass it on downstream
        CASE cid                                               -- now check message content
          cid.daq.integ.start.spec                             -- message is start of a coadd
            SEQ
              SEQ samples = 0 FOR multi                        -- read frame "before" data multi times at start of itime
                SEQ
                  index := 0                                   -- init pointer into input data array
                  SEQ i = 0 FOR 256                            -- read data in 256 blocks of 256 pixels
                    SEQ
                      subframe IS [frame FROM index FOR 256] :   -- define block in frame array
                      SEQ
                        int.enable := half.full                -- specify interrupt on half full flag of FIFO buffer
                        master := master.int.en                -- enable interrupts
                        event ? event.byte                     -- wait for interrupt event
                        result := int.poll /\ half.full        -- mask half full bit in interrupt result register
                        should.be := half.full
                        IF
                          result <> should.be                  -- check interrupt really was half full
                            CAUSEERROR()                       -- if not we have a fatal error, bomb out
                          TRUE
                            SKIP
                        int.enable := 0                        -- turn off interrupts
                        [dma FROM 0 FOR 256] := [FifoRdLo16 FROM 0 FOR 256]  -- read in chunk of data
                        subframe[  0] := subframe[  0] - dma[  0]  -- subtract "before" samples from buffer
                         :
                        subframe[255] := subframe[255] - dma[255]
                      index := index + 256
                  FifoRst := 0                                 -- empty the FIFO at the end of the frame sequence
              SEQ samples = 0 FOR multi                        -- read frame "after" data multi times at end of itime
                SEQ
                  index := 0                                   -- init pointer into incoming data array
                  SEQ i = 0 FOR 256                            -- again we will take in 256 blocks of 256 pixels
                    SEQ
                      subframe IS [frame FROM index FOR 256] :   -- define block in frame array
                      SEQ
                        int.enable := half.full                -- set up to interrupt on FIFO half full flag
                        master := master.int.en                -- enable interrupts
                        event ? event.byte                     -- wait for interrupt event
                        result := int.poll /\ half.full        -- mask half full bit in interrupt results register
                        should.be := half.full
                        IF
                          result <> should.be                  -- check it really was half full that caused the interrupt
                            CAUSEERROR()                       -- if not we have a fatal error so bomb out
                          TRUE
                            SKIP                               -- otherwise go on to read data
                        int.enable := 0                        -- disable the interrupt
                        [dma FROM 0 FOR 256] := [FifoRdLo16 FROM 0 FOR 256]  -- read in block of data
                        subframe[  0] := subframe[  0] + dma[  0]  -- now coadd the "after" pixels into the buffer
                         :
                        subframe[255] := subframe[255] + dma[255]
                      index := index + 256                     -- increment pointer
                  FifoRst := 0                                 -- empty the FIFO at the end of a frame
          cid.abort.spec                                       -- command is abort observation
            SEQ
              coadding := FALSE                                -- this will drop us out of "WHILE coadding"loop
          cid.daq.integ.end.spec                               -- command is end of integration
            SEQ
              coadding := FALSE                                -- this will drop us out of "WHILE coadding"loop
              SEQ i=0 FOR 65536
                frame[i] := frame[i] / multi                   -- divide frame data by number of reads
              IF
                tnum = 0                                       -- first transputer only, send "frame ready" to root
                  out ! msg; cid.frame.ready.spec; spec
                TRUE
                  SKIP
              out ! data; tnum; 65536::frame                   -- send out frame data
              SEQ i = 0 FOR (15 - tnum)                        -- pass on data from downstream processors
                SEQ
                  in ? CASE
                    data; tid; frame.size::frame
                      out ! data; tid; frame.size::frame
:
```

3.6 Multiple correlated sampling (aka Fowler sampling)

On entering the code section for multiple correlated sampling, the clock generator enters a WHILE loop, which it will execute once for each co-add, incrementing `coadd.num` each time. There are two ways for the program to exit this loop. The usual way is for it to complete the requested number of co-adds (`coadds.spec`) and end the observation. Occasionally it will exit because the flag `not.abort` has been set false by an abort message from the host.

In the acquisition transputers, routine `dcorrsample` is called with parameter `multi` equal to the number of requested coadds, which is some integer greater than one (the value one gives double correlated sampling, described in the previous section). The routine first clears the data buffer `frame` and the FIFO hardware buffer, and sets the flag `coadding` to TRUE, before entering a WHILE loop on flag `coadding`. This loop will also normally be executed once per co-add. The two reasons for `coadding` becoming FALSE, forcing an exit from the loop are again that the requested number of co-adds is done, or the host has requested an abort. The loop starts with a message input from the clock generator. There are three possible commands; take another coadd, abort the frame, or complete the frame. The acquisition transputer has no advance knowledge of the number of coadds it will do, and in fact it doesn't even keep track of how many it has done.

3.6.1 Normal completion

Each time it starts its co-adding loop, the clock generator sends a message to the acquisition transputers, with command identifier `cid.daq.integ.start.spec`. This is the command that tells the acquisition transputers that another co-add is to start, so they should expect data.

The acquisition transputers are waiting at the top of their `WHILE coadding` loop for any incoming message. When a message arrives, each one passes it on to the next, then parses the command with a CASE statement. If the command is `cid.daq.integ.start.spec`, it then goes into a loop which it will execute `multi` times. Inside that loop is an enumerated loop where it sets up an interrupt on the half full flag of the FIFO input buffer. This interrupt will go off once data starts to arrive.

Once the `integ.start` command has passed from the last acquisition transputer to the clock generator, the clock generator now knows that all the acquisition transputers are ready for data, so it can start the co-add. It takes a time-stamp from the on-chip clock, and clears the detector array using routine `global.reset` (`reset` on the SCAM). Next, the clock generator goes into an enumerated loop which is executed `multi` times. Each time through this loop it calls routine `clockwave` to generate the readout clock pattern, then pauses briefly before doing it again. This clocks through the array, selecting the pixels in sequence, and latches a stream of pixels into the acquisition transputers' FIFO input buffers.

The acquisition transputers are waiting in two nested enumerated loops for pixel data to arrive. The outer loop is executed `multi` times. In the inner enumerated loop, it reads in the pixel data. Since

these pixel values are taken before the integration time, they are subtracted from the co-add buffer values. The pixel data at the end of the integration will be added, so the resulting data will be the difference between the two sets. Every time the input FIFO buffers reach half full (256 pixel values) the hardware interrupts the acquisition transputers (the line `event ? event.byte`), which then read in data. After each block of 256 pixels is read, each transputer goes through a long sequence of lines to subtract the values from the buffer, then goes back to the top of the enumerated loop. It will execute this loop 256 times (128 times for the SCAM), so each acquisition transputer buffers 65536 pixels (32768 on the SCAM). The acquisition transputers now wait for the next `multi` frames worth of pixels.

The clock generator then waits for the integration time to expire. Since during the integration time the clock generator is sitting idle, it is here we allow messages from the host to interrupt the flow. We use the Occam ALT construct to handle this situation. The ALT allows the clock generator to sit idle and wait for input from either the time expiring or a message arriving. (We will talk about what happens when a message arrives in the next subsection).

Once the integration time expires, the clock generator again goes into a loop where it calls routine `clockwave` to generate the readout clock pattern times. This latches a second stream of pixels into the acquisition transputers' FIFO input buffers. Finally it increments the counter `coadd.num`.

The acquisition transputers have entered another loop on the variable `multi` and are waiting at the top of the inner enumerated loop for this second set of pixels to arrive. Again the half full signal from the FIFO input buffers generates interrupts, and the transputers read in 256 pixels on each interrupt. After a block of 256 pixels is read, each transputer goes through a long sequence of lines to add these new values to those already in the buffer, then goes back to the top of the enumerated loop. As before it will execute this loop 256 times (128 times for the SCAM). Once the multi frames worth of pixels have been read in, this co-add is over, and the acquisition transputers then go back to the top of their `WHILE coadding` loop and wait for another command.

This whole sequence will repeat until the clock generator has counted up the right number of co-adds (`coadd.num` equals `coadd.spec`). It then exits its co-adding loop. Once out of the loop it checks whether `not.abort` is true. If it is, meaning the coadd loop ended normally, it sends command `cid.daq.integ.end.spec` to the acquisition transputers and waits for it to be passed back.

On receiving this `integ.end` command, each acquisition transputer passes it on to the next, and so back to the clock generator. Once the clock generator gets this acknowledgment it has completed all it will do for this frame, and goes back to its main program section where it idles waiting for messages from the host or messages or data from the acquisition transputers.

After passing on the `integ.end` command, each acquisition transputer sets `coadding` to FALSE, which will end the loop once it gets back to the WHILE at the top. First though, the data have to be sent to the host. Each acquisition transputer has a unique number, `tnum`, which is downloaded to it from the host at run time, so it knows where it is in the chain. The first acquisition transputer has

a `tnum` of zero, and so on up to 15 (just 0 or 1 in the SCAM), The first one sends a message with command identifier `cid.frame.ready.spec` to the clock generator, which passes it back to the host computer to alert it that data are on the way. The first acquisition transputer then sends its data to the clock generator, which then passes it back to the host. It then enters a loop where it will take in the data blocks from the other 15 acquisition transputers and pass them along. It knows from the value of `tnum` how many times it will have to do this. The next one along has a `tnum` of 1, so it will send its own data, then pass along only 14 blocks from the others, and so on down the line to the last one, which sends its own data, but doesn't have to pass along any.

Having sent back their data, the acquisition transputers too can go back to their main program loop where they wait for further commands.

3.6.2 Abort

If the host sends an abort command during an integration, it will be received by the clock generator during the idle period of the integration between the first and second times it clocks out the data. If that happens, the clock generator has to do a couple of things. It can't just pass the message straight on to the acquisition transputers, since at this point they are waiting for the second set of incoming data, so first it clocks the array `multi` times, so that the acquisition transputers will complete the current co-add and go back to listening for messages. It doesn't matter that this is happening before the end of the allotted integration time, since the data will be trashed anyway. It then sets the flag `not.abort` to FALSE, so it will exit its own loop, then acknowledges the abort message with an echo to the host. Once it has exited the co-adding loop, it checks the value of `not.abort`. If it's false then it passes the message along to the acquisition transputers and waits for the acknowledgment that they have all received it.

When the acquisition transputers get the abort message, each passes the message along to the next, then sets flag `coadding` to false. This will cause it to drop out of its co-adding loop without ever sending the data back. The data are effectively discarded since the frame buffer is cleared at the start of the next observation.

Clock generator code for a multiple correlated sample

```
cid.go.scam                                  -- take data
  SEQ
    coadd.num := 0
    not.abort := TRUE
    to.daq ! msg; cid; param                 -- pass command to the acquisition transputers so they get ready
    CASE sampmode                            -- now execute the section for the appropriate sample mode
      <--------
               Single sampling and DCS code here
                                              --------->
      mult.corr.sample
        SEQ
          WHILE ((coadd.num < coadds.scam ) AND not.abort)
            SEQ
              to.daq ! msg; cid.daq.integ.start.scam  ; param
              loadwave (reset)
              loop.back ? CASE                              -- waiting for DAQ ready signal (echo of integ.start)
              msg; cid; param
              SKIP
                clockwave (reset)
                loadwave (read)
                clock ? time.now                             -- start integration time
                SEQ i = 0 FOR multi                          -- read the "pedestal" level multi times
                  SEQ
                    clockwave (read)
                    wait.micros (multiread.pause)
                    ALT                                              -- wait for integration to end or abort
                      clock ? AFTER time.now PLUS itime.scam.ticks --wait for integration time to elapse
                        SEQ
                          SEQ i = 0 FOR multi                        -- read out array multi times
                            SEQ
                              clockwave (read)
                              wait.micros (multiread.pause)
                          OutputMode    := registerHL              -- select direct output register
                          DirectBoth[0] := read.enable             -- turn on the read.enable line
                          coadd.num := coadd.num + 1
                      root.to.clock ? CASE                         -- poll abort signal
                        msg; cid; param
                          SEQ
                            CASE cid
                              cid.abort.scam
                                SEQ
                                  SEQ i = 0 FOR multi              -- read out array multi times
                                    SEQ
                                      clockwave (read)
                                      wait.micros (multiread.pause)
                                  OutputMode    := registerHL      -- select direct output register
                                  DirectBoth[0] := read.enable     -- turn on the read.enable line
                                  not.abort := FALSE
                                  clock.to.root ! msg; cid.abort.scam ; Scam
                               ELSE
                                 SKIP
          IF
            not.abort
              SEQ
                to.daq ! msg; cid.daq.integ.end.scam; 0
                loop.back ? CASE         -- receives the echo of the cid from the last DAQ
                  msg; cid; param
                    SKIP
              TRUE
                SEQ
                  to.daq ! msg; cid.abort.scam ; 0
                  loop.back ? CASE         -- receives the echo of the cid from the last DAQ
                    msg; cid; param
                      SKIP
```