**George Brims**


**Revised February 18, 1999**

**NIRSPEC Software Programming Note 18.00**

## Clock generator programs

### 1 Introduction

This note describes the two programs which run on DAQ17 boards and clock out the two infrared detector arrays in NIRSPEC. The two arrays are completely different in terms of their clocking patterns, but otherwise the two programs are pretty much the same. As well as handling clock generation, both programs control the analog chains' gain and filter bandwidths and analog offsets, and also, in the spectrometer channel only, the detector bias.

During the acquisition of a frame, the clock generator programs interact in various ways with the acquisition code in the DAQ15 boards. This process is explained in NSPN33, and the acquisition code in NSPN19.

### 2 Hardware description

The DAQ17 board is more fully described in NEAN05, but I will give the programmer's view of it here. The board has one T805 transputer and 4 MBytes of memory. Via firmware, the transputer sees the outside world as a selection of registers where output data are loaded and input data read back.

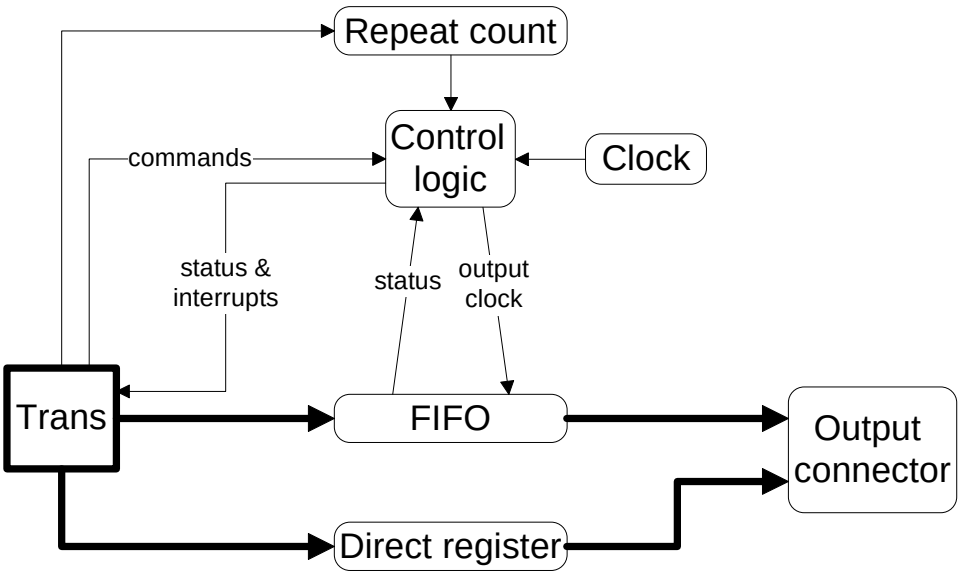



**Figure 1:** DAQ17 clock generation block diagram.

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 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. The contents of the FIFO buffer are then sent out the number of times specified in the repeat count register. 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 (to be read by the acquisition boards). Eight of the clock output lines are also copied to pins on the VME bus connector. We use one of these lines 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. The DACs set the analog offset voltages for the pre-amps in both cameras, and also control the detector bias for the Aladdin array of the spectrometer channel. We use 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). The gain/bandwidth settings and offset voltages are never changed during an integration.

## 3 Program structure

The overall structure of the clock generator programs is simple. They are one-process programs (no parallel operations) which sit in an event loop waiting for commands from the host, then perform the requested functions. They use three of their four serial links: one for talking to the host (via the root transputer), and the other two to talk to the acquisition transputers. In each camera section the clock generator and acquisition transputers are connected in a loop. During the clocking and acquisition of a frame, each acquisition transputer passes a message along in the same direction. The message will eventually come back to the clock generator to let it know that all the acquisition transputers are ready for the next step in the data-taking process.

The main loop of the program is headed by an ALT statement. Recall that an ALT is a construct that lets a transputer efficiently handle multiple events. There are four possible inputs to this ALT; three inputs on the serial links, and a clock input. Three of these cases are very simple. When the clock input triggers at set intervals, the code does a reset of the detector array, so that it doesn't build up any persistent charge during periods when the instrument is idle. Handling any input (message or data) from the acquisition transputers is simple: everything is simply passed on to the host via the root transputer.

The input to the ALT that causes the clock generator to do the most work is the link from the root transputer. Any message coming in from the root transputer will be a command from the host, so it is passed to a CASE statement which decides what to do with it. Most command identifiers (cid's) simply order a change in the value of some program variable such as the sampling mode or integration time. If the command identifier is a “go” message, the data acquisition process will start. A few other cid's command other code sequences such as generating test output. The response to each cid is listed in the next section.

## 4 Response to commands

This section describes what the program does in response to each possible cid value. Any cid value not listed has reached the clock generator transputer by mistake, and will be ignored. The CASE statement that handles the cid's lists them in random order, rather than alphabetical by name or numeric as listed in the main include file *nirspec.inc*, and I've kept to the same random order here so it's easier to follow through the code.

### 4.1 cid.test

This cid is used as a probe of the transputer network. It is routed through the network according to the value of its accompanying parameter value. If this value is 2 (SCAM) or 6 (spectrometer) the message is echoed back to the host with 1000 added to the parameter. Otherwise it is passed on to the acquisition transputers. See NSPN21 for an explanation of the use of this command.

### 4.2 cid.set.gain.spec / cid.set.gain.scam

This command orders a change in the analog chain's gain setting. Gain and bandwidth control bits are combined in a word called `aux.data`. The control bits for the gain setting are cleared, then the word is OR'ed with the bitmask for the requested gain setting. Finally `aux.data` is written to motor port 0. The command parameter can be 1, 2, 3, or 4, but these are *not* the actual values of the gain selected, just the number of the section of hardware that sets it.

### 4.3 cid.set.freq.spec / cid.set.freq.scam

This command orders a change in the analog chain's filter setting. Gain and bandwidth control bits are combined in a word called `aux.data`. The control bits for the bandwidth setting are cleared, then the word is OR'ed with the bitmask for the requested setting. Finally `aux.data` is written to motor port 0. The command parameter can be 1, 2, 3, or 4 to select different cutoff frequencies for the highpass filter stage of the analog chain.

### 4.4 cid.fifo.test.spec / cid.fifo.test.scam

This command exercises the clock output hardware and also puts out a pattern that can be used to test signal continuity in the hardware. It illustrates in a compressed fashion everything that goes on in routines `makewave`, `loadwave` and `clockwave` (see below in Section 5). It also illustrates how to address the memory-mapped hardware. To send a piece of data out to the hardware you just assign its value to a memory-mapped variable. For instance the FIFO buffer is addressed through variable `Fifo32bit`. This is actually declared as an array of 16484 32-bit words, so that we can send data to it using a DMA transfer. Obviously we limit the size of the declared array to the actual size of the FIFO buffer, and the physical addresses for the different variables are far enough apart that we don't overwrite another one by doing so. The names of many of the variables include things

like “HL” or “32bit”. These are used because the clock output hardware is actually split into two 16-bit subsystems, which can be used either separately or in sync. We always use them synchronized, as indicated by these variable names. The “H” and “L” indicate high and low words, so there are registers with “H”, “L” for the separate halves. The nomenclature “32bit” means using all 32 bits of the port.

First the FIFO buffer is cleared by writing a zero (actually any value will do) to register `nFifoRst`, then a loop inserts a sequence of ascending words (1, 2, 4, 8...) into the FIFO buffer through memory-mapped variable `Fifo32bit[0]`. This will have the effect of sending out pulses on each clock line in turn from LSB to MSB as the words are clocked out. The value of the command parameter is loaded into the repeat count register. The hardware limit on repetitions is 4096 so the value is trimmed if it's bigger. This emulates what happens in `makewave`, which composes waveforms, and `loadwave`, which puts it in the FIFO buffer and loads the repeat count.. The FIFO buffer is now ready to send out the loaded data, but first, just as in a real array waveform, it sends out some data via the direct register (recall that data written to either register ends up at the same hardware output port).

The code now sets output mode to the direct (non-repeating) register by writing value `registerHL` to control register `OutputMode`, then goes into a loop where it repeatedly writes a sequence of pulses to the direct register. Again the command parameter value is used as a repeat count. The bit pattern is all bits off, all bits on, then all bits off again. This is written to memory-mapped variable `DirectBoth[0]`. This output will appear at the connector as it is written by the code.

Next output mode is switched to the FIFO buffer by writing value `fifoHL` to register `OutputMode`. Writing the value `Common` to register `FifoControl` makes the upper and lower sections of the output hardware work together, and the repeat count is written into both the upper and lower halves of register `Repeat`. Output is triggered by writing the value `runHL` to the control register `Go`. This sets the hardware running to write out the contents of the FIFO buffer the requested number of times.

Now although the FIFO clock output is now entirely in the hands of the hardware, the code has to wait until it is done before doing anything else, so it goes into a loop where it waits for a series of interrupts, one for each time the hardware triggers the re-transmit of the FIFO. This will occur after each output is complete, except for the last time, so the loop count is one less than the repeat count. First any prior interrupts are cleared by writing a zero to the interrupt control register `int.enable`, then `RetransL` is written to the same register to specify that the retransmit is the interrupt we want. Writing `master.int.en` to the interrupt master control register sets up the interrupt. The code now waits until a byte is available through the dummy input channel `event`. This channel returns an input when the hardware toggles the hardware event line to the transputer. After triggering on the retransmit the appropriate number of times, the code sets up and triggers on the FIFO done interrupt (`DMAdoneHL`), which indicates the last output is complete. The code then writes the `StopHL` value to the `Go` register.

Finally the code repeats the loop where it sends bits to the output through the direct output register, and sends the command and its (possibly clipped) parameter back to the host.

The net effect of all this, if you look at any one output bit on a scope, is `3n` pulses, where n is the repeat count you requested with the command. The first `n` and last `n` will be the same equally spaced output for every bit, and the `n` in the middle will also be equally spaced but start earlier for the least significant bits and later for the most significant.

### 4.5 cid.test.frames.spec / cid.test.frames.scam

This command triggers another test mode for clocking. In many ways it is more useful than the previous one, since it puts out the actual array clocking waveforms, and can go on for a long time. The code executes a loop, sending out the read waveform once every half second. The repeat count is the parameter sent with the cid.

### 4.6 cid.go.spec / cid.go.scam

This is of course the most important command of all, triggering the code to take data. The process of taking data is described in much more detail in NSPN33. In that document the coordination between the clock generator and data acquisition transputers is described in depth. Here I will just describe what goes on in the clock programs, with brief mention here and there of what the acquisition transputers do.

When the clock generator gets the go 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 then does a CASE on the value of `sampmode`, and goes into the appropriate code section for the current sampling mode.

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.

#### 4.6.1 Single sampling

On entering the code section for single sampling, the clock generator enters a WHILE loop. It will execute this loop 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 arriving during the integration time.

#### 4.6.1.1 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, so it ends up back at the clock generator (input on channel `loop.back`).

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. The acquisition transputers receive the pixels and co-add them into their frame buffers, then wait for another command. Finally the clock generator increments its counter `coadd.num`.

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 the `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. The data from the frame it just clocked out aren't passed back to the host until it gets back to this main loop, and the acquisition transputers send their data through.

#### 4.6.1.2 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 expecting 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 done 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 goes back to waiting for a new go command, without sending any data back.

### 4.6.2 Double correlated sampling

On entering the code section for double correlated sampling, the clock generator enters a WHILE loop. It will execute this loop 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 arriving during the integration time.

#### 4.6.2.1 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, so it ends up back at the clock generator (input on channel `loop.back`).

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), then 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. 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 clock the array again. The final data for each co-add is the difference between the pixel values in the two frames. The acquisition transputers receive the pixels and co-add them into their frame buffers, then wait for another command. Finally the clock generator increments its counter `coadd.num`.

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 the `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. The data from the frame it just clocked out aren't passed back to the host until it gets back to this main loop, and the acquisition transputers send their data through.

#### 4.6.2.2 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 clocking out the first and second sets of 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 expecting 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 done 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 goes back to waiting for a new go command, without sending any data back.

### 4.6.3 Multiple correlated sampling

On entering the code section for multiple correlated sampling, the clock generator enters a WHILE loop. It will execute this loop 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 arriving during the integration time.

#### 4.6.3.1 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, so it ends up back at the clock generator (input on channel `loop.back`).

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 now goes into a loop where it calls routine `clockwave` to generate the readout clock pattern multiple times (control variable is `multi`). 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 ready for this number of pixels because they too know the value of `multi`. 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 loops around calling routine `clockwave` to clock the array `multi` times. The final data for each co-add is the difference between the totals of the pixel values in the two sets. The acquisition transputers receive the pixels and co-add them into their frame buffers, then wait for another command. Finally the clock generator increments its counter `coadd.num`.

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 the `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. The data from the frame it just clocked out aren't passed back to the host until it gets back to this main loop, and the acquisition transputers send their data through.

#### 4.6.3.2 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 clocking out the first and second sets of 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 expecting data, so first it executes a loop where 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 done 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 goes back to waiting for a new go command, without sending any data back.

### 4.7 cid.itime.spec / cid.itime.scam

This command receives a new value of integration time from the host. The parameter gives the time in milliseconds. This is converted to transputer clock ticks (microseconds).

### 4.8 cid.sampmode.spec / cid.sampmode.scam

This command gets a new value for the sampling mode. The possible values are `single.samp`, `correlated.double.samp` and `mult.corr.sample`. The message is passed on to the acquisition transputers, which also need to know the sampling mode.

### 4.9 cid.samprate.spec / cid.samprate.scam

This command sets the sampling rate for pixel clocking. It is specified in kHz. The maximum rate the A-D converters can handle is 500 kHz. As soon as this new value is received, the program calls routine `makewave` to compose new clock waveforms at this rate.

### 4.10 cid.coadds.spec / cid.coadds.scam

This command receives a new value for the number of coadds to be used in taking frames.

### 4.11 cid.multi.spec / cid.multi.scam

This command gets a new value for the number of samples to be taken in multiple correlated sampling.

### 4.12 cid.quad1.offset.spec / cid.quad1.offset.scam

This command receives a value between 0 and 4095, which is sent using routine d2a.write to the serial DAC in the interface board. If the value is out of range it is set to either 0 or 4095, depending which end of the range it crossed. It sets the analog offset voltage for quadrant 1 of the detector. The actual voltage produced is circuit dependent.

### 4.13 cid.quad2.offset.spec / cid.quad2.offset.scam

This command receives a value between 0 and 4095, which is sent using routine d2a.write to the serial DAC in the interface board. If the value is out of range it is set to either 0 or 4095, depending which end of the range it crossed. It sets the analog offset voltage for quadrant 2 of the detector. The actual voltage produced is circuit dependent.

### 4.14 cid.quad3.offset.spec / cid.quad3.offset.scam

This command receives a value between 0 and 4095, which is sent using routine d2a.write to the serial DAC in the interface board. If the value is out of range it is set to either 0 or 4095, depending which end of the range it crossed. It sets the analog offset voltage for quadrant 3 of the detector. The actual voltage produced is circuit dependent.

### 4.15 cid.quad4.offset.spec / cid.quad4.offset.scam

This command receives a value between 0 and 4095, which is sent using routine d2a.write to the serial DAC in the interface board. If the value is out of range it is set to either 0 or 4095, depending which end of the range it crossed. It sets the analog offset voltage for quadrant 4 of the detector. The actual voltage produced is circuit dependent.

### 4.16 cid.detbias.spec

This command gets a new value for the spectrometer channel detector bias, and writes it using routine `detbias.write` to a serial DAC in the interface board. As with the quadrant offsets, the parameter value is between 0 and 4095. The SCAM detector bias is set in hardware.

## 5 Description of routines

The clock generator programs use a number of routines, which actually make up the bulk of the code in terms of sheer line count.

### 5.1 makewave

Called as: `PROC makewave(VAL INT wavetype, VAL INT rate, INT error)`

This routine is used to compose in advance the waveforms that are sent out by routine `clockwave`. Here we will describe the way the routine works, with the details of the actual waveforms given below in Section 6.

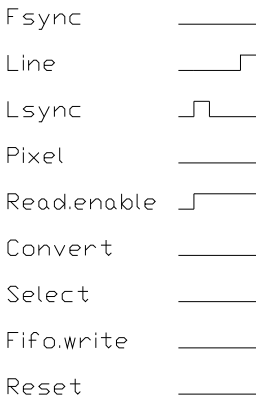
There are two different types of waveform, reset and read, and each is made up of three separate sequences. Arrays for storing these waveform sections are declared at the top of the main program. The sizes of the arrays are defined in the include files **scam_clockdefs.inc** or **spec_clockdefs.inc**.

Each time `makewave` is called, it composes the initial sequence sent via the direct output register, the sequence to be sent out via the repeating FIFO buffer, and the end of frame sequence that also goes out via the direct register. In each case the data are composed and stored in temporary array `waveform.temp` and copied to the storage array when complete.

Composing a waveform is very simple. There is a 32-bit integer called `currentword` which holds the current state of all the output bits as we go through the sequence. Bits in `currentword` are set or cleared using bitmasks named for each of the output lines, as defined in the include files. For instance for the spectrometer channel (Aladdin array) the bitmask for `PhiSyncSlow` is 1, for `PhiSlow1` it's 2 and so on. If we do a bitwise OR of `currentword` with `PhiSyncSlow`, the `PhiSyncSlow` bit will be turned on (the operation AND NOT will turn the same line off). We can turn on or off as many lines as we want at any point in the waveform. To have this state of the output lines last a certain number of output clock ticks, we copy `currentword` into the `waveform.temp` array that number of times, keeping track of position within the array using pointer `pulse`. This is all simplified by a number of short routines, `on`, `off`, `tick`, `select.top` and `select.bot`, defined at the start of `makewave`. These are described below.

In a way these routines define a simple “language” for composing waveforms. Looking at the sample section of code below along with the corresponding segment of the waveform, we can see how this “language” operates.

```
tick (1)
on (lsync)
on (read.enable)
tick(1)
off (lsync)
tick(2)
on (line)
tick (1)
```




We start with all lines off, and do one time tick. We then turn on lines `lsync` and `read.enable`, and do one more tick before turning `lsync` off again. We then do two ticks and turn on `line`, then one more tick. If we wanted the lsync pulse to be one time slice longer, we would have typed

```
on (lsync)
on (read.enable)
tick(2)
off (lsync)
tick (1)
```

Before going into the sequence of composing all the waveforms, there is a question of overall timing. The minimum time slice available from the hardware is 200ns. For any sampling rate of the A-Ds, there is a number of time slices that will comprise one pixel time (for example 10 or 20 slices for 500kHz or 250kHz). Once the waveform has selected each pixel, there is a fixed sequence we have to perform, toggling the A-D convert signal and latching the data for the previous pixel into the acquisition transputers. This sequence is only part of the total pixel time even if we're going at the fastest rate possible (500kHz). The rest of the pixel time is defined by variable `npause`. To calculate `npause`, we first figure out the number of ticks per pixel from the sample rate, then subtract the length of the fixed part. Stretching `npause` makes the waveform longer and therefore slower. There is a minimum value of `npause` which is set by the shortest possible pixel time of 10 ticks, and a maximum which is different for each type of waveform, but is set by the capacity of the FIFO buffer. The whole repeating part of the waveform has to fit inside the 16384 deep buffer. The slow speed limit is in the 80kHz range for both detectors, and is set as a limit on the value of the sample rate keyword in the host software.

So, for each segment of the waveform we go through the sequence of `on`, `off` etc. as in the examples above, then finally copy the temporary array to the array for that sequence. The length of the data placed in the array is copied to array `wave.length`. Generally the start and end sequences are pretty short, but the repeating section can be pretty long. For the repeating section we also store the number of times it should be clock out in array `wave.reps`. Within the repeated section there is still more repetition, which we implement by indexed loops just like any other repetitive task. The

advantage of our hardware-based clocking scheme is that there is no timing penalty in using loops as there would be if we were to clock directly from the transputer.

**5.2 on**

Called as: `PROC on (VAL INT line)`

This routine turns on the bit specified by bitmask `line` in `currentword`.

**5.3 off**

Called as: `PROC off (VAL INT line)`

This routine turns off the bit specified by bitmask `line` in `currentword`.

**5.4 tick**

Called as: `PROC tick (VAL INT nticks)`

This routine puts `nticks` copies of `currentword` into the temporary array `waveform.temp`, The pointer `pulse` is incremented by `nticks`.

**5.5 select.top & select.bot**

Called as: `select.top()` & `select.bot()`

In order to make the waveform slightly more readable, we added these two routines as a special case of `on` and `off`. Each analog board has two A-D converters, feeding their data out over just one connector. The `select` line controls which of the two A-D converters is seen at the output connector, the top one or the bottom one. Rather than have the waveform-generating code read

```
select (on)
tick (1)
select (off)
```

it reads

```
select.top()
tick (1)
select.bot()
```

The readability of the code is improved slightly, in that one doesn't have to continually remember whether it's on or off which means the top or bottom A-D is reading out.

### 5.6 loadwave

Called as: `PROC loadwave (VAL INT wavetype)`

This routine loads the stored data for the type of waveform defined by `wavetype` to the FIFO output buffer in preparation for clocking out by `clockwave`. It stops FIFO output and clears the FIFO first, then copies the data to the buffer. The nomenclature

```
[Fifo32bit FROM 0 FOR npulses] := [waveform.rep[wavetype] FROM 0 FOR npulses]
```

requests a DMA transfer of data from one location to another, so this operation is as swift as the processor can manage.

### 5.7 clockwave

Called as: `PROC clockwave(VAL INT wavetype)`

This routine does the actual sending out of the data comprising the three parts of the waveform. To send a piece of data out to the hardware the code assigns its value to a memory-mapped variable. For instance the FIFO buffer is addressed through variable `Fifo32bit`. For each waveform type the data are picked up from the storage array by their index (read or reset).

The code first sets output mode to the direct (non-repeating) register by writing value `registerHL` to control register `OutputMode`, then copies the first sequence of pulses from `waveform.start` to the direct register.

Next output is switched to the FIFO buffer is cleared by writing `fifoHL` to register `OutputMode`. Writing the value `Common` to register `FifoControl` makes the upper and lower sections of the output hardware work together. The stored repeat count for this waveform is copied from `wave.reps` into both the upper and lower halves of register `Repeat`. Output is triggered by writing the value `runHL` to the control register `Go`. This sets the hardware running to write out the contents of the FIFO buffer the requested number of times.

Now although the FIFO clock output is now entirely in the hands of the hardware, the code has to wait until it is done before doing anything else, so it goes into a loop where it waits for a series of interrupts, one for each time the hardware triggers the re-transmit of the FIFO. This will occur after each output is complete, except for the last time, so the loop count is one less than the repeat count. First any prior interrupts are cleared by writing a zero to the interrupt control register `int.enable`, then `RetransL` is written to the same register to specify that the retransmit is the interrupt we want. Writing `master.int.en` to the interrupt master control register sets up the interrupt. The code now waits until a byte is available through the dummy input channel `event`. This channel returns an input when the hardware toggles the hardware event line to the transputer. After triggering on the retransmit the appropriate number of times, the code sets up and triggers on the FIFO done interrupt

(DMAdoneHL), which indicates the last output is complete. The code then writes the StopHL value to the Go register.

Finally the code switches back to writing through the direct output register, and sends the end sequence from waveform.end.

**5.8 undeselect**

Called as: PROC undeselect ()

This routine is used in the spectrometer channel only. The Aladdin 2 array has a facility that allows you to turn off ("deselect") one or more row pairs which have defective pixels, particularly if they are hot pixels which impact the background on the rest of the array by emitting photons. When the array powers up, it has *all* rows deselected, so the array doesn't work at all. This routine turns all the rows on.

First it writes to the direct register five times with only the PhiSyncSlow line on, then five times with PhiSyncSlow and all four PhiDeselectn lines on, and finally five times with only PhiSyncSlow again. This routine is called as soon as the clock generator code is run.

**5.9 global.reset**

Called as: PROC global.reset ()

This routine is used in the spectrometer channel only. The regular reset is a line by line reset. This routine resets the whole array at once. To do so it just writes (35 times) a clock word with only the VrstGlobal line turned on.

**5.10 wait.micros**

Called as: PROC wait.micros (VAL INT micros.delay)

This simple array is used to pause a requested number of microseconds, using the on-chip clock. It calculates the number of clock ticks needed for the requested delay, takes a time stamp, and waits for the timer to hit the needed delay after the timestamp. There is a 10-20 microsecond calling overhead so this routine isn't useful for small delays.

### 5.11 wait.ms

Called as: `PROC wait.ms (VAL INT ms.delay)`

This simple array is used to pause a requested number of milliseconds, using the on-chip clock. It calculates the number of clock ticks needed for the requested delay, takes a time stamp, and waits for the timer to hit the needed delay after the timestamp.

### 5.12 initauxport

Called as: `PROC initauxport (INT aux.data)` (SCAM)
`PROC initauxport (INT aux.data0, INT aux.data1)` (Spectrometer)

This routine initializes the data sent out over the digital I/O ports to perform auxiliary functions. These functions are control of gain and bandwidth, and setting of quadrant analog offset and (spectrometer channel only) detector bias. Because we use the same hardware port to send a static bit pattern for gain/bandwidth control (2 bits each) and a time sequence to the serial DAC that generates the quadrant offsets, we keep a record of the current state of the bit pattern in integer `aux.data` (SCAM) or `aux.data0` (spectrometer). Integer `aux.data1` is used to keep track of the data for the other port, which is used to write to the serial DAC for the spectrometer channel detector bias. Each routine that affects the value of these integers changes only the bits it's concerned with and not the others (see following sections).

There is also a hardware feature to deal with. The signals from the transputer hardware go into the analog hardware through Burr-Brown ISO150 capacitive isolator chips to cut down on noise injection from the digital hardware into the analog signals. These chips power up with their output state indeterminate (actually it's not so much indeterminate as circuit-dependent) rather than copying the input. Once they have seen a transition from high to low and back again they operate properly. The first section of this routine writes all bits of each output port low then high then low again to take care of this requirement.

The routine then gets the default values for the output words and copies them to the output ports so the hardware has the default gain and bandwidth settings.

### 5.13 d2a.write

Called as: `PROC d2a.write(VAL INT chan, VAL INT offset, INT aux.data0)`

This routine writes the serial data for the quadrant offset to the Maxim MAX536 DAC on the bias board. The serial DAC has four channels, one per quadrant, so `chan` is 0, 1, 2, or 3 for quadrants 1 to 4. The input data `offset` is in the range 0 to 4095. The exact voltage this gives is circuit dependent. The host software controls the offset by sending this data value. Integer `aux.data`

(SCAM) or `aux.data0` (spectrometer channel) is the holder for the bit pattern sent out over digital I/O port 0, so the gain/bandwidth bits are unaltered by the operation of this routine.

The data word to be sent out comprises two bits to select the channel, two control bits (always 0, 1) and the 12-bit data. We use three bits of the output to send this data word to the chip.

The chip select bit is normally held high all the time. First it is turned off in aux.data and aux.data is written out. It stays low during the whole write period.

Now there is a short pause, then the data word to be sent is composed from the offset value, the control bits, and the channel number.

The program now goes into a loop where it sends out the lower 16 bits of the word `data`. For each bit, `data` is right-shifted and AND'ed with `datamask` to see whether that bit is set. Depending whether it is or not, the `sdata` bit is turned on or off in `aux.data` ,and the word is written out. The word is then written twice more, with the `sck` bit turned first on then off, with 1 millisecond of delay between each write. The effect of this is to pulse the sck bit so as to latch the value of the data bit (`sdata`) into the DAC.

Once all 16 bits of the data have been latched into the DAC, the `cs` bit is returned to its default high state, ending the write operation.

**5.14 detbias.write**

Called as: `PROC detbias.write(VAL INT detbias, INT aux.data1)`

This routine is only used on the spectrometer channel to control the detector bias. On the other channel the detector bias is set in hardware. It is identical to routine d2a.write, except that it only writes to one channel of the DAC, and the output port is `MotorWr1` not `MotorWr0`.

## 6 The waveforms

The two following subsections describe the composition of the read and reset waveforms for the two detectors in detail.

### 6.1 Spectrometer detector (Aladdin 2)

#### 6.1.1 Detector description and clocking details

The Aladdin 2 detector is laid out as four quadrants, each 512 by 512 pixels. Each quadrant has eight output lines, with every eighth pixel across a row going to the same output. There are separate lines into the chip to clock the four quadrants independently, but in NIRSPEC the detector is connected so that all four quadrants are clocked in sync. We compose and send out a clock pattern for one $512^2$ quadrant, which actually clocks out four quadrants simultaneously. Since we have wired the detector this way, and each quadrant is clocked from the outside corner to the center of the array, we can clock out either the whole detector, or a subarray that has mirror symmetry about the lines dividing the quadrants.

##### 6.1.1.1 Selecting rows and pixels

Clocking through the rows and pixels of the array is performed by feeding pulses to two shift registers. The slow shift register selects rows, and the fast shift register selects pixels. The currently selected pixels in the currently selected rows are seen at the analog outputs. At any point in the clock pattern, we have selected the same row in each quadrant, and the same 8 pixels in each row. This is illustrated in Figure 3 below. The figure shows the first set of pixels in the first row, the last set in the last row, and an arbitrary set in an arbitrary row. (The figure actually shows a 128 by 128 "Aladdin" array so we can see the pixels, and the selected pixels are the third set of the $16^{th}$ row.) The slow shift register clocks through rows from the top and bottom edges of the array towards the middle, and the fast shift register selects sets of 8 pixels starting at the right and left edges and working in towards the middle.

The slow shift register is driven by four clock lines. There is a sync pulse (`PhiSyncSlow`) which is used to initialize the shift register, and two clocks (`PhiSlow1` and `PhiSlow2`), which are asserted in turn to shift to consecutive *pairs* of rows. The fourth signal is `PhiEvenOdd`, which selects between the odd-numbered or even-numbered row of the selected pair. To set up the register (once per frame), we pulse `PhiSyncSlow` followed by `PhiSlow1`. To select the first row pair, we then set `PhiSlow2` high. To get the next pair we make `PhiSlow2` low, pause, then assert `PhiSlow1`. We then assert these two clocks in turn until we have selected all the row pairs one after the other. During the time each of `PhiSlow1` or `PhiSlow2` is high, we clock through the fast register (described below), first with `PhiEvenOdd` low to get the odd-numbered row of the pair, then with `PhiEvenOdd` high to get the even-numbered row.

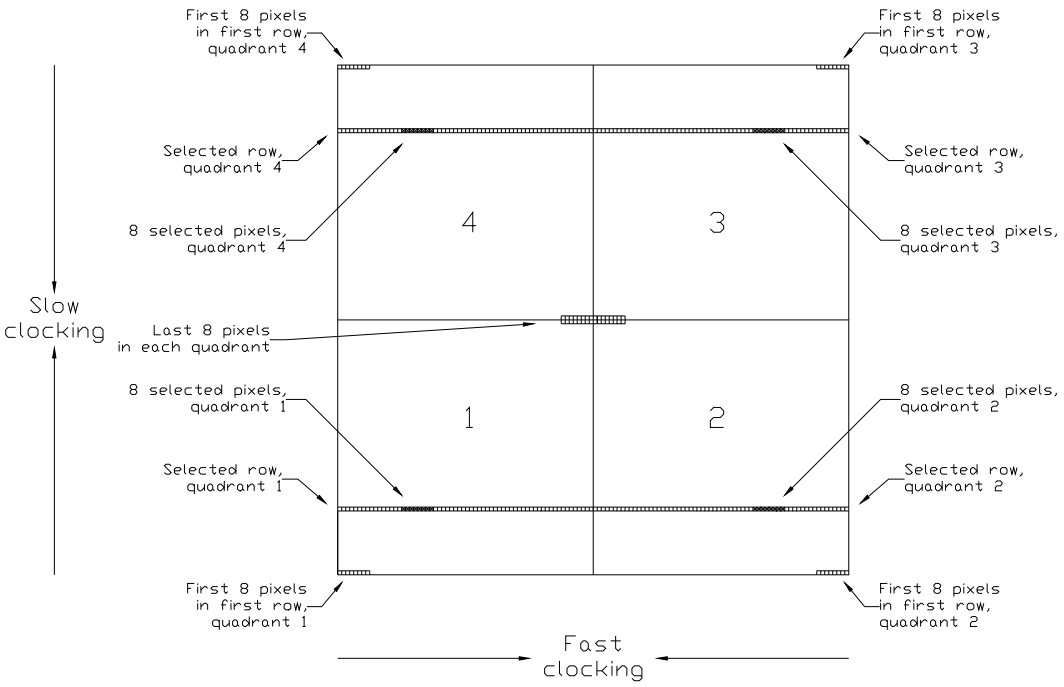



**Figure 3:** Selecting rows and pixels

The fast shift register is similar to the slow one, but needs only three signals. There is a sync pulse (`PhiSyncFast`) which is used to initialize it, and two clocks (`PhiFast1` and `PhiFast2`), which are pulsed in turn to shift to the next set of eight pixels. To set up the register for each row, we pulse `PhiSyncFast`, followed by `PhiFast1`. We then assert `PhiFast2` to select pixels 1 - 8 of the current row in each quadrant. We then make `PhiFast2` low, pause, and assert `PhiFast1` to select pixels 9 - 16. We then alternate `PhiFast1` and `PhiFast2` to clock right through the row. Each selected pixel goes to its own analog output line, so the first output line will read out pixel 1, 9, 17 and so on, line 2 will read out 2, 10, 18... and so on.

There are 4 possible states of the slow shift register. These are `PhiSlow1` on - `PhiEvenOdd` off, `PhiSlow1` on - `PhiEvenOdd` on, `PhiSlow2` on - `PhiEvenOdd` off, and `PhiSlow2` on - `PhiEvenOdd` on. We select each of these states in turn, and for each we clock through the fast shift register (previous paragraph) once. Once we have gone through rows 1 to 4 we repeat the pattern for 5 to 8, then 9 to 12 and so through the quadrant to row 512. This means that the segment of code we need to compose to put in the FIFO output buffer is this four row sequence. The hardware can clock the whole array by repeating this four-row sequence 128 times to go through all 512 rows in each quadrant.

#### 6.1.1.2 Other clock signals

In addition to the shift register clock signals, the array takes two other signals, `VResetGlobal` and `VresetRow`. `VresetGlobal` is used to trigger a global reset of the array, i.e. all the pixels at once. `VresetRow` resets all pixels in the currently selected row pair.

#### 6.1.1.3 Data control signals

There are also three signals, generated as part of the clock pattern, that go to other parts of the electronics to control data-taking. These are `convert`, `select`, and `fifo.write`.

These three signals are used as follows. Once the fast register has been clocked to the next set of pixels, we wait a little while for the new analog signal to settle, then toggle `convert`. This tells the A-D converter to hold the current value of the analog signal and digitize it. Since the A-D converters are capable of up to 500kHz operation, it will take just under 2 microseconds for the new data values to be available. Meanwhile the digital data for the *previous* set of pixels is still waiting in the output latches of the converters, so we have to latch those numbers into the acquisition boards. Recall that there are two A-D converters per analog board. The state of the `select` line determines which one is seen at the output connector, and the `fifo.write` line tells DAQ15 boards to trigger their FIFO buffers to take in the data. We pulse `fifo.write` twice, once with `select` low and once with `select` high, so we get both the data values from each analog board.

### 6.1.2 Read frame waveform

When reading the following description, it is essential to have a copy of the code (routine `makewave` in file `specclock.occ`) and a printout of the clocking pattern available. The clock pattern is shown in Figure 4, but a larger printout from AutoCAD drawing #613143 is easier on the eyesight. The vertical dotted lines are one time tick apart. Each waveform is divided up into several segments, denoted by the slightly heavier dotted lines. Along the top of the diagram each of these heavier lines has a letter, used to reference the segments in the subsections below.

Along the bottom of the diagram are horizontal lines spanning different sections. The upper set show how many times each segment from b—c to n—o is written to the FIFO buffer. The lower set shows how many times each segment is written to the output: once each for a—b and o—p, and 128 times for the sections written to the FIFO buffer.

#### 6.1.2.1 Segment a — b

This section of the waveform is generated by writing to the direct register, so the ticks are not actually to the same time scale as the rest of the waveform, which goes out through the repeating FIFO buffer. They are about twice as long but the timing is not critical. The only thing we do during this segment is to initialize the slow shift register by first pulsing `PhiSyncSlow`, then `PhiSlow1`.

#### 6.1.2.2 Segment b — c

This segment is written once to the FIFO buffer. Here we turn on `PhiSlow2`, which selects the first row pair. Since we are leaving `PhiEvenOdd` low, we are now looking at the first row (odd-numbered) of the pair. We then initialize the fast shift register by pulsing `PhiSyncFast` followed by `PhiFast1`. There is a pause of several ticks after we do all this so that the shift register has some time to settle down.

#### 6.1.2.3 Segment c — d

This segment is written once to the FIFO buffer. It takes us through the first two sets of pixels in the row (recall that each new "pixel" selected by the fast shift register is actually 32 pixels, 8 in each quadrant). First we turn on `PhiFast2` to select the first pixel set in the row, then wait `npause` ticks before pulsing `convert`. This starts the A-D converters digitizing the first set of values (1 tick duration is enough since the convert is edge-triggered). That is all we do during the first pixel time. `PhiFast2` is on for (`npause` + 7) ticks, then we turn it off.

One tick later we turn on `PhiFast1`, selecting the second pixel set. Again we wait `npause` ticks before pulsing `convert`. This time we also have data to read into the acquisition transputers - the data from the first set of pixels. This data from the A-D converters becomes valid 1.3`:`s after the first pixel set's `convert` pulse, and stays valid until 1.1`:`s after this pixel set's `convert` pulse.

To read the data into the acquisition transputers we pulse `fifo.write` twice. The rising edge of each pulse triggers the DAQ15 input FIFO buffers to read in data. After the first pulse we turn on the select line, by calling routine `select.top`. This tells the analog board hardware to switch the data from the upper A-D converter on each board to the output connector. After we have pulsed `fifo.write` the second time we call `select.bot` to return the select line to zero. This sequence of two `fifo.write` pulses and one `select` pulse occurs in every pixel set except the first. Of each row. Finally we turn `PhiFast1` off.

#### 6.1.2.4 Segment d — e

This segment is written thirty-one times to the FIFO buffer. Following the first two sets of pixels in each row (previous subsection), where we trigger `convert` twice but only read data back during the second set, we go through the rest of the row with this sequence, triggering the convert and reading data during every pixel period.

First we turn on `PhiFast2` to select an odd-numbered pixel set, then wait `npause` ticks before pulsing `convert`. This starts the A-D converters digitizing the analog signals. Then we read data into the acquisition transputers - the data from the previous set of pixels. Again as described in the previous section, we pulse `fifo.write` twice, once with `select` low and once with `select` high. `PhiFast2` is on for (`npause` + 7) ticks, then we turn it off.

One tick later we turn on `PhiFast1`, selecting an even-numbered pixel set. Again we wait `npause` ticks before pulsing `convert`, then read the data from the previous set of pixels by pulsing `fifo.write` twice, once with `select` low and once with `select` high. `PhiFast1` is on for (`npause` + 7) ticks, then we turn it off.

#### 6.1.2.5 Segment e — f

In this section, written once to the FIFO buffer, we finish off the first row of this row pair. We clocked out section c — d once and d — e thirty-one times, making 32 × 2 = 64 sets of 8 pixels, giving us 512 pixels, which is on row in each of the four quadrants. We have selected each set of pixels along the row and read in the data from all but the last set. First we wait `npause` ticks just as we do in each pixel period, then pulse `fifo.write` twice, once with `select` low and once with `select` high, to read in that last set of data.

Now we must switch to the even-numbered row of the current pair, so we make `PhiEvenOdd` high (it will stay high for the duration of the row). We then initialize the fast shift register for the next row by pulsing `PhiSyncFast` followed by `PhiFast1`. There is a pause of several ticks after we do all this so that the shift register has some time to settle down.

#### 6.1.2.6 Segment f — g

This segment is exactly the same as segment c — d, except that `PhiEvenOdd` is high, and is written once to the FIFO buffer. It takes us through the first two sets of pixels in the row (recall that each new "pixel" selected by the fast shift register is actually 32 pixels, 8 in each quadrant). First we turn on `PhiFast2` to select the first pixel set in the row, then wait `npause` ticks before pulsing `convert`. This starts the A-D converters digitizing the first set of values (1 tick duration is enough since the convert is edge-triggered). That is all we do during the first pixel time. `PhiFast2` is on for (`npause` + 7) ticks, then we turn it off.

One tick later we turn on `PhiFast1`, selecting the second pixel set. Again we wait `npause` ticks before pulsing `convert`. This time we also have data to read into the acquisition transputers - the data from the first set of pixels. This data from the A-D converters becomes valid 1.3`:`s after the first pixel set's `convert` pulse, and stays valid until 1.1`:`s after this pixel set's `convert` pulse.

To read the data into the acquisition transputers we pulse `fifo.write` twice. The rising edge of each pulse triggers the DAQ15 input FIFO buffers to read in data. After the first pulse we turn on the select line, by calling routine `select.top`. This tells the analog board hardware to switch the data from the upper A-D converter on each board to the output connector. After we have pulsed `fifo.write` the second time we call `select.bot` to return the select line to zero. This sequence of two `fifo.write` pulses and one `select` pulse occurs in every pixel set except the first. Of each row. Finally we turn `PhiFast1` off.

#### 6.1.2.7 Segment g — h

This segment is exactly the same as segment d — e, except that `PhiEvenOdd` is high, and is written thirty-one times to the FIFO buffer. Following the first two sets of pixels in each row (previous subsection), where we trigger `convert` twice but only read data back during the second set, we go through the rest of the row with this sequence, triggering the convert and reading data during every pixel period.

First we turn on `PhiFast2` to select an odd-numbered pixel set, then wait `npause` ticks before pulsing `convert`. This starts the A-D converters digitizing the analog signals. Then we read data into the acquisition transputers - the data from the previous set of pixels. Again as described in the previous section, we pulse `fifo.write` twice, once with `select` low and once with `select` high. `PhiFast2` is on for (`npause` + 7) ticks, then we turn it off.

One tick later we turn on `PhiFast1`, selecting an even-numbered pixel set. Again we wait `npause` ticks before pulsing `convert`, then read the data from the previous set of pixels by pulsing `fifo.write` twice, once with `select` low and once with `select` high. `PhiFast1` is on for (`npause` + 7) ticks, then we turn it off.

#### 6.1.2.8 Segment h — i

This segment (shown twice in Figure 4) starts out like segment e — f, by reading the data for the last pixel set in the row, pulsing `fifo.write` twice, once with `select` low and once with `select` high.

Next we make `PhiEvenOdd` low, because we have finished the even-numbered row of the first row pair, and `PhiSlow2` low because we have now done both rows of the first row pair.

Next we make `PhiSlow1` high, which selects the next row pair, then pulse `PhiSyncFast` followed by `PhiFast1` to initialize the fast shift register.

#### 6.1.2.9 Segment i — j

This segment is the same as segment c — d, except that `PhiSlow1` is high instead of `PhiSlow2`.

#### 6.1.2.10 Segment j — k

This segment is the same as segment d — e, except that `PhiSlow1` is high instead of `PhiSlow2`.

#### 6.1.2.11 Segment k — l

This segment is the same as segment e — f, except that `PhiSlow1` is high instead of `PhiSlow2`.

#### 6.1.2.12 Segment l — m

This segment is the same as segment f — g, except that `PhiSlow1` is high instead of `PhiSlow2`.

#### 6.1.2.13 Segment m — n

This segment is the same as segment g — h, except that `PhiSlow1` is high instead of `PhiSlow2`.

#### 6.1.2.14 Segment n — o

This segment is the last one written to the FIFO output buffer. We have selected each set of pixels along the row and read in the data from all but the last set. First we wait `npause` ticks just as we do in each pixel period, then pulse `fifo.write` twice, once with `select` low and once with `select` high, to read in that last set of data.

Next we make `PhiEvenOdd` low, because we have finished the even-numbered row of the second row pair, and `PhiSlow1` low because we have now done both rows of the second row pair.

#### 6.1.2.15 Segment o — p

This segment (written via the direct register) is just a placeholder. It actually has nothing happening, since the array doesn't require any end sequence in its clock pattern. We just clock out a couple of ticks with all bits zero. This segment has occasionally been used in testing to put out a pulse on an unused line so we can trigger an oscilloscope to look at the tail end of the waveform.

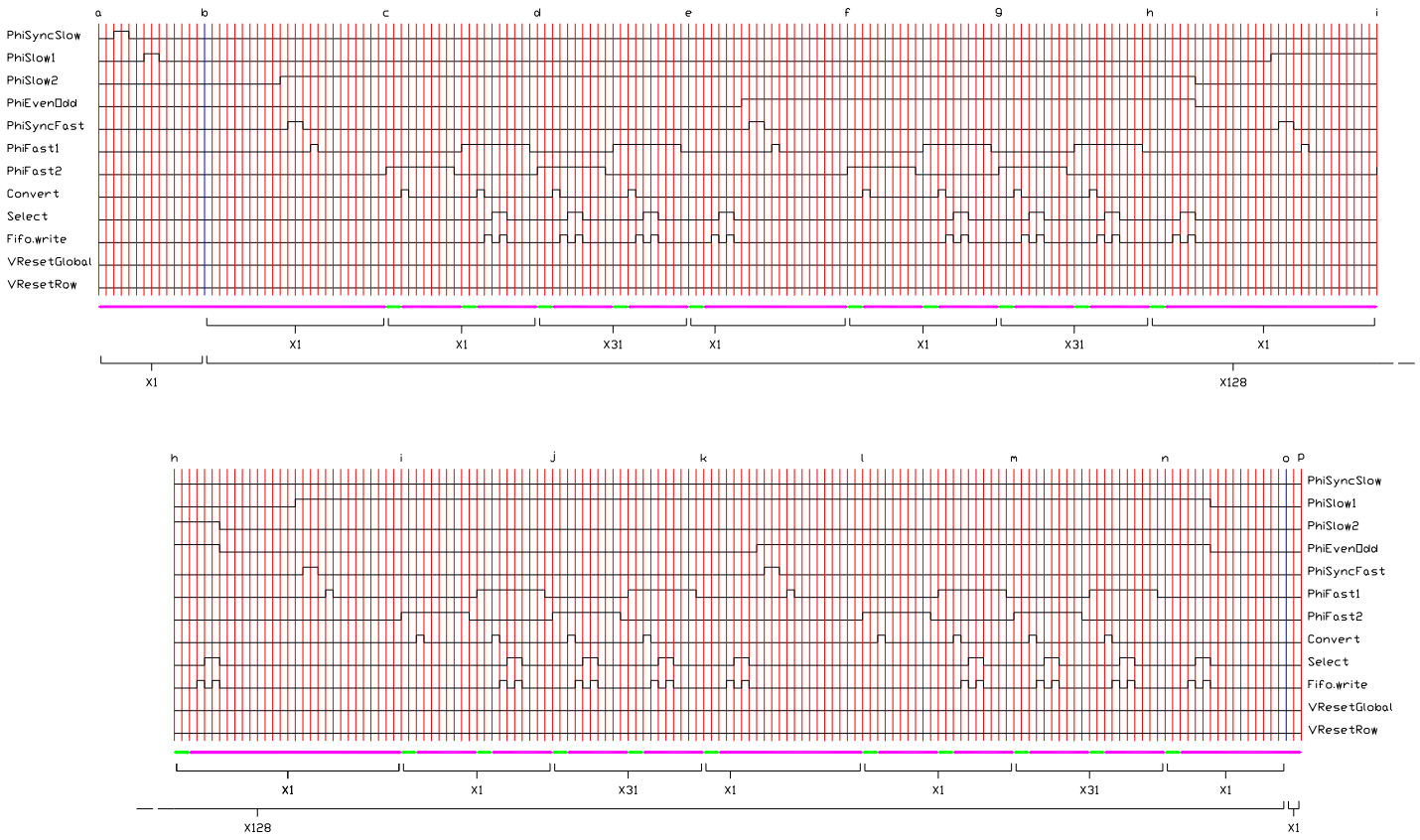



**Figure 4:** Aladdin 2 read waveform