

# Classic Front End Views

*Data correlation*

Robert Goodwin

Fri, Apr 17, 2009

## *15 Hz synchronization*

Linac-style front ends are designed to operate at 15 Hz synchronously with the Linac accelerator. To achieve this, an interrupt that is derived from the accelerator timing system triggers a series of tasks that run to completion in a small part of each 15 Hz cycle, so that all data is read into a data pool early on each cycle. Only after that step is complete, replies to active data requests are delivered. In this way, all data obtained from the data pool that is returned for an active request is correlated; each data value represents Linac data measured on that cycle. It is not possible for a partially completed data pool to be “seen” by an outside data requester.

The timing signal that starts the series of tasks is set to a time within the cycle when all accelerator signals are ready to be sampled. For the Linac, which accelerates a 50  $\mu$ s wide beam pulse every cycle, the front ends are triggered 1 ms later than beam, allowing for all sample-and-holds to fire and for the 500  $\mu$ s wide RF pulse to end, so that all is quiet in the gallery. Linac beam occurs at 2 ms after a Booster reset clock event, so the timing trigger is set to delay 3 ms after Booster reset event time. All Linac front ends perform their data acquisition simultaneously. When finished, they update all active data requests for which replies are due on that cycle.

To put numbers on this, a Linac PowerPC front end typically takes about 12 ms to collect all of its data and emit replies to active data requests. Most of this time is needed to await data from one or more SRMs via arcnet. The next step is to perform alarm scanning of all channels so enabled, which takes much less than 1 ms. The last job is used for operating the “little console” page application. All of these tasks normally complete quite early in the cycle, so that the rest of the 66 ms cycle is nearly all idle time.

## *Correlated data via server node*

As an extra measure to ensure that correlated data is delivered in response to Acnet data requests, nearly all Linac data is routed through node0600, usually referred to as the “Linac server node.” When the server node receives a RETDAT request, say, it often forwards the same request to a multicast destination that reaches all Linac front ends. This lets each front end see the request and determine whether it has a part to play in the request. (The second word of the SSDN is always the native node number that identifies the real source of the data.) If it does not, it ignores the request; but if it does, it initializes the request so that it returns (only) its part of the data requested according to the specified conditions, whether periodic, event-based, etc. As the server node receives these partial replies, it places them appropriately into its complete reply buffer. At a later (deadline) time in the cycle, usually 40 ms after the start of the cycle, the server task delivers the reply message to the original requester. To make this all work, the partial replies must be received by the server node before the server task runs. Using this server scheme, reply data sent to Acnet requesters is correlated; *i.e.*, it was all measured on the same cycle. All front ends thus operate in synchronism with the 15 Hz Linac accelerator.

## *GETS32 support*

This protocol is designed to help solve the problem of correlating data across multiple front ends, so that a requester can know that such data was measured at the same time. In Linac, this mainly implies the data was sampled on the same 15 Hz cycle. To this end, three 8-byte time stamps are included in a GETS32 protocol reply header. The cycle time stamp is allowed to be a global cycle number, defined as a global counter of clock event 0x0F occurrences. The collection time stamp is expressed as a number of milliseconds since Jan 1, 1970. For the case of a request based on clock event plus delay, this time stamp is set to the time of the event plus the delay time.

(The third time stamp, also in ms units, marks the time the reply was sent, used for diagnostics.)

In order to know the calendar time of the event, a multicast message is sent each minute by one node that runs the `TIME` local application that indicates the time of the most recent `0x0C` event in GMT terms, expressed as a long word of seconds since Jan 1, 1900, plus a long word of  $\mu$ s within the second. (Event `0x0C` occurs at 15 Hz at Booster reset event time.) Each front end maintains this GMT time, updating it each cycle for the next minute. (To maintain accuracy, it calibrates its own crystal clock by monitoring `0x8F` clock events, which occur on calendar seconds.) To derive the time of the most recent `0x0F` event in GMT terms, the `0x0C` GMT time is adjusted by the time difference between the most recent occurrences of these two events. Armed with this reference event GMT time, and having a record of the time of each event's most recent occurrence, it is easy to compute the ms since 1970 for any event, for use in `GETS32` time stamps.

Another multicast message is sent by an Acnet node each 15 Hz cycle at the time of the `0x0F` clock event. It includes a 32-bit cycle number, plus a list of the clock events that have occurred since the last such message one cycle previous. Each clock event includes a counter of occurrences of that event, plus a 24-bit time stamp that denotes the number of  $\mu$ s since the previous `0x02` event, an event that occurs every 5 seconds exactly. From the point-of-view of Linac front ends, this message, arriving at about 50 ms after Booster reset event time, is too late in the cycle to be useful in detecting clock events occurrences. These nodes must know about clock events before updating the data pool and building replies for active requests. But hardware clock decoders are included in all of these front ends, in part to obtain the cycle interrupt that keeps task activity in synchronism with the accelerator, but also to provide other needed support. This support is provided by the "Digital IP board" in 68K nodes or by the "Digital PMC board" in PowerPC nodes.

This same multicast events message, even though its event information arrives too late, includes a 32-bit counter of 15 Hz cycles. This global cycle counter, actually a count of `0x0F` clock events, is used for the cycle time stamp in `GETS32` reply headers. The counter value that arrives soon after event `0x0F` is used for replies that are found to be due on the following cycle. It is taken as an announcement of the counter value to be used on the cycle starting in about 16 ms hence. (In this context, a 15 Hz cycle is the one between successive Booster reset event times.)

Because only the 15 Hz cycle number from the multicast events message is needed, only one node needs to receive it. That IRM node is currently `node06C3`, one of the Booster BLM nodes. It runs the local application `ACLK` to support reception of the multicast events message. Every 17 seconds, it multicasts a setting of the cycle number to the "Generally Interesting Data" area in all front ends. This is how all front ends know the same counter value. In between, each front end counts 15 Hz cycles by itself to maintain the counter used in replies.

### *Event plus delay*

Consider an event-plus-delay `GETS32` request. Every 15 Hz cycle, as soon as the data pool has been refreshed, the front end checks, for each active request, whether a reply is due on that cycle. If there is a new occurrence of the event before it notices that the delay has been reached, then we have a problem. The delay may have ended during the previous cycle, but the data pool now reflects the current cycle. If the data requested includes readings from hardware that contains a circular buffer of digitized data, then it is possible to "reach back" in the circular buffer and find the data of interest. (No such hardware is used in the Linac, because of its 50  $\mu$ s beam pulse. To monitor waveforms in Linac, multi-MHz digitizers are used; neither 1 KHz nor 10 KHz digitizers can do the job.) If the same data request includes other devices, those updated at 15 Hz, their data pool values will be incorrect, and the cycle time stamp will be wrong as well.

Take a specific case, that of a request for Linac data at event `0x10` plus 30 ms delay. The Linac front

end updates active requests at, say, 12 ms after its Micro-P Start time, which is 3 ms after Booster reset event time. Thus the 30 ms delay will not be reached for another 15 ms. We are simply not prepared to return such data. The principle is that we are not supporting 15 Hz data that occurs later in the cycle than Micro-P Start time, the time that we begin to refresh the data pool with current cycle readings. This time is only a few ms after Booster event time, so the use of requests based upon (15 Hz) event plus delay is not of much use in Linac. The exception is a request that specifies zero delay. Such a request is interpreted as applying to the data pool data that has just been refreshed shortly after the event has occurred, especially if the event is a Booster reset event.

Consider a similar request for a Booster HLRF node, specifying event 0x10 plus 30 ms delay. Since the Booster acceleration cycle lasts 33 ms, the Micro-P Start time for those nodes is about 35 ms, which is just after beam extraction. Such a request will be detected during the proper cycle, and if it is from the 1 KHz digitizer, it will be suitably correlated with the current data pool, and the cycle number returned will be correct.

Consider a Main Injector HLRF node, for which a likely event used in a data request might be event 0x23, which may occur every 2 seconds. A request for data for that event plus 30 ms will cause the front end, operating at 15 Hz, to detect that delay early in the 15 Hz cycle following the delay after event. It will reply with data just refreshed, and for circular buffer channel readings, it will find the data in the hardware buffer. No problem here, because the event is not a 15 Hz event.

### **Error return**

The logic for support of event plus delay in a GETS32 request has recently been changed so that, while awaiting the specified delay after event, if there is another occurrence of the same event, then an error is returned. This corresponds to the case described above for Linac, in which the delay reached past Micro-P Start time. That made it too late to be included in that cycle's reply data. Note that, should the event *not* be repeated on the next 15 Hz cycle, the request would be satisfied, and any circular buffer digitizer sampling would take place, but the data pool and cycle number will reflect the cycle *after* the event plus delay. There would not be an error return for this case, which may or may not be what the user desires. The software does not attempt to predict when a specified event will repeat in the future, whether that next event would occur before the delay is noticed, but if it does, global error status "36 -9" is returned.

### **Details**

The ReplyDue function is called for each active data request to determine whether a reply is due on the current 15 Hz cycle. It bases its decision on the contents of the data event string, which is analyzed during request initialization, and upon the event bit map that is updated at the start of each operating cycle. Return values are 0, 1, and 2, where nonzero means true; *i.e.*, a reply is due. The special return value of 2, applying to event-plus-delay requests, also means true, but an error status return is needed, because of a second occurrence of an event before noticing the delay.

To recognize a second occurrence of the event, a field in the request support block keeps as a reference the low 24 bits of the  $\mu$ s time of the event when the event was seen, initiating the wait-for-delay state.

For server requests, the ReplyDue function is also called early in the cycle, so that the server node reaches the same conclusion whether a delay after event has been reached. When the server task runs later that cycle, it uses the earlier determination to signal delivery of a complete reply.