

GETS32 Protocol Notes

Overview document ponderings

Wed, Oct 25, 2006

The GETS32 protocol is an enhanced replacement for RETDAT that is used by the Java Acnet consoles to request data from the appropriate DAEs. The DAEs, in turn, currently convert these request messages into the RETDAT protocol for transmission to front ends that do not support the more extensive GETS32 protocol. But in order to support all the new features allowed by GETS32, each front end must support it as well, so that the DAEs can merely pass on the original GETS32 request to the front ends. This note is meant to document reactions and interpretations gleaned from the GETS32 overview document that was written by Kevin Cahill dated October 8, 2003.

The stated new features include 32-bit length and offset specifications for each device, time stamps for resolving correlated data, and an expanded data event specification string. The 32-bit parameters are unlikely to be of much interest for our front ends. The correlated data solution, however, can allow for correlation between different replies across front ends, which is not easily obtainable with the current protocol and console data pool design.

The data event specification string is fairly general. It allows specifying, for periodic replies, whether an immediate reply is to be returned. (Right now, our front end support always includes an immediate reply, except for clock event-based requests.) The clock event case allows for specifying a delay after the clock event when the data is to be sampled. In our 15 Hz front ends, this has been handled for Booster reset clock events by assuming that the data sought is that to be found in the data pool that was produced during the 15 Hz cycle following the 0x0F event, which occurs about 16 ms prior to the Booster reset event. The new specification also allows for a "hard," "soft," or "either" type of event, the definitions of which are as yet unclear.

Time stamps

The time stamps included in the GETS32 reply message header are of three types, each of which is given in units of milliseconds since the start of the year 1970. This number currently occupies 40 bits, so that each time stamp comfortably fits within a 64-bit field. The cycle time stamp marks the occurrence of the 0x0F clock event that precedes the data collection event. The collection time stamp should specify the collection event time, or the time at which the data is sampled. Since all devices in a request share the same set of time stamps in the reply message header, this time stamp may be treated as a rough average. The reply time stamp is to mark the time at which the reply message is ready for delivery to the network. These three time stamps should occur in the order of cycle, collection, reply.

Return to the cycle time stamp. In order that all front ends have the same value for this time stamp, on each and every 15 Hz cycle, each front end should listen to the multicast clock event message that is sent following every 0x0F clock event. Within that message is to be found the time values from which one can derive the required cycle time stamp; thus, every front end will use the same value. It may be processed as a kind of a unique identifier for each 15 Hz accelerator cycle. It should not require much time to interpret this multicast message every cycle, but it will place a constant load on network processing that one can view via the Page F network diagnostics, sometimes referred to as a "poor man's Sniffer."

Data Events

The data event string specification has four types, referred to as immediate, periodic, event, or state transition. The string for each type allows for fields separated by commas. The immediate case is merely the string "i".

The periodic event string is something like "p,1000,true". The second field is the period expressed in milliseconds. The last field is a boolean string "true" or "false", which specifies whether an immediate reply is to be returned. The 15 Hz period is specified as "p,66,true", for example. We may round off this number to a whole number of cycles.

The clock event string includes fields for the event number in hexadecimal, the character "h", "s", or "e" to mean hard/soft/either, and a delay after an occurrence of the event. An example might be "e,1D,h,300". If we round off the delay to derive an integral number of cycles, this can cover anything in the data pool. One might consider for some cases whether we should try to sample readings more accurately in the KHz (or 10KHz) circular buffer.

The state event string includes the name of a state device, a decimal compare value, a delay, and a compare operator string. The latter may be "*" for any, "=" for equal, or "!=" for not equal. An example might be "s,v:CLDRST,9,1000,=", which means a delay of one second after the collider state device reading announces ("=") the inject pbars ("9") state. It is possible that a decimal device index value can be used in place of a state device name. It is not yet clear how we can implement this feature.

Server node support

We support the use of a server node for RETDAT, so we should also support it for GETS32. This requires examination of the SSDNs that are part of either protocol to discover the actual source node for the device whose data is requested. Following such examination, a decision is made whether server support is required for the current request. If it is, then the entire request is to be sent to a multicast destination so that all nodes can see the same request message. Not only that, preparation must be made for receiving replies from each contributing node and suitably sprinkling its reply data into the composite reply buffer.

We have used server nodes as a concentrator so that reply data stemming from multiple nodes can be grouped together, with an eye toward delivering correlated data. With the new GETS32 support, the time stamps are designed to solve the correlated data problem. Still, the server node provides diagnostics and also reissues requests to contributing nodes that appear to have dropped out. The latter facility serves to remind a node, following a reset, of the active data requests of which it is a part.

Assuming that we support a server node, how can we deal with the time stamps? Consider first how the time stamps are generated for the contributing nodes. Early in a cycle, each node should capture the cycle time stamp from the multicast message that was received about 16 ms before the current Booster reset clock event. It should also capture the collection time stamp that is appropriate for the current cycle, which means the time of Data Access Table processing, when the data pool is updated. Finally, when it has prepared a reply message for delivery to the network, it should include the time stamp that corresponds to "right now." To facilitate that, the local microsecond counter can be captured at the same time that corresponds to the previously captured collection time stamp. One can then easily compute a time stamp for any time thereafter during the present cycle. Take the elapsed microseconds, divide by 1000, and add the quotient to the collection time stamp.

Now return to the server node and its setting of time stamps for the composite reply message. It can copy the first two time stamps (cycle and collection) from the last received contributing reply message header. For the reply time stamp, it can deal with it the same way that a contributing node does. The reply time stamp should mark the time of delivery of the composite reply. It can easily be more than one cycle after the cycle time stamp, especially for

nodes whose “ μ P Start” time is late, such as the 40 ms used for a Booster node. This is because the reply time for server requests is scheduled at about 40 ms after μ P Start.

Note that the above scheme for server node time stamps works well for one-shots.

Sans multicast event message

What if the multicast message is not forthcoming? One can consider using a value for the cycle time stamp that is derived from the GMT0C time maintained by each front end that was developed for support of MiniBooNE. This result may not precisely match that which would have come from the multicast message, but it will be close. The format of GMT0C is a 32-bit count of seconds since 1900 and a 32-bit count of microseconds within the current second. To build a cycle time stamp for this, first work out the GMT value for the latest 0F event, subtract an offset equivalent to 70 years of seconds, multiply that result by 1000, and to that product, add the number of microseconds divided by 1000.

If the multicast event message contained such an accurate value for calendar time of the 0x0F event that prompted it, it could be possible for a node to create its own such time stamp, assuming that the client side processing allowed for at least a 1 ms deviation in the value compared with that included in the multicast event message header. Thus, a node that cannot receive the multicast message would still be able to support the GETS32 protocol.

Support via LA

It may be useful to support the GETS32 protocol via a local application. The FTPMAN protocol is currently supported by LOOPFTPM and may be used as a model. But there are complexities that must be handled by the LA, including waveform access, 7.5 Hz special handling, and data averaging. The name of the LA may be GS32. Diagnostics may include a GS32LOG data stream to log the arrival of requests. The request memory block type used to support each such active requests may be 0x072.

Multicast events

The multicast event message cannot be used for bringing event information to our front ends, because the event information comes too late. Consider a Linac node that starts its cycle at 3 ms after the Booster reset event. The most recent multicast message would have been received about 20 ms earlier, at which time the reset event was yet to be seen. And it will be about 50 ms after the Booster reset event (47 ms after the start of the cycle) before the multicast message arrives that carries the news about the Booster reset event that started this cycle. This news will thus not arrive in time to be useful in determining whether a reply message for an event-based request is due. We therefore need hardware clock decoding in each front end. Perhaps this means that we cannot support “soft” clock events as defined in the GETS32 protocol.

Multicast event message synchronization

Examine the timing of key events more carefully. The multicast message is sent out following the 15 Hz clock event 0x0F. It is of course received by all nodes shortly thereafter, so that it arrives well ahead of Booster reset event time. The ACLK local application current runs in node06C3 to receive this message to capture the low 16 bits of the cycle number (message counter) for use as a common cycle counter time stamp. Every 256 cycles, or about 17 seconds, it shares this counter value with all nodes by targeting multicast node 0x09F9. This sharing serves to insure that all front ends have the same cycle counter for use during the following 15 Hz cycle, beginning with their own μ P Start timer interrupt. This number is currently used for the 7.5 Hz data replies that support reliable 15 Hz waveform collection, originally done for Booster BLMs. The other use is for reading 15 Hz readings at rates of 1

Hz, say, where an array of readings is returned, along with the accompanying cycle counter.

Assume we have something like `ACLK` installed in each front end node, but with the `target node#` parameter set to zero. Then the present code will set the cycle counter only locally. In addition, we could have `ACLK` capture something else from the header that would provide the proper time stamps for the `0x0F` event as used in the `GETS32` protocol. This would be saved in a low memory global, perhaps, so it can be used by code executing in the following cycle.

At the start of a new cycle, maybe even when `GS32` executes its cycle activity, it can watch for a new cycle time stamp `cyclTSR` that was captured by `ACLK`. It can copy this into `cyclTS`, accepting it as the reference cycle time stamp for the current cycle. Also, it can measure how long it has been, in microseconds, since the last `0x0F` event, referencing the `0x0F` entry in the Event Times table, which can be captured as `cyclMic`. Convert the elapsed microseconds to milliseconds and add `cyclTS` to form the collection time stamp `collTS`. Use the `cyclTS` and `cyclMic` again later for building any reply time stamps for reply messages that become due during this cycle. Write a function called `GS32Now` that use `cyclTS` and `cyclMic` along with the current microsecond counter reading (the same one that is used in the Event Times table) to produce an 8-byte result in the format required by the `GS32` protocol. This function can be used to produce both the `collTS` and any later `replTS` needed. It may be useful to keep `cyclTS` as a low memory global to facilitate comparing values across front ends. The rest may be simply maintained in the static memory structure maintained by `GS32`.

But what if `GS32` sees no change in `cyclTSR`, implying that `ACLK` did not receive a multicast event message? We might manufacture a new `cyclTS` by deriving a `GMT(0F)` from `GMT0C`. Then convert that into the milliseconds-since-1970 format. The other time stamps can again be based upon this “best effort” result.

When a new server request message is processed, and the reply message block is initialized, copy the recent values of `cyclTS` and `collTS` into the header. This would suffice for the case in which no contributing node replies to give us those two time stamps to copy into the composite reply message header.

So much for planning. See the note, *GETS32 Protocol Support*, for the implementation.