

Classic Time stamps

Protocol Enhancement

Sat, Apr 21, 2001

Time stamps as described here are used at Fermilab to identify the 15 Hz acceleration cycle during which associated device values were measured. More specifically, it identifies measurements of accelerator devices for the cycle during which a specific Linac beam pulse was accelerated into the Booster, then extracted for injection into the Main Injector. It is an important tool for collecting correlated device data from different control system front-ends.

The meaning of the time-stamp is not really to specify time in the usual sense, but rather to specify the 15 Hz cycle on which the beam was accelerated. The units of time merely need to be 15 Hz cycles. In the accelerator control system, there is one front-end that multicasts a message containing a record of Tevatron clock event activity during the previous 15 Hz cycle. This message is transmitted following the occurrence of clock event `0x0F`, which occurs about 47 ms through the 66 ms cycle, as defined from the time of Booster minimum magnetic field, `BMIN`, when Linac beam is injected, to the next `BMIN`. This message also includes a 32-bit cycle count number, which can be used to identify a specific cycle and thereby serve as a time stamp. Here we interpret the meaning of a given cycle number as received from this message to be the associated time stamp for device data measured during the following 15 Hz acceleration cycle. It therefore announces to all front-ends ahead of time what cycle number will apply to the next 15 Hz device data. The use of this message as a source for the time stamp cycle number is that it is available to any front-end who is enabled to receive such multicast messages. Note that it is not necessary that a front end receive this message every cycle, as the cycle number for each 15 Hz cycle is always one more than that of the previous cycle.

Support was added last year for including this time stamp in replies to certain `RETDAT` requests. It was done to solve a problem that arises from the asynchronous operation of Acnet console application programs. These programs are invoked at a nominal 15 Hz rate, on the average, but this rate is not synchronized to Fermilab accelerator 15 Hz operation. (Scheduling of this nominal 15 Hz rate is accomplished by repetitively asking the operating system for time delays of 70, 70, and 60 ms.) Because of this, 15 Hz replies to the application program cannot be replied upon being complete. It can easily happen that two successive invocations of the application span the time during which two successive 15 Hz replies are received from a given front-end by the console data pool manager. Acnet consoles therefore cannot be expected to collect data at 15 Hz reliably.

The solution used to solve this problem was to have the front-end respond to a request for delivery of two 15 Hz cycle sets of data every other cycle. In order to make the scheme work, the application code watches for the arrival of new data every nominal 15 Hz execution, so that it is very unlikely to miss a 7.5 Hz delivery. The `RETDAT` support as used in IRM front-ends interprets certain requests for 7.5 Hz replies to be requests for a time-stamped data structure. This structure is a 4-byte header followed by two sets of data, each set measured on successive 15 Hz cycles. The header consists of two 16-bit words. The first word is a count, either 1 or 2, of the number of sets of data present in the reply structure. The second word is the low 16-bits of the 32-bit cycle number as delivered in the 15 Hz cycle just previous to that during which the first set of data was measured. The first set of data is to be associated with this value, and the second set with one more than this

value. The reason for the count word is that the first reply to any repetitive non-event request occurs immediately, referencing data already existing in the current cycle's data pool. The first reply of a time-stamped structure will therefore only have one 15 Hz cycle of data included. The space in the reply structure reserved for the second set will be meaningless (and probably zero). Subsequent replies will always include two sets of data, however. Using this approach to collect correlated 15 Hz measurements made it possible for an application program to capture correlated Booster beam loss monitor waveforms from up to 70 BLMs, arriving from 8 separate front-ends, at 15 Hz. In practice, it may be that all front-ends do not reply during the same 15 Hz cycle, because that depends upon whether they all received their requests for such 7.5 Hz replies during the same 15 Hz cycle. But the time-stamp values allow every data set to be identified so it can be correlated with the stream of data sets coming from any other front-end.

The above scheme was implemented in the support for the Acnet RETDAT protocol. But what about such support for the Classic protocol? One popular user of Classic protocol for many years has been the Macintosh-based Parameter Page program written by Bob Peters. To collect correlated data across multiple front-ends, its logic passes this potentially tricky logic to one of the front-ends contributing data for the overall data request in question. This is the server approach, in which a special flag bit is set in the request message that asks the receiving front-end to act as a server for all of the data in the request, forwarding the request for the device data included in the request that is to be sourced from other nodes via a multicast request message. When replies are returned to this server node, they are meshed into the complete reply buffer, and the complete reply message is delivered to the original requesting client. Since all front-ends of a single project operate their 15 Hz cyclic operation simultaneously, the server node uses a deadline to determine when to deliver the complete reply. Any data whose source node did not return its reply by that time—perhaps it was “down,” for example—causes an error status to be returned to the requesting node, indicating that not all data is correlated.

One may ask, why cannot the RETDAT protocol also use server support. The answer is that it automatically does so when it is applicable. If a front-end receives a RETDAT request message that includes devices that reside on different front-ends, it will act as a server for that request, multicasting the request to the rest of the nodes in the project and arranging the subsequent replies before forwarding the complete answer to the requesting node.

So one way for Classic protocol clients to deal with the need for correlated data is to ask for server support from one of the contributing nodes. Another way to do this is by slightly modifying the Classic protocol, which is the main subject of this note.

The Classic protocol will not be laid out in detail here. But its structure begins with 3 words of the following format:

Size	Total length of message in bytes
Dest	Destination node# for this message
cmdnd	Command includes message type

Examining the 3rd word for a request, it has three parts. The upper 4-bits specify the message type, bit #11 (mask 0x0800) specifies whether a request/reply is of the server type. The lower 11 bits are used for a “list number” that, with the requesting node, serves

to identify a specific request. When using UDP/IP protocols to carry the message, the “requesting node” specification also includes the source UDP port#, so that multiple client ports may, without confusion, specify the same list number to identify a specific request.

The 3rd word for a reply message has the same 3 fields, except that the message type is 0 rather than 2. If a request is a server request, indicated by the server bit being set in the request command word, then the corresponding reply will also have the server bit set.

The significance of the Destination node field is questionable. It can be used by a receiver to determine whether a request message was multicast, in which case, the `Dest` word has the value `0x09Fx`, allowing for 16 multicast destinations available. In the case of a reply message, it has no particular meaning. Currently, the receiving node of a reply message requires that it either match the local node number, or it be zero, without any distinction made. Any other value is assumed to be an illegal message.

This note proposes to use the `Dest` field of a reply message to carry the same 16-bit cycle number that is used in time-stamped `RETDAT` replies. This can allow a Classic protocol requester to correlate reply messages that come from multiple nodes. The changes required to make this new accommodation are simple. The receiver of a reply message should not look too critically at its `Dest` field, but rather accept it as the cycle number identifying when all the data in the reply message was measured. A new routine will allow an application to retrieve the latest reply data for a given request, so that it could also receive this “time stamp” cycle number. A possible routine for doing this is

```
GetDatC(short listNum, short *status, short *cycle, short *data);
```

When a server node receives multiple replies from various contributing nodes, each will carry such a cycle number. The cycle numbers from the contributing nodes should have the same value. That same value should be copied into the complete reply message that is returned to the requesting node. What if the cycle numbers in the contributing node reply messages do not match?

When a server request is forwarded via multicast to the rest of the nodes in a project, it is possible, though unlikely, that different nodes receive the same request message on different 15 Hz cycles. But when the server node receives any replies other than the first immediate reply, they are expected to arrive early in the commonly-interpreted cycle. The deadline imposed by the server is traditionally 40 ms after the start of the cycle. At that time, the server delivers a complete answer that is due to be delivered on that cycle.