

Acnet Data Requests

Overview of RETDAT support

Mon, Feb 3, 2003

Support for the RETDAT protocol is included in the system code of front ends used for Linac and for various other projects at Fermilab. This note is meant to provide an overview of the RETDAT support in these front ends.

Overview of an overview

A RETDAT protocol request is based on an Acnet header whose destination task name of RETDAT (in RAD-50 encoding) is followed by a 3-word header and an array of 16-byte packets that loosely refer to Acnet devices that are known to the Acnet database. This message is received by a front end via UDP port 6801. Via the third word of the header, called the frequency time descriptor, or FTD, it specifies when replies should be returned. A one-shot request implies that a single prompt reply is expected. It may specify that replies are to be returned repeatedly in units of 15 Hz cycles, since that is the basic operating cycle of the Fermilab accelerator system. (Both the Linac and Booster operate at 15 Hz.) The other choice is to specify that replies are to be returned on a selected Tevatron clock event, based upon decoding the synchronizing hardware serial clock signal that is distributed throughout the accelerator complex.

The specification of each Acnet device in a RETDAT request includes a device index, which is a key in the Acnet database, a property index, which roughly indicates the type of data sought about that device, an SSDN, which is an 8-byte structure of arbitrary design that comes from the database entry associated with the given device and property, a length, which is the number of bytes of data requested, and a 2-byte offset, which may be thought of as indexing into a large target structure.

Underlying data organization

Within the front end, all data that is accessible is characterized by two made-up index values. One is called a "listype," which is in the range 0–127, and the other is called an "ident." The meaning of a listype is roughly the kind of data that is sought. The meaning of an ident, when interpreted for the specified listype, is roughly "which one" of the specified kind. As perhaps the simplest example, consider listype 0, which means analog channel reading. The associated ident specifies the analog channel number. An ident may vary in size, but it is always an even number of bytes, with the first two bytes specifying a node number. In the example of an analog channel, the ident is 4 bytes in length, consisting of a 2-byte node number word followed by a 2-byte analog channel number.

SSDN specification

In order to map to the underlying data organization, we need to specify the listype and the ident in the 8-byte SSDN. The following structure is used for that purpose:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
ltyp	1	listype number
flags	1	offset flags (hi nibble), ident size (lo nibble)
node	2	node number
ident	2/4	ident beyond node number (2 or 4 bytes allowed)
pad	1	n.u. (only if ident size nibble = 1)
size	1	size per item (only if ident size nibble = 1)

A simple example of an SSDN for an analog channel reading property might be as follows:

```
0001 0611 0102 0000
```

Here, the node number is 0x0611, which implies its network name is node0611.fnal.gov, and the channel number is 0x0102. The listype number is 0x00. No flags are set. The ident size is 1, implying that one word is needed beyond the node number word. The size byte is not used in this example.

Flags are used for special cases, in order to allow definition of a single Acnet device that could allow

access to any user-specified channel number. If one specified the following SSDN:

```
0811 0611 0000 0000
```

It would permit access to the 64-byte analog descriptor record for any analog channel by allowing the user to specify the analog channel in the offset word. The flag bit that is set causes the support code to add the offset value to the ident index (here specified as 0x0000) and process it as if the resulting channel number were specified in the SSDN. One Acnet device is specified, presumably using the reading property, but one can access any analog descriptor record in the entire front end. The uses for such special devices are rather limited in practice, so that one does not normally set this flag bit. For the case that the ident size nibble = 2, which might work for a memory address ident, if the flag bit is also set, the 16-bit offset value is added to the 32-bit ident value. This would allow access to only a 64K byte region of memory with one Acnet device. But if the flag (hi nibble) is 2, and the ident size is 2, the offset is shifted left 8 bits before being added to the 32-bit ident value, thus allowing access to a 16M byte region of memory. Again, in practice, the flag bits are not normally used.

The use of the `size` byte in the 4th word of the SSDN is another option that is not often used. In the case that the user-specified size is larger than the item of interest but is an integral multiple of the item size, the data returned is an array of items addressed by a series of consecutive ident values. Again, consider a simple case, where the data sought is known to be stored in the analog readings of channels 0x0130 through 0x013F. By specifying a `size` byte value of 2, one could make a request that specified a `length` of 32 bytes, and the returned data would be an array of 16 readings of consecutive channels. This would be an appropriate SSDN:

```
0001 0611 0130 0002
```

Again, this option is not often used.

RETDAT message path

A datagram containing a RETDAT request, upon arrival at the front end, is routed to the Acnet dispatching task. For the 68K-based IRMs, the ethernet receive interrupt handler passes the message to the SNAP task, which provides UDP/IP interpretation before passing it to the Acnet task. For the PowerPC-based Linac front ends, the vxWorks NetTask passes it to the Socket Reader task, which has the `recvfrom()` call for all protocols, which in turn passes it to the Acnet task. In either case, the Acnet task has a pointer to the received datagram as it resides in a generic circular receive buffer of total size 128K bytes. The Acnet task recognizes the message as a request message and the target task name as RETDAT. From a matching entry in the Net Connect table, a reference to the request message is passed via a message queue to the ACReq task, which supports the RETDAT protocol.

All data requests in these front ends are handled in two steps: request initialization and request updating. During request initialization, errors are checked, and an intermediate structure is prepared for each device. During request updating, answers are built by interpretation of the intermediate structure, which is called the “internal pointer” structure. Request updating occurs every time a new reply is due, according to the FTD specified in the request. For one-shot requests, cancelation occurs as soon as the single reply has been sent. For other requests, cancelation does not occur unless a cancel message is received.

The internal pointer structure approach is designed to make request updating more efficient. It is analogous to programming language source code compilation that produces an intermediate byte code designed for efficient interpretation.

Server request option

The first part of request initialization, besides checking for errors, makes a determination of whether the request requires server support; i.e., shall the node receiving the request collect it from the various other nodes identified in the device SSDNs? This determination depends only on the node number fields in the SSDNs and on whether the request arrived via multicast. If all the node number fields match the local node receiving the request, then no server support is implied. But if any node number does not match the local node, and if the request is received directly (not multicast), then

server support is required. The first step in providing server support is to forward the message to a target multicast address, designed so that all other nodes of the same project receive a copy of the request.

Consider the case of one of the nodes receiving the multicast request from a server node. Because it was multicast, it interprets the request in a special way, ignoring all devices except its own. (If no SSDN includes its own node number, it simply ignores the request completely.) It arranges to build replies according to the specified FTD and returns them to the requesting node, which in this case is the server node that forwarded the request via multicast.

The server node, meanwhile, knows exactly what was in the request it forwarded, so it knows exactly where to place the replies it receives from the contributing nodes into a complete answer buffer. After allowing sufficient time to receive all contributing node replies, it then returns the complete reply to the original requesting node.

There are two reasons for providing support for server requests. One is that it simplifies the job for the client node, so that the client node does not have to receive so many replies. The second is that Linac data is designed to be supported in a correlated fashion; i.e., it should be possible to collect a set of Linac data that was all measured on the same beam cycle, even if items in the set originate from multiple front ends. The server logic is designed to operate synchronously, as all front ends in one project operate synchronously with the accelerator clock system. For example, all Linac front ends begin their 15 Hz operation every cycle at 3 ms after the Booster reset clock event, which is 1 ms after the 40-microsecond beam has been accelerated through the Linac. At that time, the Linac RF systems are idle, and the gallery is relatively quiet. Sample-and-holds are used to capture all beam-related data during beam time.

Other efficiencies

In order to reduce the number of datagrams processed by the front ends under heavy load conditions of many active requests, multiple messages of the same type that are destined for the same target node at the same time are concatenated into a single datagram, respecting maximum datagram size limits. To this end, each front end delivers replies to all of its active requests at once within its operating cycle. A linked list of active requests is maintained in order of requesting node and type. (This note considers only the RETDAT type.) As each request is processed to determine whether a reply is due on the present 15 Hz cycle, each reply message that is due is queued to the network. When the queue is flushed, all replies that target the same node will be contiguous and likely candidates for concatenation.

Note that for a project that uses a server node, many requests that are active and that pass through the server node will likely be concatenated by the contributing nodes, as they will all target the server node.

Another efficiency comes with a kind of load balancing. When many client nodes have issued requests that specify 1 Hz replies, they arrive at random 15 Hz cycles within any one second period. This means that replies to different client nodes will be generated quite often on different 15 Hz cycles. These requests may all be 1 Hz requests, but they are supported with different phases according to when the original requests arrived. A front end that receives a request specifying 1 Hz replies first returns an immediate reply, just as if it were a one-shot request, then returns replies at 1 Hz intervals from that first reply. For a server request, the server node knows all about this, so that it is therefore in a position to complain with either a 36 -8, or a 36 -7, error status if a contributing node's reply seems to be missing. The 36 -8 status is used if no reply has been received from that node since the request was started, which could imply that the node is not "up." The 36 -7 status means that at least one reply has been received from that node since the request was started. This means that the reply was tardy, or it got lost. One often sees 36 -7 errors if the various nodes involved are not operating in synchronism with each other. Within any particular 15 Hz cycle, contributing replies are sent to the server relatively

early. The server delivers the complete replies to the original client nodes at a time later in the cycle. That “deadline” time is normally 40 ms beyond the start of the 15 Hz operations in that project.

More RETDAT details

When a RETDAT request is initialized, it may turn out to be a server request or not, according to the content of the request message. For a normal (non server) request, three memory blocks are allocated to support it. One is the basic request block itself, which will be included in the linked list of active requests upon successful initialization. The second is a block to hold the internal pointer structures that are used when building replies. The third is the reply message block that is queued to the network after the reply data is updated. The basic request block includes pointers to the other two blocks.

For a server request, three memory blocks are also used. One is the basic request block, including a target node field that is used for forwarding the original request and for targeting the ultimate cancel message. The second is a block that holds the forwarded request message. The third is the complete reply message block. The additional resource needed for forwarding a server request is a message id that is allocated from a pool derived from the LISTP system table. Although a message id arrived when the server node received the original request, it is necessary to assign a unique message id to use for the forwarded request. This assigned message id is returned to the pool when the request is canceled. The logic that passes out message ids from the pool takes care that it does not soon re-use a message id that has just been returned to the pool.

Network handling of request/reply messages

After non-server request initialization, an immediately reply is produced by the Update task, which is told to follow the linked list of active request and reply to each new request. When the network message queue is flushed and datagrams are built and sent to the network, the messages in the network queue that comprised the datagram are recorded. When the datagram has been sent, those messages are marked to show that they are “done.” One of the jobs of the QMonitor task is to keep an eye on the network queue entries after they have been marked as done. Depending on the details, it may free the message block, or it may leave it as is. For a reply message to a one-shot request that is now done, it will invoke a routine called ACDelete that cancels the request, freeing up all its resources. For the case when a request is canceled for which a reply message has already been queued to the network, the reply message block cannot be freed until it is done. In such a case, the other two blocks are freed, but the reply message block is marked as to-be-freed. As soon as QMonitor task notices a reply that is done and has this marking, it frees the reply message block.

After server request initialization, the forwarded request message block has already been queued to the network. As contributing replies arrive, they are steered by the Acnet task to ACReq. The message id in the reply is used to look up the associated request block via the LISTP table. From the source IP address, the source node number is found via a network look up table. This is the key that is needed to distribute the reply data to the correct parts of the complete reply message. As soon as the last immediate reply from the contributing nodes has been processed, the complete reply message is queued to the network to be delivered to the original client node. For subsequent replies, the deadline time is used for delivery of complete replies. In this way, one-shot server requests are replied to expediently.

Averaging analog readings

One feature that is supported by the RETDAT code is averaging of analog channel readings. Most Acnet data requests that are not one-shots specify a reply rate of 1 Hz. But this is not very useful for Linac, as it implies sampling the readings every 15 cycles. Since the Linac only accelerates beam on particular cycles, a 1 Hz request might have a hard time finding such beam cycles. Of course, Linac data could be collected at 15 Hz, which would make all readings of a set of channels available. But Acnet clients do not want to receive data that often. (In fact, because Acnet clients run asynchronously with the accelerator and only execute application code at roughly 15 Hz, they cannot reliably receive

15 Hz replies.) They are much more comfortable with 1 Hz replies, as it is a more natural screen update rate for numeric values. To satisfy such 1 Hz requests, a kind of “smart averaging” scheme was designed. It preferentially averages 15 Hz data measured on Linac beam cycles, returning the average of all such beam cycles over a 15-cycle period. In the event that none of the 15 cycles includes a Linac beam cycle, then the average of all 15 non-beam cycles is returned. Using this scheme, an Acnet requester of 1 Hz data gets valid readings of beam toroids and other beam-dependent signals that reflect what is happening during all beam cycles.

Waveform access

Another feature specifically supported via RETDAT is access to waveform data. In order to recognize such a request, the `length` (number of bytes requested) must be larger than 2. (If it is 2, then the ordinary analog channel reading value from the `ADATA` table entry is accessed.) In addition, a `CINFO` table entry must be installed that ties the channel number to a waveform memory address. The offset flag in the `SSDN` should not be set, and the `size` byte should be zero. Then the returned data is copied from the waveform buffer. The `offset` value can be used to specify a byte offset within the waveform array, so that one can access any piece of the waveform buffer. This was originally designed to support access to the Booster BLM waveforms.

Time-stamped data

As mentioned previously, an Acnet client cannot reliably receive data replies at 15 Hz. For access to BLM waveform data, it was desired to be able to access all cycles and to do it across multiple front ends so that the requester can know that all the data was measured on the same 15 Hz cycle. A scheme for replying at 7.5 Hz with two 15 Hz cycles of data was implemented to solve this problem. A two-word header is included in the reply to indicate whether one or two sets of data are present in the remainder of the buffer, and a time-stamp value, actually a 15 Hz cycle number, whose value matches that delivered by the 15 Hz multicast event datagram. The time-stamp value applies to the first (or only) set of data in the reply buffer. A second set of data has a time-stamp that is one more. When the client receives a reply, it knows exactly to which 15 Hz cycles the data applies. It may receive replies from different nodes, but in each case, it knows which cycle(s) of data it received. Using the time-stamp value as a key to match up the recently-received data sets, it can achieve correlated data across all nodes.

The waveform data support described above can also be time-stamped in this way. If the user desires to receive 100 data points at 15 Hz, a request for 204 bytes should be made, specifying that replies should come at 7.5 Hz. The client program must look for new replies at 15 Hz in order not to miss anything. Although the clients cannot reliably receive replies at 15 Hz, it is assumed by this scheme that they can reliably receive replies at 7.5 Hz. A special analog channel called `B:BREVNT` was set up to hold the Booster reset clock event number that applies to the current 15 Hz cycle. This value can also be sampled using the time-stamped scheme, so that the reset events for all other data accessed via the same scheme can be known. To access this 2-byte value, request 8 bytes at 7.5 Hz. The two-word header will be followed by the reset event numbers for two consecutive 15 Hz cycles.

Clock event based requests

If the `FTD` specifies that data should be returned on a clock event, some special considerations apply. The front ends operate at 15 Hz, as has been made clear above. They operate at some delay from a 15 Hz clock event. (Linac nodes use a 3 ms delay for this. Booster HLRF nodes use a 40 ms delay.) If a clock event is specified, the meaning is taken to be that the data will be returned on the 15 Hz cycle following the specified clock event. Suppose that the specified event is `0x1D`, for example, which means that Linac beam is to be accelerated for MiniBooNE. That `0x1D` event occurs 3 ms ahead of when Linac nodes commence operation; therefore, they will know that the current cycle is a `0x1D` cycle and will produce reply data on that cycle. That same event occurs 40 ms before Booster HLRF front ends commence operation, and they will also consider that the current cycle is a `0x1D` cycle. If Booster extracted beam is the device requested, this interpretation will provide it in a consistent manner. Linac beam occurs in a 40 microsecond burst, whereas the same beam is injected into the

Booster and accelerated over the following 33 ms, after which it is extracted in 1.5 microseconds. Correlated data measured on 0x1D events does not mean that the devices were digitized when that event occurred, but rather that the beam that was accelerated following that clock event is measured. Even though Linac beam is measured at a different time than Booster beam, it is the same beam and is therefore correlated.

The additional characteristic of clock event based requests is that no immediate reply is returned. Replies can only be returned on each occurrence of the selected clock event, as described above. For a server request, it is still possible for it to detect that a node has not responded, because the server node, as do all nodes, know about clock events. A clock event decoder is included in each front end, and the Tevatron clock signal is plugged into each node. To again take the former example, if the server node sees that clock event 0x1D has occurred within the previous 15 Hz cycle, it can recognize whether replies have just been received from the various contributing nodes.