

FTPMAN Snapshots

Fast digitized data

Tue, Apr 8, 1997

Introduction

Snapshots are captured digitized waveforms, in this case acquired via the Fast Time Plot Manager (FTPMAN) protocol used in Acnet at Fermilab. This document describes the implementation of support for Swift digitizers, designed to operate on 8 channels at rates of up to 800KHz, and Quick digitizers, a family of commercially-available VME boards with sample rates in the range 1–10 MHz.

IRM 1KHz digitizers—*for slow waveforms*

Internet Rack Monitors (IRMs) have supported 1KHz digitized data accessible via Acnet's Fast Time Plot Manager (FTPMAN) for some time via the *continuous* plot option. The selective data return feature of the protocol can be used to limit the return of data to the time window defined by the user for the plot in order not to inundate the host with data ineligible to be plotted. The IRM hardware digitizes all 64 A/D channels at 1KHz, placing the results in a 64K-byte circular buffer memory. As a result, any number of users can request such data without interference. Users share access to the data as easily as they share use of the clock on the wall.

Swift digitizers

For faster digitizations, a new analog interface board has been developed that is called the Swift Digitizer. It supports 8 channels at rates up to 800KHz. It also has a 64K-byte memory, so that 4K (16-bit) words house the captured waveform for each channel. A digitize sequence can be triggered by any of the fixed set of Booster reset clock events (11–17, 19, 1C), which make up accelerator 15Hz, or it can be triggered by any selected single clock event, in each case plus an optional delay of up to 65535 μ s. For the case of the Booster HLRF IRMs, for example, one might use a digitize rate of 100KHz, so that a waveform of the entire RF pulse can be captured in the 4K-point memory that would cover 40.96 ms.

Digitizer sharing

A simplification in the design of FTPMAN snapshot support could be realized if the Swift digitizer configuration were considered to be fixed. In that case, each user would not be able to choose the configuration options for each snapshot taken. After some consideration of this option, it was decided *not* to adopt this simplification for the Swift digitizers. (It was adopted, however, for the Quick digitizers.)

Customized data acquisition

Allowing the *client* to specify the parameters of the digitizer is more user-friendly from the point-of-view of an individual client user. It may not, however, be user-friendly to another user who is looking at the same waveform—or another from the set of 8 signals attached to that digitizer. What is needed, therefore, is a manager of the snapshot digitizations. The local application SWFT provides such management. To do so, it needs to detect when a new waveform has been collected. It operates at 15Hz, but a digitizing sequence can complete as quickly as 5 ms, assuming the maximum 800KHz digitize rate, or as slowly as 0.65 seconds (about ten 15Hz cycles) assuming the minimum rate of 6.25KHz. Use of the not-busy interrupt from the Swift digitizer IndustryPack (IP) board can enable detection of a complete new waveform; however, this feature didn't work in early versions of the hardware already installed. A second approach is the delayed timer interrupt that, except for the external trigger case, indicates that the digitization sequence has begun and that the memory address register is counting. By watching this address register until it becomes large enough to include the number of waveform points requested, SWFT notices when the digitization sequence is complete and sets a completion state variable accordingly. FTPMAN, in turn, notices this completion status set by SWFT. Then FTPMAN captures the waveform into an allocated block of memory for later perusal by the client.

In order for SWFT to maintain proper control over the collection of new waveforms, it must manually arm the digitizer. (One might rather use the auto-trigger control bit, but it can allow a new digitize sequence to be initiated by a second trigger without guarantee that the server has noticed that the data from the previous waveform has been collected and has a chance to capture it.) Manual arming allows the server to control the hardware activity so that the waveform data can be captured before another trigger occurs to alter it.

FTPMAN snapshot protocol

The FTPMAN protocol for snapshots works in the following way: The client sends a request that identifies the common set of parameters—event, delay, sample rate, and number of data points—for up to four signals whose waveforms are to be collected. The server IRM checks whether the indicated signals have Swift digitizer support. Status messages, returned to the client a few times a second, include the set of parameters honored for the measurement, and they announce progress on snapshot data collection—waiting for event+delay, waiting for digitizations, done. When a snapshot is complete, the entire waveform is collected and saved for client retrieval, thereby allowing the digitizer hardware to be operated on behalf of other snapshot users. When the client receives status indicating a given waveform is complete, it issues one-shot requests to collect the data from each signal, in response to which the server merely copies from the captured buffer for the given waveform. When a cancel message is received, or a new request is received from the same client node/task, the server resources associated with that request are freed. As an option, before a client cancels a request, it can ask that the snapshot be repeated, without formally making a new request. In this case, the server waits for another snapshot to occur and captures the data again, re-using the same resources already reserved for the original request.

Simplifying assumptions for parameter selection

In order to make an easy user model for parameter selection, a few simplifying assumptions may help. For digitizer start control, assume that only the delay timer is used. (This means that neither an external trigger nor the CPU itself initiates a digitization sequence.) Assume that only 1MHz resolution is used for the timer delay. Assume that a channel's value can be encoded automatically into the appropriate 3-bit code to pick the rate. We have the following four parameter values that describe the choices made available to the user:

event#
 delay in μ s
 rate in KHz
 #points

A value of event=FF means that the 15 Hz fixed set of Booster reset events is selected, else use the event indicated. A value of rate=0 means that the digitize rate signal is external, else it will be found among the values 800, 400, 200, 100, 50, 25, 12, 6. The delay is in the range 0–65535 μ s.

A local application (SWFT) manages the operation of the Swift digitizer. It does the manual arm when it is ready to collect a new waveform. It examines the user-selectable channel parameter values to control the hardware accordingly.

Message queue scheme

The SWFT local application gets its instructions from a message queue. At a time when it believes that the hardware buffer is dormant and all active requests have captured the data, it reads from the message queue. If a new command is found, the parameter values therein specify the event, delay, rate, and #points to be measured. These values are loaded into the Swift digitizer registers, the current trigger delay interrupt counter is sampled, and an arm is delivered, thus enabling the board to detect a trigger that starts the digitization process. Once the event trigger occurs, and the delay times out, an interrupt is generated that merely increments the delay interrupt counter.

When SWFT notices that the interrupt counter has changed, it moves into a different state of monitoring the memory address register. When it notices that the memory address is high enough, it knows that the digitization is complete, and it sets the state variable to zero to so indicate. (Since auto-trigger is not used, the hardware memory will remain stable until it is re-armed.)

For the FTPMAN local application, when a request for a snapshot is received, the set of parameters specified is loaded into the message queue. Status is returned to the requester a few times a second, giving the state of the measurement progress. When the memory is stable, the status is changed to reflect that fact, and the desired waveform data is copied into an allocated memory block for expected client-directed perusal via one-shot requests. From this point, since the waveform data has been captured, FTPMAN is no longer holds up SWFT checking the message queue for another set of parameters and making a new measurement.

Duplicate snapshot parameters in queue

The message queue read by SWFT is searched prior to adding a new entry to determine whether the new entry needs to be added at all. If the queue contains a given set of parameters that matches the set to be entered, there is no reason to queue up the new set, as it has yet to be used. If multiple users select the same parameters, they share viewing the same data. If their parameters are different, however, they will be served by SWFT in turn according to their order in the queue.

FTPMAN plan

Accept parameters, place into queue, and wait for SWFT to signal completion via a state variable. When completion is signaled, check the parameters of the measurement. If they match the current request, then capture the needed data, else await the next completion. During the wait for completion (and match), report to requester the progress status at 15Hz. Once the waveforms are complete, slow down the response rate to 2 Hz. Entertain client requests for the data points via one-shot requests. If a re-measure command is received, queue up the same parameters specified in the original request. When a cancel request is received, or a new request is received from the same task, free all old allocated resources. Note that the queuing may not have to take place, if the parameters match something already in the queue. But FTPMAN must wait for a new measurement to be made; it cannot supply stale data that was already measured before a request is received. If two FTPMAN users are operating simultaneously, and if they are using the same parameters, then each will see the same data sets, because each will notice the same completion status from SWFT at the same time. Since the parameters match, each will capture the same data points. (Actually, they might be capturing different waveforms on the same board, so that the data sets might differ.)

SWFT plan

Parameters needed for SWFT are:

enable bit#

status chan# (event chan#, delay chan#, rate chan#, #points chan#, intr cntr chan#)

address of Swift digitizer registers (2 words)

The status chan# is the initial one of a sequence of diagnostic channels, so that only four words of parameters are needed altogether.

At initialization, create the "swft" message queue if not already available. Enable delay timer interrupt. Set interrupt counter to 0. Set state 0.

Perform the following actions at 15Hz:

State 0:

Read from queue. If queue entry found, and if parameters ok, load into registers on Swift digitizer

IP board. Update status and a copy of the parameters in channel readings. Capture delay interrupt counter value. Arm hardware. Go to State 3 until at least the next cycle.

State 3:

Watch delay interrupt counter. When value changes, a timer delay has finished following an event trigger, and the digitization sequence has begun. Go to State 2 immediately.

State 2:

Watch memory address register. When it becomes $\geq \#points * 8$, stop, record completion status. Go to State 0 until next cycle. (In between, FTPMAN notices this completion status by detecting a nonzero to zero change in state#. It then checks for a match with a given request's parameters, and if there is a match, it captures the waveform data requested from the hardware memory.)

Watching the state variable cycle by cycle, it is 0 until there is a command found in the message queue, at which time it changes to 3. It will remain at 3 until the selected clock event occurs and the delay has timed out. If by the next cycle the digitizing process is not complete for the number of points requested, the state will be set to 2, else it will change to 0, signalling that the waveform(s) are ready to be sampled. Only a single cycle is guaranteed available for FTPMAN to capture the data, so that SWFT will be free to setup another snapshot from another command.

Use of data stream as message queue

An advantage to using a message queue implemented via a data stream is that it is easy to adapt the setting log page application to show the contents of the queue as a diagnostic log. The structure for both the setting log and network frame data streams is a circular buffer of 16-byte records. In each case, the last 8 bytes give the time-of-day. For this snapshot data stream queue, we can thereby capture the time-of-day of recent snapshot activities. The first eight bytes include the event, delay, rate, and #points. The SWFT page application makes a data request for the data stream called SWFTCMND, formatting the results for display.

When an entry is to be placed into the data stream queue, a means of checking the previously queued, but not yet complete, entries is needed. As stated before, we need *not* add an entry to the queue that matches one already waiting. If DSWrite were used to write into the message queue, the matching function would not apply. So FTPMAN uses a queuing routine that checks for a duplicate entry first.

There is no way to detect a full data stream by definition; although there is an IN pointer, there is no OUT pointer. One must be sure that the queue cannot ordinarily overflow, although an error return can be used if a new entry cannot be placed due to the queue being full. This isn't expected to be a problem, especially with logic that doesn't repeat entries that are already waiting.

A communication area that is defined in the header of the data stream queue is used to store the information about the current digitization parameters—the status, event, delay, rate, and time of queuing. In addition, an OUT pointer is maintained so the matching can be done against a tentative set of new parameters. The SWFT local application advances OUT only after the waveform is stable, so that a new matching parameter set can be queued as soon as the current one is finished, in case a user wants to repeat the same snapshot that was just completed.

Cancelling a snapshot request

If the host cancels a snapshot request, it is desirable to remove it from the queue in the case that it was not complete. This is especially important if the request that was queued specified an event that will not occur. For as long as such a request is active, the status returned will indicate that the event has not occurred. Without removing it from the queue, any other snapshot

command, either already queued or yet to be queued, will be held up. Since the command that is current is still in the search range, we should find it with a search that refers to the common OUT offset in the diagnostic area of the data stream queue header.

But finding it in the queue is not enough to be able to remove it. Because a command that is already queued that matches a new one does not itself get entered in duplicate, a use count is necessary to keep track of how many users share that command. The high byte of the word in the command that specifies the 8-bit event# can be used for this purpose. Use the upper two bits for status information about the entry, to be described later. Then the lower 6 bits can be used as a use count. When a command is to be queued that matches one already in the queue, merely bump the use count and refrain from entering a duplicate command. When a request is cancelled, and its command is still in the queue, decrement the use count. If that decrementing makes the use count zero, then mark it deleted. We can't actually remove it from the queue because of the nature of the queue. But we can mark it so that when FTPMAN reads from the queue and finds an entry so marked, it can simply ignore it and look for the next one. Also, while SWFT is watching the current command to detect when it finishes, if it notices that the current command in the queue becomes marked for deletion, it should abandon waiting for the current one to finish and immediately read from the queue again to find another snapshot to initiate.

If a user does not cancel a snapshot request, the allocated resources remain in use, and there is no protection for a user requesting a snapshot that waits on an event that never occurs. It is not clear how to deal with this potential problem. One cautionary step can be taken, however. When a request is received that specifies a clock event# that has not occurred in a long time, it can be refused and an error status returned. For IRMs, all clock events are known. The relative time for each clock event is recorded, in μs units, and the delta time between the last two such events is computed. But if enough time (30 minutes) passes without a given event's occurrence, this information is cleared. So if a new request is received that requests a snapshot triggered by a given event, but the delta time field in the event table entry for that given event is zero, the request can be rejected. This may perhaps be inconvenient for a user who wants to set up a snapshot to be triggered when a very unusual clock event occurs, but if he is allowed to do it, no other snapshots can be made for any of the Swift digitizer data in that IRM as long as that special case holds up the command queue.

Status of Swift command queue entries

Status of each command queue entry is maintained in the upper two bits of the word containing the event# parameter. The SNAP page application shows this information symbolically. The four states and symbols used are as follows:

00	command queued	(blank)
01	command current, waiting	—
10	command complete	.
11	command aborted by cancel	*

The entry date and time is updated according to when the status was last updated.

Server mode

For a well-distributed system, a data request that includes a list of devices from many different front end nodes can result in heavy network reply activity directed to the requesting client node. Client nodes in Acnet have been declared to be non-real-time nodes. They have many jobs to do in supporting X-windows and user interfaces, so that they cannot be expected to handle too much network traffic. Therefore server mode was implemented for the IRM nodes and other local stations. With server mode supported, a client node can send a data request that includes many devices to a single server node, and it will subsequently receive replies only from that single node, representing the composite of replies from possibly many other contributing nodes that hear

about the request from the server node. As a result, network-handling requirements are minimal for the client requesting node.

Server mode is supported for FTPMAN requests. This means that a server node may receive a request for data that is actually sourced by another node or set of nodes. An SSDN is received for each device in a data request, whether it be for RETDAT or FTPMAN. The second word in every SSDN supported by IRMs and/or local stations is the source node#. This allows a node that receives a data request to discover what nodes source the data. By analyzing this information, it can determine how the request will be handled, whether server support should be applied or not. If all the devices in a request are local devices, or if the request was received via multicast addressing, then direct support is granted to the request; the node replies with only its own device contributions to the request, and any devices in the request that are sourced from other nodes are ignored. On the other hand, if the devices in the request are sourced from at least one node that is not the local node, and if the request was received directly (not multicast), then server mode support must be granted to the request.

Server mode means that the node receiving the request must act as a server. It forwards the request to a target destination. If only one non-local node is represented in the received data request, then the target is that node. If more than one non-local node is represented in the request, then the target is a multicast address that reaches all candidate nodes. Replies subsequently received from the contributing nodes are used to build up the ultimate reply to the original requesting client node. In order for this to work, the assumption is made that all nodes operate simultaneously; they are each triggered at 15 Hz at the same time. A server node assumes that 40 ms into the current 15 Hz cycle is the deadline for replies that are received from the contributing nodes. At that time in each cycle, if it is the cycle on which a reply is due the client, server mode replies are transmitted. For the special case of one-shot requests, in order to improve response time, server replies are delivered as soon as the last contributing node has replied to the server.

The above description of server mode is a property of any node. Any node can support a server-style request, as all nodes run the same software. In practice, however, in the case of Booster HLRF as an example, only one node is identified in the central Acnet database as the source for all Booster HLRF data. This means that all Acnet requests for Booster HLRF data target that single node, which in turn forwards the request to and deals with the subsequent replies from up to 18 other nodes. This can cause a lot of network activity, but IRMs are real-time front ends. They use the network efficiently, especially in consolidating multiple reply messages that target the same node into common datagrams. Note that the replies to multiple requests that pass through the common server node will, if due on the same cycle, similarly be consolidated into common datagrams to the server node.

The FTPMAN protocol imposes special considerations in providing server node support. According to the FTPMAN protocol specification, only a single request from a given client task can be supported by a given front end. If a subsequent request is received from the same client task, while a previous FTPMAN request from that task is active, the previous task must be cancelled and the new request then accepted. But when forwarding an FTPMAN request, a server node must not merely forward the clients task name, which is part of the actual FTPMAN request message, not part of the Acnet header. (If it did, there is danger that another client node might also send a request that used the same task name through the same server node, and when the server node forwarded the request, previous requests would be automatically cancelled by the contributing nodes that were in common between the two requests.) The server node must therefore make up a unique task name and replace that which it received from the client, before forwarding the request to the contributing nodes. Not only that, but the server node must also make up its own message id to be used in the Acnet head, not merely pass on the same one received by the client. As a simple

solution to the task name problem, we can merely use this same message id as the task name as well, as that value will be unique across multiple requests that are forwarded from the server node.

An additional wrinkle to server support for the FTPMAN protocol stems from the data retrieval request specification. A data retrieval request refers to an already active snapshot request. This correlation is based upon a match with an active FTPMAN request that was received from the same client node and which specified the same task name in the request message. A server node, in forwarding a data retrieval request, must therefore use the same task name that it made up when it forwarded the original snapshot request, in order that the contributing node may also make the proper correlation with the original snapshot setup request. And because the contributing nodes, in this implementation, squeeze out of the request all devices that are non-local, the data retrieval request, when forwarded by a server node, must renumber the device index (range 1–4) according to the single targeted contributing node. Note that a single node must be targeted rather than a multicast target address that may have been used for the forwarding of the original snapshot setup request.

Quick Digitizers

Quick digitizer hardware is used in the Linac area to digitize waveforms at rates typically between 1–10 MHz. The Linac control stations consist of a VME crate with a 68020-based CPU board and about 4 other boards. The Quick digitizer is a VME board that supports 4 channels, all driven at a common sample rate. There is memory enough for 64K samples for each of the four channels. A gate duration register holds the number of points to be digitized, so that one needn't always digitize 64K samples. The Linac beam pulse may be only about 50 μ s in length. If a board operates at 5 MHz, this would be covered by 250 points, rather less than 64K!

Consider how to interface these QD boards with FTPMAN. Each board is installed to operate in a certain way, so that the user should not be expected to alter its parameters. This is a great simplification of the problem. It means we do not need to implement a Quick digitizer command queue and manager program, as was done for the Swift digitizer support. But we do need to detect when a waveform is complete, as the triggering event may not occur at 15Hz. When a user of FTPMAN makes a request for a snapshot, he has a right to expect that only new data will result, not stale data. Upon receipt of a snapshot request, by examining the gate duration register and placing a data word at the end of the indicated waveform memory, FTPMAN can notice when the value is overwritten, thus indicating that the desired waveform is complete so that it can be captured to await the host's one-shot data retrieval requests. Note that multiple requesters do not interfere with each other, because of the look-but-don't-touch approach taken; the hardware registers are not modified. This means that multiple users are always assumed to be using the same parameters, and sharing the data is an automatic benefit. Still, because of the nature of the FTPMAN protocol, it will be necessary to capture the waveform into a separate buffer to satisfy leisurely data retrieval requests from the client.

With two possible types of snapshot hardware, there is the problem of determining which the user wants to use. If we assume that a given system would not have both Swift digitizer and Quick digitizer hardware for the same channel, then there should not be a problem. We may want to limit all devices in a request to use the same type of hardware. The two types of digitizers won't likely operate at the same sample rates.

Protocol support in toto

For the complete implementation of snapshot support for IRMs, then, both non-server as well as server support was written for the new snapshot setup request (typecode 7), the data retrieval request (typecode 8), and the snapshot miscellaneous request (typecode 5). The latter is

used for sequential data retrieval pointer reset and also for restarting the same snapshot that was used in the original request. All of this was done for both Swift and Quick digitizers.

In addition to the FTPMAN server support itself, three other programs were written to assist during the development. The first is the SWFT local application that manages the Swift digitizer command queue, as mentioned above. It accepts commands from the queue and passes them to the Swift digitizer hardware to perform each waveform measurement in turn, allowing one cycle for a user such as FTPMAN to capture the waveform data it needs from the hardware memory, before it examines the next entry, if any, in the command queue. It also maintains a communications area in the header of the data stream SWFTCMND that functions as the command queue, indicating the current state of the snapshot progress. The FTPMAN support monitors this state information to provide continuous status replies for the requesting client.

A page program called SWFT was also written to monitor the contents of the command queue. Because the command queue was implemented as a data stream, it can display It uses the listype for accessing data stream queue records to collect the most recent Swift commands found in the queue, presenting it in a form similar to what was done for the setting log activity and for recent network frame activity. The status of each command is also shown, so that one can monitor the successful completion of each command. If a command is aborted, such as when a snapshot request is cancelled before the command has been fulfilled, it will be so indicated in the resulting command list.

Another page program called SNAP was written to exercises the snapshot client protocol for testing the functionality of the FTPMAN snapshot support. The present version only supports a single device in the request, but it can be sent for both the server case and the non-server case. When the returning status updates indicate that the requested waveform is available, it sends a data retrieval request to collect the first part of the data and displays it in hex. This snapshot test page program also supports the request for plot class information as well as the request to restart an already existing snapshot request.

Program sizes

The following list shows the current size of each supporting high level language program relating to the implementation of FTPMAN snapshots for IRMs.

<i>Program</i>	<i>#lines</i>	<i>Code</i>
LOOPFTP	2710	16K
LOOPSWFT	583	2K
PAGESWFT	680	4K
PAGESNAP	697	4K