

Timer Trigger Events

Acnet access method

Mon, Jan 26, 2004

Introduction

Many timer modules of the 177-variety exist in Linac, MiniBooNE, MIRF, and TRF. Each contains a 256-byte array that is sufficient to house trigger bits for each of the 8 timers for any set of the 256 possible clock events. This trigger RAM is designed to be efficient for the hardware to process as it decodes each clock event that it sees. It is not such an easy software interface, however. The software support in these systems expects that a set of placeholder event channels is associated with each timer, one event channel holding one event number for its associated timer. In practice, there are typically 8 such channels allocated for this purpose for each timer, which means that up to 8 events can be set to trigger the timer. One can see what channels are associated with a timer by looking at the readings of the relevant set of event channels for that timer. This approach is described more fully in the note called *New IP Timer Board*.

Acnet consoles use a different scheme that is supported by the Parameter Page client written by Jim Smedinghoff. If an Acnet device is known to be a timer of the 177-type, which it can get from the `SAVE` property for that device, and if it has a `SIBLING` device, and if that device has a `SETTING` property, then that property can be accessed to get a list of clock events that are related to the original timer device. The form of the 16-byte structure used for this is an array of bytes, in Vax byte order, where the first byte is a count in the range 0–15, and the following bytes hold the set of events that trigger the timer. Obviously, this scheme has a limit of 15 events that can trigger a timer, but in practice, this is not a problem. It also possible to establish a new set of trigger events by performing a 16-byte setting to that same `SETTING` property. This note discusses how to provide support in the front-end for this client logic.

New listype

The usual way to provide access to a new data structure that does not already exist in memory is via a new listype. Suppose a new listype is defined to use the usual analog channel ident format, in which the channel would be that of the timer delay device. To find its associated event channels, the family word in the `ADESC` entry would be used to build a list of such related channels. When a data request for 16 bytes is made that targets the `SETTING` property of the `SIBLING` device, the sequence of event channels is scanned and the 16-byte structure built in the proper format to reflect all the events that trigger the timer delay device. To implement this listype format, only new read-type and set-type routines would be needed.

By defining a new listype, both Classic and Acnet `RETDAT` protocols would work, although it may not be used by Classic protocol clients.

Set-type routine

The support for setting a new event structure is a bit tricky. Suppose we try to avoid altering the hardware except where it is necessary. The logic will work with a set of event channels and a set of events to be set. Scan the set of event channel values, and for each one, check whether it is also in the array of events to be set. Only if it is not, implying that the user desires to remove that event from the trigger RAM, is a zero written to that channel to clear it. After that scan is finished, scan the events to be set. For each one that is not in the event channels, it means that it is a new event to be set, so find a cleared event channel and set it to that event. If there is no event channel cleared, then the setting cannot successfully be made, and one should return an error code.

Because this logic is not trivial, can it be done by a local application instead? One might define a data stream that could assist by serving as a message queue to hold the 16-byte setting data plus some timer addressing information, perhaps the targeted channel. It could also provide some diagnostic record of such settings; a 32-byte record format could be used to include time-of-day and source node, say.

Vax byte-order

Acnet uses little-endian byte order, so their client support expects the 16-byte structure to be in that form. One could work with it in the usual big-endian form, then swap bytes before sending to the Acnet client, or one could build the array in a sneaky way that ends up with the expected result. For example, in building up a new 16-byte structure, suppose an index value is initialized to the value 1, which corresponds to the count byte in the structure, and the count byte is cleared. When adding a new element to the array, increment the count byte at offset 1, advance the index, then store the 8-bit event number using that index. The logic for advancing the index can be as follows:

If the index is odd, subtract one.
If the index is even, add 3.

This logic produces the following series of index values:

0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14

These can be used to store up to 15 events in a little-endian array.

Data file approach

As an alternative to using the family word in the ADESC entries for all 177 time-related channels, a data file could be used to store the information necessary to make the connection between the channel number used for the timer delay and the first of a series of channels used to house the trigger event numbers. A separate data file would have to be created for each node. It would include a delay channel number and the first event channel and the number of event channels associated with that timer. Conceivably, it might need to allow for more than one series, in case the original series had to be increased in length to accommodate more event channel possibilities.

Memory buffer approach

Consider another method for this support that would not require a new listype. Reserve some memory to be used for each timer, the contents of which is the 16-byte structure referenced above that may be sent by an Acnet client. A local application would be written to manage all of the required logic. Every update period, perhaps every 15 Hz cycle, the LA would review all the event channels for that timer and determine whether the 16-byte structure accurately reflects the current set of events that trigger that timer. Only when there is a mismatch would something be done, which would include building the 16-byte structure according to the set of trigger events that currently exist. For the next update period, then, nothing different is noticed, so nothing happens.

The tricky part comes when an Acnet client makes a setting that overwrites the 16-byte structure with something that does not reflect what events are presently assigned to that timer. How can the LA know that it should perform the required settings, accepting the 16-byte structure as what is desired, rather than simply overwriting it? Suppose that anytime the LA builds a new structure that shows what is in the trigger ram for that timer, it also retains a copy of it in a separate reference buffer. Then, when it discovers that the current set of event

channels does not match what the 16-byte structure says, it compares the 16-byte structure with the reference structure to see whether it has been altered by someone else. If it has, then the 16-byte structure is taken as the desired set of trigger events for that timer, and the appropriate changes made.

Although there have been a few Acnet memory-addressed devices defined in SSDNs, one is normally a bit reluctant to place absolute memory addresses into the database. To improve this situation, we can make use of a recent modification to the system code that provides a new listype that allows access to LA static memory. This way, instead of memory addresses, we would use offsets with a certain LATBL entry. That static memory is dynamically allocated, so that we do not have to statically define memory for this purpose, and we can ensure that the offsets that apply for these buffers remain stable, since that would be part of the LA design.

Using an LA to do this with such memory buffers means that supporting this feature, which is admittedly a kind of kludge, does not imply addition to the system code itself. Any node with such timers would simply need to have this LOOPTRAM LA installed.

LA design considerations

What would the parameters of the LA have to be? Assume that the delay channels for the timers are in sequence. We could use one parameter for the first channel number in the sequence of timer widths, plus another to hold the count of channels to scan. For 8 timers, the count would have to be 16, since the actual sequence of defined channels is delay, width, delay, width, etc. As for what offset to use within the static memory block allocated by the LA, one could use another parameter, just to make it obvious to a DABBEL user who can interpret the Page E display that shows the LA parameter values. Perhaps better, one could just "know" the offsets from a document that describes the LA. Assuming that we only need a parameter for the starting channel and count, we could support more than one timer module with a single LA instance. But perhaps it would be better to use more than one instance of the LA, each one assigned the task of managing 8 timers. In that way, the offsets to be used could be the same, since each instance has its own static memory block.

To be specific, suppose that the offsets used for the memory buffers start at 0x100, and each buffer is 16 bytes in size. Then the offsets used are 0x100, 0x110, 0x120, etc, for the 8 timers. Within the same static memory block can be the LA's own reference copy of the buffers, say starting at 0x180. This assignment allows for 256 bytes of LA static memory variables for diagnostics and other purposes preceding the buffer areas, but it gives offset values for the memory buffers that are easy to remember. If 128 bytes is enough for this other purposes, the reference copies could start at 0x080.

When the LA is initialized, it clears out both the 16-byte structure buffers and the reference buffers. This often occurs soon after reset, in which the settings restore has just been done, updating the hardware to reflect the event channel settings. The first time the LA scans the delay channels of a timer, it sees that the event channels do not match the 16-byte structure, but the structure matches the reference buffer, so it merely writes a correct 16-byte structure and copies it into the reference buffer. Until someone makes a subsequent setting, nothing more will happen, since every time the LA reviews the situation, it sees that the 16-byte buffer is accurately reflected in the current set of event channels. Only when an event channel is changed, or when an Acnet client sends down a different 16-byte structure, will anything change. There may be a pathological case of the 16-byte structure being written at the same time as an event channel is changed. The likelihood of this happening is small enough to be ignored, but in that unlikely case, the Acnet setting would win out.

LA logic design in more detail

Review the required logic for the LA. Upon initializing its static memory, it clears out both the 16-byte structure buffers and their reference copies. On each update scan, for each timer encountered during the scan that has a nonzero family word, it builds a list of event channel numbers by using the family word fields in the analog descriptors to advance to the next such channel until a zero family word is encountered, or until a channel is reached that is not of the event selection type. This can be detected because its analog control field is not of that type, which is 0x1B. (If a loop is defined that circles back on the delay channel, this will stop the scan.) Since 15 timer triggers is the maximum supported by Acnet, we can also limit this list to 15 entries. Also collect the related readings in this list of event channel numbers. Scan through the list, looking for values that are nonzero, which are valid event assignments. (A value of 0x0100, or 256, is the valid nonzero assignment for event 0x00.) For each such value, scan the array of event numbers found in the 16-byte structure. If it is there, do nothing more; if it is not there, note this difference for later use. After the scan is complete, start a new scan through the 16-byte structure, this time checking that each such event there is included in the values of the event channels. Note each case of not finding it.

At the completion of these two scans, one has a list of the events from the event channels that are not in the 16-byte structure, and one has another list of those events in the 16-byte structure that are not included in the event channels. If either list is non-empty, there is a mismatch for which action is required. Compare the 16-byte structure with its reference copy. If it matches, then the required action is to build a new 16-byte structure from the event channels and copy the result into the reference copy. If it does not match, then accept the 16-byte structure and make the event channels match it. If there are not enough event channels defined to hold the number of events included in the 16-byte structure, then return an error and do not make changes; instead, build a new 16-byte structure that reflects what the event channels say and copy it into the reference buffer. This allows the Acnet user to see what is actually there, even though he seems to want to set more events than the event channels can hold. The recovery from this situation is to define more event channels and link them together with new family words.

Given the two lists resulting from the scan and the required action of updating the event channels to conform to what the new 16-byte structure indicates, clear the event channels in the first list and then, for each event in the second list, find an empty event channel and set it to that event number.

The simpler action is to build the 16-byte structure from the event channels. Scan through all the event channels. For each one that is nonzero, append the low 8 bits to the (initially empty) 16-byte structure using the sneaky logic described above. (In this case, the two lists generated during the initial scans are not used.)

The support for Acnet reading the 16-byte structure is free, in that it is always there in the buffer; the LA does not have to do anything more to provide it. Whenever an Acnet user sends down a new 16-byte structure, assuming there are not too many events present, it is that very structure that it reads back, without its being reordered, say. Of course, if someone changes one of the event channels, a new 16-byte structure will be composed, so the Acnet user can see that latest set of event triggers.

Even more detailed logic

Assume `evtBuf` is an array of up to 15 events, plus a count, in the Acnet format, and `refBuf` is a reference copy of same, used for detecting when a change has been made by an

Acnet user that targets `evtBuf` via the listype used for addressing LA static memory.

If `evtBuf = refBuf`, then no Acnet user has made changes. Build list of timer delay channels, and for each one, a list of event channel values, plus a count of same. Let `nEvtC[j]` be the count of event channels, and let `evtChan[j][k]` be the array of events, where `k` ranges from 0 to `(nEvtC[j] - 1)`. (Use -1 for the event channel value when an event channel reading is zero.) Let `dlyChan[j]` be the delay channel for timer `j`. Armed with this info, populate `evtBuf` and copy the result to `refBuf`.

If `evtBuf <> refBuf`, then an Acnet setting has been made to attempt to change the trigger events for the timer. Build the same list of event channels as above, to get started. Scan each nonzero event from the event channels for a match in the `evtBuf`. Count all for which there is no match; *i.e.*, an event channel value is not found in the `evtBuf`. Then determine how many of the `evtBuf` events are not found in the event channels, so that they will need to be added to the event channel values. That number cannot exceed the number of event channels available. Passing this check, clear the event channels indicated, and find empty event channels to populate the events from the second list. Finally, copy `evtBuf` to `refBuf`.

Let function `GetEvtChans` build the arrays `dlyChan`, `nEvtC`, and `evtChan`. It is called by both paths of logic. Let `clrBuf` be the array of events that are to be cleared, and `nClr` be the size of this array.

Post-implementation notes

The local application `LOOPTRAM` was written along the lines described above. The Acnet-accessible buffers are located at `0x0100`, `0x0110`, ... `0x0170` offsets in the LA static memory block for the 8 timers. Every 15 Hz cycle, the program builds a set of arrays by scanning the set of timer channels described by its parameters. These arrays include the timer delay channel numbers and also the event channel numbers and values. For each of the up-to-8 timers, a determination is made whether the Acnet buffer has changed, compared to its reference copy. If it has, an Acnet user has modified it, so its contents are compared with the current set of event channel values. Then event channels are then cleared and/or set to effect the change. An effort is made to only make the minimal changes required to square the two representations of trigger events.

If there has been no change in the Acnet buffer, the event channels are checked for any changes that may have been made there compared to the most recent Acnet buffer. If there is a change, either because a user changed an event channel value, or because the LA just started up, a new Acnet buffer is built that reflects what is in the event channels.

In order to make the comparison between the event channel values and the Acnet buffer, a scheme of building the corresponding 256-event bit map from each is made. Then the two 256-bit maps are compared to see whether a different set of trigger events is specified. The Acnet buffer bit map only needs to be made if that buffer changes. The event buffer bit map is constructed each time. To compare two 256-bit maps, it comes down to comparing 8 longwords.

Under quiescent conditions, with no changes being made, the time for LA execution, including checking 8 timers, where 2 event channels are associated with each one, is about 0.5 ms. The part of this time required to build the arrays of event channels and values is about 0.2 ms. These times are for the 68040-based IRM. The PowerPC is likely to be much faster.