

NServer Logic Flow

RETDAT non-server processing

Robert Goodwin
Mon, Mar 10, 2008

This note describes the logic flow of the `NServer` function in the `ACReq` module. The function has evolved over a long time into what it is today. As a result, a review may be helpful.

`NServer` is called by `REQUEST` to initialize a `RETDAT` (or `GETS32`) data request. It loops over the number of request packets that comprise the request message. Each such 16-byte request packet specifies the 8-bit property index, the 24-bit device index, the 8-byte `SSDN`, the 2-byte length, and the 2-byte offset. Variables set ahead of the loop, in addition to the loop count, are the local `node#`, a ptr to the newly allocated (type #12) request block, a ptr to the first request packet, and a ptr to the first device request block (`DRB`) field within the request block. `NPTOTAL`, the count of the number of longwords needed for all internal ptrs, and `NBTOTAL`, the total #bytes of answer data, including the status word, are initially cleared. In addition, an error code response is cleared.

Within the loop, a check is made that the request packet `node#` field, which is the 2nd word of the 4-word `SSDN` structure, matches the local `node#`. (If it does not, then ignore this request packet and advance to the next one.)

Call `CYCLEADJ` to check for automatic switch to `Cycle` data access. This refers to a request for recent 15 Hz data that is archived for all analog channels into a circular buffer (the `CYCLE` table) to fulfill this kind of request. It only works for a periodic request in which the period is not 2 cycles (7.5 Hz) OR the number of bytes requested is less than 8. (This means `Cycle` data access is NOT indicated for a 7.5 Hz request that asks for 8 bytes or more.) The reason is that 7.5 Hz is reserved for cases of requests for two cycles worth of data delivered together. To request such data means that at least 8 bytes must be sought, in order to hold the number of sets of data (1 or 2), the reference `cycle#`, and the two words of data.

Next, the `listype#` is checked. If it is 0, used for analog channel readings, then check for the length in the range 2–66, the offset = 0, and the array case not specified in the 4th word of the `SSDN`, in which case it modifies the `listype#` from 0 to 88, since that is the `listype#` used to access integer `Cycle` data. Similarly, if the `listype#` is 40, used for accessing analog channel readings as engineering units floating point values, then check for the #bytes requested to be in the range 6–70, the offset = 0, and the array case not specified, in which case it modifies the `listype#` from 40 to 89, which is used for accessing floating point `Cycle` data. Whew!

After the call to `CYCLEADJ`, the `listype#` is used to obtain the ident length, via `LTTID` and `LTTIDT`. The ident length is then compared with the ident length derived from the low nibble of the 1st word of the `SSDN`. This nibble only has the values 1 or 2, which correspond to ident lengths of 4 or 6 bytes.

Next, the #bytes requested is checked for being > 0 and less than the maximum allowed size of a datagram, which is about 8K bytes or so.

If the ident length is 4, then the array case may be possible, in which the low byte of the 4th word of the `SSDN` is nonzero. The function `SIZEADJ` is called to check for this possibility. (Note that for ident length = 6, there is not room in the `SSDN` for this optional byte value.) What `SIZEADJ` does is check for that low byte being nonzero and less than the number of bytes requested, and if so, it divides the low byte into the number of bytes, requiring the result to have no remainder. If all this works out, then the number of bytes requested is set to

the low byte value, so that it becomes the number of bytes per ident, and the quotient is returned as the new number of idents; otherwise the number of idents will be 1.

The number of idents must be in the range 1–256. Then, from the listype id, via `LTTPSZ`, obtain the number of longwords in the internal ptrs structure, for each ident. The usual result is 1, but for pointer types in the range 32–49, the result is obtained from the `PSZTAB`. Accumulate the product of the result times the number of idents into `NPTOTAL`, which measures the total number of longwords in the internal ptrs array, yet to be allocated.

The offset specified in the request is checked, and if it is nonzero, `OFFSADJ` is called to deal with it, possibly returning a zero offset value to replace the nonzero one. All this has to do with the special use of offset that is enabled by a value of 1 or 2 in bits 7–4 of the 1st word of the `SSDN`. If the special use applies, the ident (in the `SSDN`) is modified accordingly. If the offset code = 1, then apply the offset to the ident in the `SSDN`, whether it is a 4-byte or a 6-byte ident. If the offset code = 2, and we have a 6-byte ident, then apply the offset to that ident, but shift the 16-bit unsigned offset value left by 8 bits first to build a 24-bit offset. The intended purpose for this shifted option is to allow access to memory over a 16 MB range, where the offset specifies where to start with 256-byte resolution.

Assuming no errors so far, obtain the read type#, via `LTRDRT`. Deposit it into the `RDI` field of the 8-byte `DRB` (Device Record Block). At this point, we have the `NID`, `NBY`, `RDI` and `PSZ` fields already set. The only remaining fields to be set are the `RDI` flags, in the hi byte of the `RDI` field, and the `PPI` field, which comes later.

If the property index is `ANALBL`, for the analog alarm block, check that the listype# is either 2 or 56. If it is neither, then declare an invalid listype error and exit.

Check for the analog or digital alarm block property cases, and if either, check that the listype# is either 2 or 56. If it is 2, then capture the `DI` (device index) via `ADEVXCAP`; if it is 56, then capture the `DI` via `CDEVXCAP`. (The latter applies only for comment alarm cases.) This capture of `DI` values allows an Acnet alarm message to identify the devices in alarm by `DI` rather than `EMC`, saving alarm handler `AEOLUS` from having to perform a data base lookup.

The next part of the logic looks for cases of accessing waveforms or “time-stamped” data, which refers to 7.5 Hz requests for delivering two consecutive 15 Hz cycles worth of data, along with a cycle number “time stamp” that refers to the first cycle.

The property index is checked for being `READNG` or `SETTNG`, and the listype# is checked for being either 0 (analog reading) or 1 (analog setting). In addition, the length (#bytes requested) and the offset are checked to both be even.

For an event-based request with nonzero delay, check that the offset = 0, the listype# = 0, the array case is not specified (low byte of 4th word of `SSDN` = 0). If all this applies, then set the `RDI` flags (in the hi byte of the `RDI` field-to-be) to `0x28`, which denotes `RDI` bits 13 and 11.

For the case of the length (#bytes requested) = 2, and the offset = 0, and a period specified slower than 7.5 Hz, set the `RDI` flags to `0x80`. For the case other than a periodic 7.5 Hz request, set the `RDI` flags to `0x20`. For the case of a periodic 7.5 Hz request, with length = 8 and offset = 0, set the `RDI` flags to `0xC0`; for a length ≥ 8 , set `RDI` flags = `0xE0`.

Now, backing up a bit, still working with a property index of `READNG` or `SETTNG`, if the listype# is 90, used for accessing floating point readings from `FDATA`, check that both the

length and offset are multiples of 4. Further, if the length = 4, and we have a periodic request slower than 7.5 Hz, then set the RDI flags to 0x90. If the length is greater than 4, and we have a 7.5 Hz periodic case, then set the RDI flags to 0xD0.

If the top 2 bits of the RDI flags are set, we have one of the “time stamped” cases, with RDI flags set to 0xC0, 0xE0, or 0xD0. Reevaluate the #idents (NID) field = $(NBY - 4) / 2 / item$, where *item* is the low byte of the 4th word of the SSDN and whose meaning is the #bytes of data saved for each of the two cycles. Replace the NID field with this result.

If the ms bit of the RDI flags is set, then it means we need “per cycle” work done, even though the request is not for 15 Hz replies, so add NID to NPTOTAL to allow for an extra longword in the internal ptrs structure. Further, if bit 14 of the RDI flags is set, set the AVGFLAG; otherwise, zero it. This AVGFLAG, if set, just means that averaging logic will be used somewhere in the request.

If the property index specifies a digital alarm block (DGALBL), and the listype# = 2, then modify the property index for the next step to act as if it were ANALBL.

From the property index, derive an offset to a post-processing routine to be called each time after building the answer data for this device. Install this offset into the PPI field of the DRB. (An exception is made for the basic status (BSTATS) property in which the listype# = 0; in this case, there is no post-processing needed. There is also a special check for listype# = 56, in which a special post-processing routine is used.)

Finally, ensure that the #bytes requested (length) is even, by adding 1 if necessary, then add that value, plus 2 to allow for the status word, to the NBTOTAL variable, which sums up the total #bytes of data to be returned in each reply to this request.

The result of this routine is to fill the DRB entry for each device in the request. The internal ptrs structure will comprise NPTOTAL longwords, but they are initialized by DOPTRS.

DOPTRS

After REQUEST calls NServer to prepare the DRB structures for each device, it calls DOPTRS to prepare the internal ptrs structures. Using NPTOTAL, an internal ptrs block (type #14) is allocated and initialized. Certain fields are significant:

PBLKFTDC	#cycles since last reply update
PBLKSUMC	#cycles of data in averaging accumulation
PBLKBEAM	#beam cycles in accumulation
PBKHZSF	flags to be set if any KHz sampling indicated
PBSPARE	(n.u.)
PBLKAVGF	copy of averaging enable flag AVGFLAG
ANXTRA	count of #idents used for short ANALBLS
KHZSFLG	flags to be used for KHz sampling
KHZSPAR	(n.u.)

A loop over the number of request packets is the main organizational logic. For each packet:

Check that the 2nd word of the SSDN matches the local node#; else, go to next packet.

If the listype# = 2 (ADATA nominal field used for analog alarm blocks), and the #bytes

requested < 6, then if the #idents (NID) > ANXTRA, update ANXTRA with NID. All this has to do with handling requests for short analog alarm blocks, as a request for the alarm flags only means we must access 6 bytes from the nominal field to get the tolerance and alarm flags.

If NID > 1, then build an array of consecutive 4-byte idents that will be passed to the ptr type routine to generate the internal ptr structures. If NID = 1, then simply pass the single ident that is in the 2nd and 3rd words of the SSDN. (Usually, NID = 1. It is only > 1 when the ident length is 4 bytes and the low byte of the 4th word of the SSDN is nonzero.)

Call PREQDGEN to generate the array of internal ptr structures for this request packet.

Perform special logic using the RDI flags. For the entire RDI word, considering that the flags are in the high byte only, check bit 13. If it is set, it implies a waveform case, in which case this flag bit is cleared, after which it checks bit 11. If it is set, then it appends a zero longword after the internal ptr just planted by PREQDGEN and calls WKHZADJ; otherwise, it calls WAVEADJ.

Consider WKHZADJ first. Check whether the channel# in the 3rd word of the SSDN is one for which 1KHz sampling applies, as indicated by a suitable CINFO table entry. If it is, then overwrite the ADATA reading ptr (that has already been placed in the internal ptrs structure) with the ptr to the IRM digitizer memory address word (that is updated for every word digitized and stored by the 1KHz digitizer hardware) followed by the base address of the 64KHz circular hardware buffer, overwriting the zero longword that was just appended before this routine was called. Also place the 6-bit hardware channel# (the same as the low 6 bits of the ADATA channel#) into the low 6 bits of this base address that is always a multiple of 64K bytes and thus has its low 16 bits clear. Then set bit #0 in KHZSFLG to indicate that 1KHz sampling is used in this request.

Continuing on with the WKHZADJ logic, in the case that the channel# was *not* one for which 1KHz sampling applies, it might be suitable for HRM 10KHz sampling, so check for that case. If it is, set bit 12 of the RDI flags, and overwrite the ADATA reading ptr with a ptr to the HRM hardware block counter followed by the base address of the 2MB hardware circular buffer. Again, insert the hardware 6-bit channel# into the low bits of this base address and set bit #1 of KHZSFLG to indicate that 10KHz sampling is used in this request. If the channel# was not suitable for 10KHz sampling, then clear bit 11 of the RDI flags to indicate that we failed to find that the expected 1KHz or 10KHz sampling applies, after all. (Bit 13 is already clear.)

Now consider WAVEADJ. It handles a different kind of waveform hardware, in which the points of the waveform are stored consecutively in memory. Examples of this are the Swift, Quick, and Quicker digitizers. To support such cases, we will need to replace the internal ptr already stored (that points to the ADATA reading field) with a suitable ptr to the waveform. So, the logic loops over the number of idents NID, calling CHKWAVE to get the appropriate waveform address, replacing the read type# with 9 in the Quick digitizer case, since Quick digitizer memory can only be accessed by words, not longwords.

As for CHKWAVE, which is called by WAVEADJ, its logic checks whether the channel# in the SSDN matches a Swift digitizer entry (in the CINFO table), and if it is, it calls SWIFTMEM to get the address. (Note that this covers the Quicker case, too.) If that check was unsuccessful, it checks for matching a Quick digitizer entry, and if it is, it calls QUICKMEM to get the appropriate waveform address. In order for the caller to detect the Quick digitizer case, as noted in the previous paragraph, the low bit of the returned Quick digitizer address is set.

Returning to `DOPTRS` flow, having taken care of the waveform cases, if the offset parameter in the request is nonzero, and if the read type# (low byte of `RDI`) is 1 or 9, then call `OFFSADJB`, which applies the offset parameter of the request to all `NID` nonzero internal ptrs. This can allow access to part of a waveform by using the offset as a simple byte offset.

Next, if the `RDI` flag bit 15 is set, then we have per cycle logic needed, and if bit 14 is set call `NBYADJ`; else call `AVGADJ`. The function `NBYADJ` establishes the number of bytes to copy each cycle, which is needed for the 7.5 Hz support of two consecutive 15 Hz cycles of data. This #bytes (`NBY`) is placed in the high word of the extra longword appended to each internal ptr. The low word of that longword will be used as an offset for determining where the copied data will be placed in the answers area. The function `AVGADJ` distributes each internal ptr so that it is followed by a zero longword, used for the averaging accumulation.

This is the end of the loop in `DOPTRS` over each request packet. After the loop exits, copy `KHZSFLG` into `PBKHZSF` to denote whether any KHz sampling is to be done. Finally, a pointer to the associated request block is placed into the `RQPBLKPT` field of the internal ptrs block.

`RDI` flags meaning:

<i>Bit#</i>	<i>Meaning</i>
15	per cycle work needed
14	periodic 7.5 Hz request used to access two cycles of 15 Hz data
13	waveform case, either waveform segment or event+delay KHz sampling
12	floating point case; also used to indicate 10 KHz sampling rather than 1 KHz
11	KHz sampling case
10	n.u.
9	n.u.
8	n.u.

For the following, property = `READNG` or `SETTNG`, listype# = 0 or 1, length and offset even:

For event case, nonzero delay, with offset = 0, listype# = 0, non-array case, set bits 13, 11.

For length = 2, offset = 0, period slower than 7.5 Hz, set bit 15.

For length \neq 2 or offset \neq 0, and either non-periodic or periodic 7.5 Hz case, set bit 13.

For periodic 7.5 Hz, length = 8, offset = 0, set bits 15, 14.

For periodic 7.5 Hz, length \geq 8, set bits 15, 14, 13.

For the following, property = `READNG` or `SETTNG`, listype = 90, length and offset multiples of 4:

For length = 4, period slower than 7.5 Hz, set bits 15, 12.

For length > 4, period = 7.5 Hz, set bits 15, 14, 12.

Note that after the call to `PREQDGEN` to create the internal ptr structure, `RDI` flag 13 is cleared. It is only needed until the internal ptr structure has been modified appropriately.

The waveform sampling from a KHz buffer needs a second longword for the internal ptr structure. This is allowed for, and `PSZ` is bumped, at the same time that `RDI` bit 11 is set. That second longword is cleared in `DOPTRS` when it sees that `RDI` bit 11 is set.

ACUPDATE

To update a RETDAT request, filling the reply buffer with answers, loop over each DRB in the request and call the appropriate read-type routines. What is appropriate is given by the RDI field, the upper byte of which contains flags and the lower byte is the read-type#.

If RDI bit 15 is set, we are using per cycle processing. If bit 14 is set, we have the periodic 7.5 Hz case, in which two cycles of data are assembled into the reply buffer. If that is the case, the answers have already been collected, as a result of per cycle processing, so that no read-type routine need be invoked.

If RDI bit 15 is set, and bit 14 is clear, check bit 12, used as a floating point flag in this case. If bit 12 is clear, we are using integer averaging; if it is set, we are using floating point averaging. Either special read type routine AVGANSW is called, or AVGANSF is called.

If RDI bit 15 is clear, there is no special per cycle processing used. If bit 11 is set, we have the KHz waveform sampling case, and special read type routine SAMP1K, or SAMP10K is called.

If RDI bit 15 is clear, and bit 11 is clear, we have the usual case, and READTYPE is called to return the ptr to the read type routine to be called, based on the low byte of RDI.

After the chosen read type routine is called, the PPI field is checked. If it is nonzero, then special post-processing is needed, where the appropriate routine is called in a loop over the #idents (NID).

Next, the ptr to the answers buffer is advanced by the size of the answers just built. It is normally $NID * NBY$, but for the case of RDI flag bit 15 set and bit 14 set, indicating the special 7.5 Hz case where two cycles of data are returned together, the advancement is $2 * NBY + 4$, which allows for two sets of data and a 4-byte header. Then the ptr is advanced, and if it should be odd, it is advanced by one more byte. (Device reply data is always word-aligned.)

The ptr to the internal ptrs array is also advanced, based on the PSZ field that indicates how many longwords are in the structure for each ident. So advance by $NID * PSZ * 4$. A special case is checked, however, for $PSZ = 1$, in which case, if the RDI per cycle flag bit 15 is set, the advancement is $NID * 8$, since two longwords are needed for performing the averaging.

After the loop over the array of DRB structures, special logic is used for updating the GETS32 reply header, if it applies, and the low 16 bits of the current 15 Hz cycle number is saved in the reply message block RPYCYCL field. This is used by the logic for GETS32 ping replies.

Finally, the reply message block is ready. First, NETXCHK is called to see whether previous answers are already queued to a destination node different from that being targeted by the reply block just updated, and if so, flush any such reply messages to the network. This is the mechanism used to combine multiple reply messages into a single datagram, when possible. Then NETQUEUE is called to queue this reply block, and NETQFLG is set to indicate this reply block is "busy," in order to ensure it does not get freed before it has been used, in case of a cancel of this request.

ACUPDCHK

If the update task, when following the linked list of active request blocks, encounters a type #12 request block, it calls ACUPDCHK to handle it. This routine only handles non-server requests, which it verifies by checking that the SMSGID (server message id) field is zero. In the

internal ptrs (type #14) block, it checks PBLKAVGF, and if it is nonzero, it calls AVGACCUM to perform any required per cycle logic. Then, ReplyDue is called to test whether a reply is due on the current cycle. If it is, then ACUPDATE is called to build the reply message; otherwise, CHKPING is called to support GETS32 protocol ping replies.

AVGACCUM

This routine handles all per cycle logic for a RETDAT request. If bit 0 of PBLKAVGF (in the internal ptrs block) is set, then beam-preferred averaging logic is needed. Call BEAMSTAT to get the beam status for the current cycle. Compare this with PBLKBEAM, which is set to one when the averaging accumulation consists of beam cycles; otherwise, it is zero. If this is the first beam cycle, set PBLKBEAM and clear PBLKSUMC, the count of cycles included in the accumulation. If this is a non-beam cycle, but we have already accumulated at least one beam cycle, ignore this cycle.

If either bit 1 of PBLKAVGF is set, or we need to add the data from this cycle into the averaging accumulation as determined above, then perform special processing as follows; else, exit.

Loop over all DRB structures. Examine the RDI flags. If bit 15 is set, and if bit 14 is clear, do the averaging logic. If we need to use the data from this cycle in the averaging, as determined above, then check bit 12. If it is clear, do integer averaging; else, do floating point averaging. If PBLKSUMC is zero, we are starting a new accumulation, so place the current cycle's data into the accumulation; otherwise, add the data from this cycle into the accumulation. The two longword internal ptr structure for these cases uses the first for the ptr to the data value; it uses the second for the accumulation. Loop over NID such internal ptr structures.

If RDI flag bit 15 is set, and bit 14 is also set, then DATACAP is called to perform the special logic that supports the capture of two cycles of data for delivery at 7.5 Hz. The format of the internal ptr is the data ptr in the first longword, and the #bytes to copy in the hi word of the second longword, with the current offset into the data area in the lo word.

If RDI flag bit 15 is clear, there is no per cycle logic needed, so advance to the next DRB entry by advancing past $PSZ * NID$ longwords in the internal ptrs array.

Advance the answers ptr by $NID * NBY$, but for the data capture (7.5 Hz) case, advance the answers ptr by $NBY * 2 + 4$. In either case, the advancement must be an even #bytes.

Variety of reading properties obtainable via the same SSDN, by specifying FTD, length, offset:

Assume for all cases that the SSDN specifies listype #0. It may or may not have an offset flag set; it may or may not have the array case enabled, using a nonzero lo byte of the 4th word.

Ordinary data pool reading:
length=2, offset=0

Array of 2-byte readings from consecutive channels:
length=2*n, offset=0, size=2

If the offset flag is set, one gets the same data starting at the indicated channel+offset. This allows control over the channel# specified in the SSDN.

Note that if the FTD specifies a periodic rate slower than 7.5 Hz, the readings are averaged,

with beam cycle taking precedence.

Continuous 15 Hz data delivering two consecutive cycles of data at 7.5 Hz:

FTD = 7.5 Hz, length ≥ 8

Waveform access, assuming that channel# specified is found in CINFO table:

FTD not 7.5 Hz periodic, length > 2 , or offset > 0 .

Note that both Swift/Quick/Quicker digitizers as well as 1KHz/10KHz are supported.

Cycle data, or data measured on most recent 15 Hz cycles:

FTD not 7.5 Hz periodic, length ≥ 4 , and length < 68

Note that both integer and raw floating point cycle cases are handled. The last word is always the cycle# of the present cycle.

Data sampled from KHz circular buffer on event (or state) + delay (GETS32):

Data Event String specifies event (or state) + delay, length=2

Offset applied to waveform:

After adjusting for waveform cases, read type# = 1, 9.

If in the SSDN, the offset flag is set, the effective channel# is altered by the given offset.

If in the SSDN, the array case is specified, a request for a multiple of the number of bytes specified in the low byte of the 4th word of the SSDN will result in, say, an array of 2-byte readings starting at that channel, or at the one modified by the given offset.

All post-processing routines should be using a single longword internal ptr. There should be no post-processing used for cases of data averaging nor for 7.5 Hz cases.

REQUEST

While reviewing all this logic for handling a non-server RETDAT/GETS32 request, it may be useful to sketch the REQUEST routine that invokes NSERVER. Here is that sketch:

Ensure that the number of request packets is ≤ 600 , an arbitrary limit to be sure we are not working with crazy parameters. Then check that the message size is sufficient to cover the indicated number of request packets, again, another "idiot check."

The routine PREVIEW is called to size up the request, paying attention to the 2nd word in each request packet SSDN to determine how many devices are local and how many come from other node(s). This is important data to decide whether the request is to be treated as a server request or a non-server request. If the request is sent via multicast, or if the request includes only local devices, then the request is treated as a non-server request. If the request is *not* sent via multicast, and if the request includes at least one non-local device, then it is treated as a server request. (A request sent via multicast with no local devices included is ignored.)

Following the non-server path, a UDP node# found in an SSDN is replaced by the local native node#. This is done because the local data request support only looks for native node#.

Then DOREQ is called to allocate and initialize the type #12 request support block. Then NSERVER is called. More checks are made for a crazy request. The limits imposed are 5400 for

`NPTOTAL`, the number of longwords in the entire internal ptr structure. And `NBTOTAL` must be not too large to fit in the maximum datagram size, which is about 8K bytes and change.