

Acnet Alarm Reporting

Robert Goodwin

Wed, Oct 21, 2009

Acnet alarm messages are delivered to the Acnet alarm handler by local application `AERS`, which is installed in each front end. Alarm messages are passed to `AERS` via a message queue as a result of alarm scanning by the `Alarms` task in the system code. Alarms are reformatted and queued for timely delivery to Acnet, taking care not to overdo it, since Acnet has to deal with hundreds of front ends. This note describes the logic used by `AERS` to accomplish this delivery effort.

History

An alarm message protocol was defined for Acnet by D. Bogert, M. Gormley, and F. J. Nagy, according to Acnet Design Note 39.3, a 1983 document that was updated in 1990. From that document came the design of `AERS` that all Classic front ends use to shepherd Acnet alarms to the central services alarm handler, referred to as `AEOLUS` here, since that is the destination task name used for Acnet communications. `AERS` was originally written in 1991 and was last modified in 2005.

Overall logic flow

When `AERS` is first enabled, usually following a front end reboot, it sends a Front End Boot (`FEBT`) message to `AEOLUS`, on `centra.fnal.gov`. An acknowledgment signifies to the front end that `AEOLUS` knows it is "in business." As alarm messages occur within the front end, they are sent to `AEOLUS` in an Event Report Message (`ERM`), including one or more Event Report Packet (`ERP`) messages, each of which announces alarming news about one device. An acknowledgment to the `ERM` is expected from `AEOLUS`. If it is not received, `AERS` repeats the attempt to deliver the `ERM`, for fear the alarm communication was unsuccessful. Subsequent `ERM`'s are sent to `AEOLUS` as alarm conditions warrant, but they are throttled in order to not overrun the ability of `AEOLUS` to handle them. If there should come a time when an `ERM` fails to be acknowledged even after retries, `AERS` concludes that `AEOLUS` has "gone away," so it reverts to an initial mode of occasionally sending a `FEBT` message until it receives an acknowledgment. Then it performs an internal alarms reset, causing all currently Bad conditions to be reported again by the underlying system `Alarms` task. An Acnet user may want to issue a Big Clear to all front ends of a given project, such as Linac. That area of the user's alarm screen is then cleared, and on reception of the `BIGC` message by `AERS`, it causes any local terminal hardware to be cleared, and an internal alarms reset done by the `Alarms` task just before the next cycle's alarm scan, resulting in resending all Bad alarm conditions.

AERS parameter layout

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
<code>ENABLE</code>	2	Usual LA enable Bit#
<code>ALMNODE</code>	2	Target alarm node# for <code>ERM</code> alarm messages
<code>ERMDLY</code>	2	Timeout delay in cycles between successive <code>ERM</code> 's
<code>REPLYDLY</code>	2	Timeout delay in cycles on reply to <code>ERM</code>
<code>FEBTDLY</code>	2	Timeout delay in cycles on reply to <code>FEBT</code>
<code>ALMDLY</code>	2	Delay after first alarm before sending <code>ERM</code>
<code>NCACHE</code>	C 2	Diagnostic #alarm messages in queue, Chan#
<code>SUBSYSNO</code>	2	Subsystem# for <code>EMC</code> 's in lo 3 bits
<code>CLEARCRT</code>	2	Code for RS232 local terminal hardware, if any
<code>BIGCFWD</code>	2	Node# for forwarding <code>BIGC</code> messages

Alarms task

The front end `Alarms` task performs the basic function of alarm scanning all device readings at 15 Hz, soon after the local data pool is refreshed from the hardware. When it detects alarm conditions, it arranges to queue Classic alarm messages to the network to a multicast destination.

This allows any interested Classic protocol handler to monitor the alarms from an entire project, since all front ends in one project multicast alarm messages to the same destination address. After being sent to the network, the `QMonitor` task notices the associated alarm message blocks, and before it frees them, it passes the alarm info via a local message queue called `AERS` (appropriately enough) that the `AERS LA` monitors to learn about alarm conditions.

Classic task alarm message handling

Each front end includes a `Classic` task that listens to the Classic protocol UDP port. This allows the front end to monitor project-wide alarm messages, and if enabled via certain special Bits, it allocates a message block, fills it with the alarm info from the message received, and queues it to the network, taking care to mark the message block "used" so it will not be actually multicast to the network again! The `QMonitor` task then builds ascii alarm messages for output to the local RS232 serial port. This scheme allows building an alarms log for an entire project, which can be monitored in real time and/or saved for possible later review. See the 1992 note, *Classic Protocol*, and the 1998 note, *Classic Protocol for Clients*, for more on Classic alarm message formats.

ERP format

An ERM message sent by `AERS` to `AEOLUS` consists of one word that includes a type code and a count of ERP packets to follow. This word has the #ERP's in the hi byte and the EMC type code (1) in the lo byte. Followed this word is an array of ERP's, each formatted as follows:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
<code>pktSize</code>	2	packet size in bytes, lo byte only, SRP flag in bit 8 (mask=0x0100).
<code>aStatus</code>	2	Acnet alarm status word
<code>emc</code>	8	Event Message Code (EMC)
<code>reading1Lo</code>	2	reading value, or SOS
<code>reading1Hi</code>	2	ms word, n.u.
<code>reading2Lo</code>	2	reading value for SRP case
<code>reading2Hi</code>	2	ms word, n.u.

There are actually two formats for an EMC. The traditional one allows the front end to identify the device in a unique way using its own method. An ERM message uses this one if the ERM type code is 1. In order for `AEOLUS` to utilize this identification, it must perform a database lookup to get the device index, the principal key in the Acnet database. Here is the front end-designed EMC format:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
<code>subs</code>	1	subsystem byte
<code>node</code>	1	node# byte
<code>aTyp</code>	1	alarm type#, where 0 = analog channel, 2 = comment
<code>trnk</code>	1	trunk# byte
<code>indx</code>	2	analog channel#
<code>spar</code>	2	not used, always zero

An example for `L:GR5MID` that illustrates this format is (in hex) `4088/0009/0102/0000`. The first bit (mask=0x4000) indicates this format, rather than an earlier one. The Acnet node# word, as we use it in the front end, is `0x0988`. This alarm comes from analog channel `0x0102`. As this format specifies a front end node and channel, it is unique across all of Acnet, as required.

A newer format for an EMC allows specifying an Acnet database device index, rather than using an arbitrary front end-designed scheme. An ERM uses this form (for all its ERP's) if the ERM typecode is specified as 14, rather than 1. This approach is attractive for `AEOLUS`, because it saves it from having to do the lookup in the database to find the device index. But this implies that the

front end knows the Acnet device index for each device that is alarming within the ERM. A scheme within the front end designed to deal with this uses a new nonvolatile memory system table called ADEVX. Each four-byte entry, indexed by an analog channel#, merely holds the device index, if nonzero. Acnet device indexes are taken from requests for reading/setting access to alarm block properties and saved in ADEVX for AERS alarm reporting. The SSDN in such messages holds the analog channel#, and the Acnet device index is copied into ADEVX using that index.

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
nodTrk	2	Acnet node/trunk. For Acnet node# 0x0988, this is 0x8809.
spareW	2	n.u.
diLo	2	lo word of device index
diHi	2	hi word

Once AERS determines the set of ERP's that it will collect into an ERM, it checks whether it has valid ADEVX entries for every one. If so, it uses the new EMC format; else, it uses the original EMC format. More detail on this approach is found in the 2004 note, *Acnet Alarms via DI*.

AERS input

The AERS LA monitors a message queue that is written by the QMonitor task when freeing a Classic alarm message block. The format of these queued 16-byte messages is as follows:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
aSpar	1	n.u.
aType	1	alarm message type, 0=analog, 1=binary, 2=comment
aFlag	2	alarm flags word from ADATA/BALRM/CDATA entry
aIndx	2	channel#/bit#/comment#
aRead	2	reading word
aTime	8	time of alarm in usual BCD format, yr-mo/da-hr/mn-sc/cy-ms

AERS cache

The AERS LA caches alarms to be delivered to Acnet, because it must throttle them in order to avoid overrunning AEOLUS. Here is the entry format used for its 512-element internal cache:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
as	2	alarm status word used by Acnet
ix	2	chan/bit/comment index#
rd	2	reading word
sos	2	sos value used for Acnet digital alarm case

Alarm flags formats

The front end uses an alarm *flags* word. This is taken from the note, *Alarms Task Logic Flow*.

<i>Bit#</i>	<i>Name</i>	<i>Meaning</i>
15	ACT	1=active, so that alarm checking is enabled for this channel/bit
14	BIN	1=pattern, the 16-bit channel reading is a combined binary status word
14	NOM	1=nominal state for individual binary bit case
13	INH	1=inhibit beam on the next cycle if this channel/bit is Bad
12	FLT	1=raw floating point comparison checking to be used for this channel.
11	BST	1=scan for alarms only on beam cycles, according to Bit 0x009F
10	BYP	1=bypass this channel/bit on the next cycle
9	ACC	1=used to avoid hysteresis logic for min/max case
8	BAD	1=Bad, 0=Good

7	LOG	1=Log inhibit, no messages for this channel/bit, but do trip counting
6	INV	1=data invalid, used for missing SRM replies.
5	LIM	1=use min/ max logic, 0=use nominal/ tolerance logic
4	<i>n.u.</i>	
3-0	TRY	4-bit tries_needed specification, where 0=1 try, 1=2 tries, ..., 15=16 tries.

Acnet uses a different alarm *status* word, with the following fields:

Bit#	Name	Meaning
15	DE	1=event display enabled
14	LE	1=event logging enabled
13	EV	1=event (good/bad), 0=exception (comment)
12	HI	1=analog reading is hi
11	LO	1=analog reading is lo
10-8	K	0=nom/ tol, 1=nom/ %tol, 2=min/ max
7	AD	0=analog, 1=digital
6-5	Q	0=byte, 1=word, 2=longword
4	<i>n.u.</i>	
3	AI	0=inhibit beam if Bad
2	AB	1=allowed to set AI bit
1	GB	0=Good, 1=Bad
0	BP	0=alarm monitoring bypassed

This word appears in the analog/ digital alarm block properties, and it is used when reporting alarms for AEOLUS. The front end translates between these two status/ flags words as needed. For example, AERS reads alarm *flags* from a message queue, but it records alarm *status* in the cache. When a setting is made to an alarm block, the Acnet alarm status word must be translated into the appropriate alarm flags word. The front end does not retain alarm blocks, so it must extract alarm block fields to set the appropriate fields in the ADATA table, for example. When a request is made for an alarm block, the alarm *flags* word must be translated into the Acnet alarm *status* word.

Translation between the two formats is not perfect. The BST flag bit, used for indicating that the alarm scan for a given device is to be done only on beam cycles, has no Acnet equivalent. When needed, therefore, this bit must be set locally, via a little console, or a "Page G" equivalent. The page application EDAD, normally found on page A, allows for this access.

In more detail, consider building the Acnet *status* word in AERS, given the alarms *flags* word. This is done when an entry is placed in the cache. Start with a status word value of 0x0080 or 0x2000 for the binary or comment case, respectively. For the analog case, if the PAT bit is set in the flags word, initialize the status word to 0x4000, temporarily; otherwise, set it to 0x0000.

If the BAD flag bit is set, set the GB status bit. Always set the BP bit, since by definition, the alarm is not bypassed. Set the AB status bit always, as we always have the possibility of inhibiting beam for any device. If the INH flags bit is *not* set, set the AI bit to indicate that beam is inhibited when this device is Bad.

Later on, when the cached alarm is extracted, a check is made for setting the AD bit in the status word. If the temporary bit (mask=0x4000) is set, then set the AD bit to indicate a digital alarm, and set bit 8 in the pktsize field to indicate the SRP case. This has to do with the support for digital alarm scanning based upon a "combined binary status word" that is assembled from BYTE bits but stored in the ADATA table that is normally used for analog data. The PAT flag denotes such digital data and tells the Alarms task that it should use pattern matching, not numeric comparison.

For more on this, see the 1999 note, *SRP Support for Acnet*.

Alarm blocks

Acnet organizes alarm-related info in a 20-byte analog/digital alarm block data structure. The first 12 bytes are meaningful here:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
aStat	2	Acnet alarm status word
nomLo	2	nominal value, lo order word
nomHi	2	nominal value, hi order word
tolLo	2	tolerance, lo word
tolHi	2	tolerance, hi word
tNeeded	1	tries_needed
tNow	1	tries_now

For the analog min/max case, the nominal is the min value, and the tolerance is the max value. For the digital case, the nominal is the good pattern, the tolerance is the mask.

All analog/binary cases only support 16-bit data, so only the low order words are used. The one exception is raw floating point data, in which all 4 bytes are used.

Translation must be made between this alarm block format and the internal table entries used by the front end. For reference, here is the format for the 16-byte ADATA table entry:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
READNG	2	Analog channel reading, or combined binary status reading
SETTNG	2	Analog channel setting
NOMNAL	2	Analog channel nominal value, or combined binary nominal pattern
TOLRNC	2	Analog channel tolerance value, or combined binary mask
AFLAGS	2	Alarm flags
ACOUNT	2	Alarm trip count
MOTORC	2	Analog channel motor step counter, or combined binary SOS
SPARE	2	--

Raw floating point data is “born” as a 32-bit floating point value; it has no 16-bit integer equivalent. Such data is normally the result of local application processing. An example might be a vacuum gauge reading that covers many decades of range, so that 16-bit scaling is insufficient. Here is the format for the 16-byte FDATA table entry, used for raw floating point data:

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
FREADNG	4	Analog channel raw floating point reading
FSETTNG	4	Analog channel raw floating point setting
FNOMNAL	4	Analog channel raw floating point nominal value
FTOLRNC	4	Analog channel raw floating point tolerance value

For these cases, the same AFLAGS and ACOUNT fields are referenced from the ADATA entry for the same channel. Any analog channel may be either 16-bit integer or 32-bit raw floating point. The FLT bit in the AFLAGS field denotes which it is. (A channel can only be one or the other, not both.)

If such data needs to be set, the LA should monitor the FSETTNG field in the FDATA entry, then perform any required duties.

AERS diagnostic

As an aid to track the traffic between AERS in a front end and AEOLUS, a data stream can be defined called AERSLOG. Records are written into this data stream whenever a message passes between AERS and AEOLUS. This has been useful in the past when tracking down problems. For more detail on this, see section AERSLOG in the 2006 note, *Diagnostic Data Streams*.

AERS timing

As stated above, AERS must be careful to avoid overrunning AEOLUS, as 15 Hz alarm scanning has the potential of producing a very large number of alarms. Use of a cache as a “staging area” for alarms is a help toward this end. The timing is based on some parameters of AERS, as described above. The usual parameter values are:

<i>Parameter</i>	<i>Value</i>	<i>Meaning</i>
ERMDLY	0040	Timeout delay in cycles between successive ERM's
REPLYDLY	0400	Timeout delay in cycles on reply to ERM
FEBTDLY	0400	Timeout delay in cycles on reply to FEBT
ALMDLY	0004	Delay after first alarm before sending ERM

With these delay parameters, a new alarm message will wait 4 cycles, about 0.25 sec, before sending an ERM containing as many devices as are then ready to be reported. This allows for alarms to accumulate for a few cycles, before the ERM is sent. Note that this has no effect on when the beam will be inhibited, if the device has that option enabled. The front end asserts beam inhibit control on the same cycle—after the beam pulse, of course—on which the alarm is detected. This makes it possible to inhibit beam on the very next cycle.

After sending an ERM, 64 cycles (0x0040), or about 4 seconds, must pass before another ERM is sent. This defines the level of throttling referenced above.

AERS expects an acknowledgment reply from AEOLUS after it sends an ERM. This time is 0x0400 cycles, about a minute or so. (The minimum time for this parameter is 500, or about 30 seconds.) Failing to receive a reply in that time, AERS resends the ERM. Two retries are permitted. If AERS instead receives an error-laden reply, it means the ERM communications did not reach AEOLUS, so it arranges to retry after only 5 more seconds.

If all this fails, then AERS has no recourse but to give up on AEOLUS, at least for the time being. To that end, it reverts to the initial logic that awaits a successful acknowledgment of a FEBT message, hoping that AEOLUS will return to normal operation soon. With the parameter given above, it will keep sending a FEBT message about every minute, keeping this up until it receives the expected acknowledgment from AEOLUS that will signal all is well again. It will cause an internal alarms reset in its local node to prompt all current Bad alarm conditions to be reported again.

Note that during a period of inability to communicate with AEOLUS, the front end alarm scanning is unaffected. It continues to look for alarm conditions, inhibit beam if appropriate, reporting Classic alarms via multicast, and generating an alarm log listing, if enabled.

Conclusion

Local application AERS serves to interface alarm message from a front end to the Acnet alarm handler AEOLUS. It delivers alarm message in the standard Acnet protocol designed for the purpose, and it takes care not to overrun AEOLUS, as hundreds of front ends need its attention.