

# Alarms Task Logic Flow

Robert Goodwin

Thu, Oct 15, 2009

## *Introduction*

The `Alarms` task runs every 15 Hz cycle, after the local data pool has been refreshed with the latest readings and replies to all active data requests due on that cycle have been delivered. Its main objective is to check all analog channels for being out of limits. It also checks for binary bits being in the wrong state. Whenever a change in Good/Bad state is seen, it queues an alarm message to the network via multicast. This message is a Classic protocol message, so that it can be seen by any interested node that listens to the Classic port. Besides transmitting the message, it is also passed internally via a message queue to `AERS`, the local application that is used to shepherd alarm messages to the Acnet alarm handler `AEOLUS`. A key requirement for Linac, since 1982, is the ability to inhibit beam on the next cycle, if any of a selected set of channels is in the Bad state. This implies that a complete alarm scan must be done on every cycle.

Three types of alarm scans are done. The analog alarm scan covers numeric 16-bit readings, 32-bit raw floating point readings, and combined binary 16-bit status words. The binary alarm scan checks for individual status bits being in the wrong state. (This bit-based binary alarm scan does not cause `AERS` to pass messages to `AEOLUS`.) Finally, the comment alarms are those announcing news of interest, only two of which have ever been defined. These are the `VME SYSTEM RESET` message, emitted after a front end reboots, and the `VME ALARMS RESET` message, which is sent to denote that an alarms reset action has been performed. (This message is normally disabled lest too many front ends in a project emit it all at once.) An alarms reset action is normally triggered by reception of a Big Clear message from Acnet.

Note that option switch #6, if set at reboot time, inhibits alarm scans. This “stand-aside” switch also skips the auto-restore of settings at reset time, and it disables Data Access Table processing, so that no new data is read, and no local applications are run. It is used during system configuration.

## *Overall logic flow*

The `Alarms` task is structured as a forever loop that awaits task event 0 to spur it into action. The sequence of task operations is as follows: Every cycle, `Update` sends event 0 to `QMonitor`, which sends event 3 to `DaTime`, which sends event 0 to `Alarms`.

If task event 0, perform alarm scan:

Call `ALSCAN` perform the complete alarm scan.

If task event 5 set, `ACTBUILD`: rebuild block of currently active channels.

If task event 6 set, `ACTBUILD`: rebuild block of currently active bits.

If task event 7 set, `ACTBUILD`: rebuild block of currently active comments.

Send task event 3 to wake up `App1` task, so the current page application can run.

## *Blocks of currently active channels/bits/comments*

There is an allocated block of the currently active entries for analog channels, binary bits, and comments. This makes the alarm scan more efficient, because it does not have to examine inactive `ADATA`, `BALRM`, `CDATA` table entries. This is especially significant for PowerPC nodes, in which each access to nonvolatile memory, wherein these three tables reside, costs about 1  $\mu$ s.

Such blocks are rebuilt when a setting is made to an alarm flags field that affects the active flag, or

every 9 minutes (termed a back-up build), even when no changes are made.

Each type 0x0A block has a 32-byte header, with the following fields starting at offset 8 bytes:

Field	Size	Meaning
ACTIVT	2	Time to build table, in $\mu$ s
ACTIVN	2	Current #entries in table
ACTIVMN	2	Min value for ACTIVN
ACTIVMX	2	Max value for ACTIVN
ACTIVC	4	Diagnostic count of non-backup builds
ACTIVCB	4	Diagnostic count of backup builds
ACTIVDT	8	Date-time of last non-backup build
ACTIVA	n*2	Active indices array (channels/bits/comments)

This info is accessed by the ALRM page application to summarize the alarm scan load in one node, or in a set of nodes contained in a file DATANxxx. See note, *Alarm Scanning Info*.

### **ALSCAN logic flow**

If Bit 0x00A1 is set, call ALRESET to reset all alarms, then call ARSTMSG to output the alarms reset comment message "VME ALARMS RESET", if enabled, and reset the Bit.

If Bit 0x00A0 is set, call TRIPCLR to clear analog and binary trip counts, and reset the Bit.

For each 15 Hz cycle, after 1 second has passed since system reboot, test beam status Bit 0x009F, setting the BEAM byte variable accordingly. (If Bit 0x009F is 0, it means beam.)

If no active comments block, create one, set task event 7 internally. If comments block, call CSCAN, then check cycle counter for multiple of 8192 cycles (about 9 min at 15 Hz) plus 20. If so, set internally task events 7, 8. This will cause a rebuild of the active comments block.

If no active analog channels block, create one, set task event 5 internally. If analog channels block, call ASCAN, then check cycle counter for multiple of 8192 plus 22. If so, set internally task events 5, 8 to cause a rebuild of the active analog channels block.

If no active binary bits block, create one, set task event 6 internally. If binary bits block, call BSCAN, then check cycle counter for multiple of 8192 plus 24. If so, set internally task events 6, 8 to cause a rebuild of the active binary bits block.

If any analog channel or binary bit has its beam inhibit flag set and was found to be in the Bad state during the alarm scan, set control line (relay) to inhibit beam on the next cycle; otherwise, clear that control line.

### **TRIPCLR logic flow**

This function merely zeros the 12-bit trip counts for all channels, bits, and comments, no matter whether they are currently active. It also sets the time-of-day at offset 64 in the nonvolatile PAGEM table. This time is shown on Page B to indicate when the trip counts were last cleared. To provoke a system to clear its alarm trip counts, set Bit 0x00A0.

### **ALRESET logic flow**

This function clears the good/bad bits to good. A recent modification also clears the tries\_now nibble for each channel/bit that is currently in the Bad state. This is to ensure that the very first scan following an alarms reset that finds a channel, say, out of tolerance, will set it immediately to the Bad state, even when the tries\_needed specification is greater than 1. This is

especially important for a channel that is meant to inhibit beam when it is Bad.

### *Alarm-related system tables*

Three system tables are accessed by the Alarms task when performing its function. The **ADATA** table houses an array of entries for each analog channel that consist of integer readings, nominal/tolerance values, the alarm flags, and a trip count.

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
READNG	2	Reading value
SETTNG	2	Setting value
NOMNAL	2	Nominal value, or min value for min/max cases
TOLRNC	2	Tolerance value, or max value for min/max cases
AFLAGS	2	Alarm flags, detailed below
ACOUNT	2	Alarm trip count
MOTORC	2	Motor count, used for SOS value for digital pattern case
SPARE	2	--

The **BALRM** table holds the similar alarm-related fields for the individual binary bit alarm scan. Each 4-byte entry holds the required data for a single Bit of binary data.

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
BFLAGS	2	Alarm flags, detailed below
BCOUNT	2	Alarm trip count

The **CDATA** table holds alarm-related info for each comment, even though only 2 are defined.

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
CFLAGS	2	Alarm flags, detailed below
CCOUNT	2	Alarm trip count
CTEXT	24	Comment text
CDEVX	4	Acnet device index, used for more efficient reporting to AEOLUS

The **FDATA** table holds data for "raw floating point" channels. Such channels have only a 32-bit floating point representation, just as they were "born." There is no 16-bit integer equivalent.

<i>Field</i>	<i>Size</i>	<i>Meaning</i>
FREADNG	4	Raw floating point reading
FSETTNG	4	Raw floating point setting
FNOMNAL	4	Raw floating point nominal, or min, value
FTOLRNC	4	Raw floating point tolerance, or max, value

For raw floating point channels, the **ADATA** fields **AFLAGS** and **ACOUNT** are referenced. Bit **FLT** of **AFLAGS** is set to denote the channel is a raw floating point channel.

### *Alarm flags word bit assignments*

Bit#	Name	Meaning
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.

The tries\_needed specification requires a corresponding tries\_now count field. This is found in the upper 4 bits of the 16-bit alarm count field. The low 12 bits house the trip count, incremented every time the BAD bit changes state; hence, the number of trips is half the size of the trip count.

### **ASCAN logic flow**

Scan all active analog channels by referencing the active channels block.

Collect specs for system tables ADATA, FDATA. The latter is needed for channels that have the floating point (FLT) flag bit set, so that floating point arithmetic is used with the 32-bit floating point readings, nominals and tolerance values in the FDATA entry.

Scan through the list of channel numbers in the active channels block. For each one, perform the following logic:

Look up the ADATA entry for this channel, and check the alarm flags field. If the ACT bit is set, scan this channel for alarm conditions; else, move on to the next channel.

If BYP flag bit set, clear BYP, clear ACT, and if BAD bit set, post Good message. (When an alarm flags setting is processed that would clear the ACT bit, the BYP bit is set internally, so as to be able to post a Good message if the channel was Bad when its scanning was deactivated/bypassed.)

If BST bit set, meaning the scan should only be done on Beam cycles, check BEAM. If BEAM indicates a non-beam cycle, skip to the end to check for beam inhibit.

If INV bit set, it means that the data reading is invalid, likely because of failure to receive an expected SRM reply on this cycle. Clear both the INV bit and the BAD bit, clear the tries\_now nibble, skip to end. The idea is to assume the data is Good if it is invalid. (Alarming on SRM failures is done another way, based upon a combined binary status word that contains status of SRMs.)

There are two cases. Either the channel is currently Good or currently Bad. First the Good case:

If BIN bit set, it means we must do pattern matching logic, as this reading is a digital pattern. Calculate the SOS (status of statuses) as the exclusive OR of the reading with the nominal value, ANDed with the tolerance mask. This value has "1" bits for all status bits that are Bad. Compare the SOS value with the motor count word MOTORC for this channel. (Note that such a channel whose reading is a digital status word cannot have motor control.) If it matches, clear the tries\_needed counter and skip to the end. (The idea is that we want to post a message any time the SOS value changes, implying that one or more bits changed its Good/Bad state.) If it does not match, count tries\_now, and if not enough yet, skip to end. If tries\_now reaches tries\_needed, update the MOTORC with the SOS. If it is nonzero, it means we have new Bad-ness to report, so clear the Bad bit so that ASNEW will properly post a new Bad message. Then call ASNEW.

If `BIN` bit *not* set, we have the usual case of analog comparison checking for this channel's latest reading. There are two cases, integer and floating point.

If `FLT` bit *not* set, it means we use the usual 16-bit integer comparison logic on the reading value found in the `ADATA` table. If the `LIM` bit is *not* set, use nominal/tolerance logic. This logic includes a hysteresis characteristic, in which a channel that is currently Good will become Bad if its reading is found more than the tolerance value from the nominal value. But a channel that is currently Bad will only become Good if its reading is found within half a tolerance of the nominal value. This is done to avoid the alarm message chatter that might occur if the reading lies very nearly one tolerance from the nominal value. Note that this hysteresis characteristic applies for both integer and floating point alarm checking. For the min/max cases, there is no hysteresis component.

If the reading is within the tolerance range of the nominal value, it is still Good, so reset the `tries_needed` count, and skip to the end. If the reading is outside the tolerance range, decrement the `tries_now` count, and if it meets the requirement, post a Bad message via `ASNEW`.

If `LIM` bit set, it means we use min/max logic, in which the nominal field holds the min value and the tolerance field holds the max value. The comparison logic presumes signed values for all. If the comparison shows a reading value  $\geq$  lower and  $\leq$  upper, then we still have the Good state, so clear the `tries_now` count and skip to the end. But if we have the Bad state, in which the reading is outside the min/max range, decrement the `tries_now` count to see whether we have satisfied the `tries_needed` requirement. If we have, call `ASNEW` to post a Bad message.

If `FLT` bit set, it means we must use floating point comparison logic on the reading value in the `FDATA` table. If the `LIM` bit is set, use min/max logic, else use nominal/tolerance logic, obtaining the latter values from the `FDATA` entry.

Whenever a change in the alarm state for a channel is detected, `ASNEW` is called. It toggles the `BAD` flag bit, increments the trip count, taking care not to overflow the 12-bit field, and if the `LOG` bit is *not* set, it calls `ANEW`. Function `ANEW` builds a Classic alarm message block, then calls `QMSG` to queue it to the network. If 16 messages have been queued this cycle, call `NEXTTASK` to not hog the CPU.

At the end of the `ASCAN` code loop, the alarm flags are checked for both the `BAD` and `INH` bits set, and the `LOG` bit *not* set. If this is the case, increment `AINHIBIT`. Later on, this will help determine whether to inhibit beam on the next cycle.

For the second case of an analog channel currently in the Bad state, the logic is very similar to that above, but we are watching for a change to the Good state; otherwise, no message is needed. For the nominal/tolerance checking, the special variation of using half a tolerance applies.

### ***BSCAN*** logic flow

Function `BSCAN` performs alarm checking on individual binary bits. Its logic is much simpler than the analog channel case. The `BALRM` table has one entry for each Bit, including the 2-byte alarm flags and the 2-byte trip count. At the end of checking each Bit, just as for the analog case, a check for the current Bit having both `BAD` and `INH` bits set, and `LOG` bit *not* set, is made. If so, increment `BINHIBIT`, which will be used to help decide whether beam is to be inhibited on the next cycle. The idea is that a bit or channel that does not emit an alarm message, because its `LOG` bit is set, must not be allowed to inhibit beam. It is unkind to inhibit beam but not explain why!

When a change of state occurs for a binary bit, `BSNEW` is called. This function toggles the `BAD` bit, increments the trip count, and if the `LOG` bit is *not* set, it calls `BNEW`, which then builds a Classic alarm message block and calls `QMSG` to queue it to the network.

There is a special Bit 0x00A7 that, when set, inhibits all alarm scanning. It is not expected that this option be used often, if ever. But if this Bit is set, the only alarm scanning done is for this Bit, which allows for generating a binary alarm message when it is set, to serve as a reminder that alarm scanning (for everything else) is inhibited. In Linac front ends, this Bit is enabled for alarm scanning, and it is also set to inhibit beam when it is set.

### ***CSCAN logic flow***

Function **CSCAN** is even simpler. Each 32-byte **CDATA** table entry houses a flags word, a trip count word, and 24 characters of text. If the **ACT** flag bit is set, it tests the **BAD** bit. If it is set, it calls **CSNEW**. Function **CSNEW** toggles the **BAD** bit (to zero), increments the trip count, and if the **LOG** bit is not set, it calls **CNEW** to build an alarm message block and calls **QMSG** to queue it to the network.

### ***Alarm message multicast delivery***

The destination node# for the Classic multicast messages is specified in the 4<sup>th</sup> word of the **PAGEM** table. It is of the form 0x09F<sub>x</sub>, where **x** ranges from 0–E. The corresponding multicast IP addresses are in the range 0xEF8002F<sub>x</sub>, based at 239.128.2.240. The corresponding multicast ethernet addresses are 0x01005E0002F<sub>x</sub>.

If a front end is listening to one of these multicast addresses that is used to target alarm messages, the **Classic** task, on receiving such a message, allocates an alarm message block to hold it and queues it to the network in the usual way, but it sets the “used” flag to denote that it should not be actually transmitted. This allows the **QMonitor** task to notice such messages, and if specified via the Bits 0x00A<sub>x</sub> mentioned above, transmit them to its serial port. This is used by Linac node0616 to generate an alarms log listing for a special monitor in the **MCR**. We also use this scheme to support logging these alarm messages in test nodes via terminal emulators so the alarms log can be captured long term. As these alarm messages are produced at 15 Hz resolution, it is possible to analyze at that level the behavior of hardware, even across an entire project, when a trip occurs.

The above discussion of alarm scanning results in Classic alarm messages that are queued to the network as often as 15 Hz, if necessary. After each message is sent, one of the functions of the **QMonitor** Task is to notice that it has been sent, and before freeing the allocated alarm message block, it checks to see whether Bit 0x00A3 is set. If so, it arranges to format an ascii alarm message to send to the local serial port. Bit 0x00A2 is also checked to see whether an ascii message should be displayed on the bottom line of the little console. If Bit 0x00A4 is set, all alarms are output in this way, serial and/or bottom line; otherwise, only binary alarms are listed.

### ***Alarm messages for Acnet***

The Acnet alarm handler is called **AEOLUS**. It receives alarm messages from all front ends and delivers them to many console alarm windows for operator viewing. For support of this delivery, the front ends described herein use the local application **AERS**, which collects local alarm messages, sent by **QMonitor** via an internal message queue as it is about to free the Classic alarm message block, and sends them to **AEOLUS**. But it takes care not to speak too often, since all front ends needs access to **AEOLUS**. It sends up to 22 alarm messages in one Acnet message, after which it refrains from sending more for about 4 seconds. It also waits up to 4 cycles before sending a suddenly created message, so as to allow for others that may occur in the next few cycles to be delivered together in the same message. Note that this rather leisurely pace does not imply a delay in inhibiting beam, as each front end operates its own control line (relay) that in hardware can prevent beam from being accelerated on the next cycle.

### ***Efficiency of alarm scanning***

Each front end fills a data pool every 15 Hz cycle with the latest readings of all connected

devices. The alarm scan is based upon the readings in this data pool; hence, it can be done efficiently. The alarm scanning logic does not depend upon how the data was collected; it only deals with the data pool contents. From the description found above, it may seem to be complex, but it is actually simple. The complexity arises from so many options that are allowed to influence the logic needed for each channel. Entire alarm scans in one node take well under 1 ms.

### *Tries\_needed commentary*

A key Linac controls requirement was to be able to inhibit beam on the next 15 Hz cycle if any of a set of signals was found to be Bad. The Alarms task supports this, fundamentally. But if a device is set for `tries_needed > 1`, that device, when it goes Bad, cannot meet the key requirement. When electing the option, a user should be careful of the consequences, since the `tries_needed` number of cycles will have to be Bad before beam can be inhibited. For a device that is *not* set to inhibit beam when Bad, however, there is no such concern, because the worst that can happen is that the alarm message may be posted slightly later, yet still well within human response time.

### *Conclusion*

This note describes the logic associated with alarm scanning supported by the code in Linac-style front ends, whether based upon the 68K or PowerPC platform. All devices are read into a data pool every 15 Hz cycle, providing the basis for efficient alarm scanning logic. Especially important for Linac is the support for inhibiting beam on the next cycle when any of a selected set of devices is found to be in the Bad state. This feature has been in place for more than 25 years.

When changes in Good/Bad states occur, the front end multicasts alarm messages via the Classic protocol, allowing any node to monitor them, either via ethernet or via a serial port. It also delivers alarm messages to AEOLUS using the Acnet protocol. Alarm block settings affect alarm properties in the usual way via the Acnet SETDAT/SETS32 protocols.