

Alarms and Events in BACnet

David Fisher

3-Jun-2021



TUTORIAL

Contents

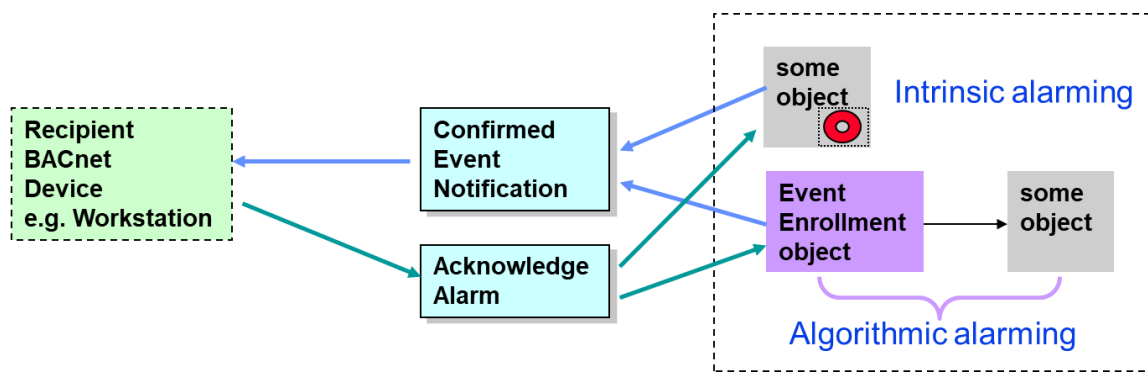
Introduction.....	3
State Diagrams	4
BACnet Alarm States	5
Intrinsic Alarming	6
Algorithmic Alarming.....	7
Standard Algorithms.....	8
CHANGE_OF_STATE.....	8
OUT_OF_RANGE.....	9
COMMAND_FAILURE.....	9
FLOATING_LIMIT	10
CHANGE_OF_BITSTRING	11
CHANGE_OF_VALUE.....	12
CHANGE_OF_LIFE_SAFETY	13
Other Standard Algorithms	13
Event Enrollment Object	14
Event Detection Properties	14
Current Event State	15
Notification Classes	16
Step-by-Step Example	18
Conclusion	23
Legal Stuff.....	23
Contact the Author.....	23

Alarms and Events in BACnet

David Fisher

Introduction

The concept of alarming in BACnet is complex and includes several optional elements. In many building automation systems (BAS) there may be individual "points" that have a 0 value when something is "normal" and a 1 value when something is "offnormal". Many people call these "alarm points" as they can be examined at any time in order to determine if there are any "alarms". This is not what BACnet means by alarms. In BACnet there are several ways that logic can be used to detect an offnormal condition and to subsequently report that condition to some other BACnet device(s). In this tutorial we are going to talk about these separate but related ideas of *alarm detection* and *alarm notification*.



This diagram shows a simplified “big picture” of the flow of information for typical alarms and events in a BACnet system. There are three fundamental types of alarm and event reporting mechanisms in BACnet: Change-Of-Value, Intrinsic Alarming and Algorithmic Alarming. Change-Of-Value is normally not used for “alarm” handling, so we are going to focus on the other two types in this tutorial.

Alarms or Events originate from BACnet objects, meaning that every alarm or event must be implicitly attached to an object. Some kinds of objects, typically BACnet standard objects like Analog Inputs and Binary Outputs, can have built-in alarm detection features. These are called *Intrinsic Alarming Objects*. Although any kind of object, including non-standard ones, can have intrinsic alarming features, standard objects that claim to provide intrinsic alarming must provide certain properties to control alarm reporting behavior, and make those parameters network-visible. In the special case of standard objects that claim to provide intrinsic alarming, these properties become required and not optional.

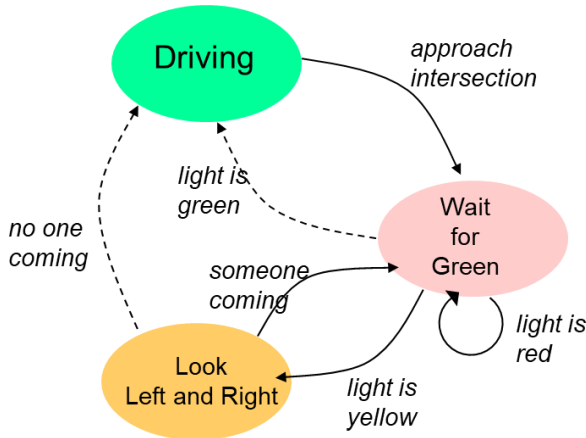
BACnet also allows alarm detection to be implemented outside of the object. In this case, an Event Enrollment object is used to monitor potential alarm or event conditions in one or more other objects. By applying one of several standard monitoring algorithms, the Event Enrollment object can detect different types of changes as if those objects had intrinsic alarming built-in.

In either case, once an alarm or event has been *detected*, one or more other BACnet devices may be notified of the occurrence of the alarm or event using either `ConfirmedEventNotification` or `UnconfirmedEventNotification` services. Typically, a workstation or supervisory controller is the recipient of these notification messages.

In some cases, it is also important for the initiating device to receive a confirmation that a human (or very responsible program) has not only received the notification, but that it is now OK for the initiating device to stop worrying about it. This handshaking uses the AcknowledgeAlarm confirmed service.

State Diagrams

Before we can talk about alarms, it is important to introduce the idea of a *state diagram*. This kind of picture allows us to describe the various "states" that constitute alarm detection logic. Let's use a familiar real-world example of the process of driving a car in the U.S.

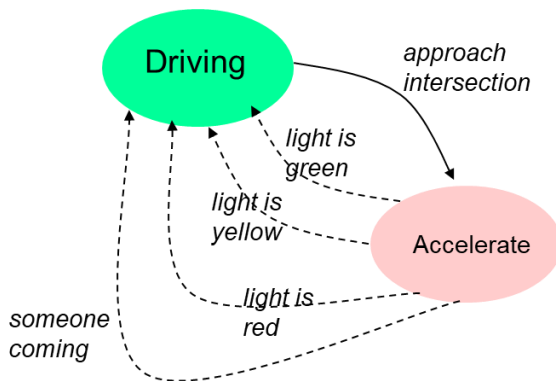


Most of us are familiar with driving a car. When we are in that "state" if we approach an intersection, we enter a more cautious state where we (are supposed to) look at the traffic lights before entering the intersection. If the light is green then it's OK to move through the intersection and continue driving. If the light is red, we stop and wait for the light to turn green.

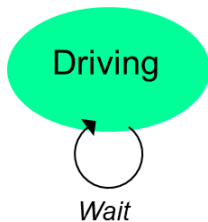
If the light is yellow then we enter a new state where we are supposed to look right and left before proceeding. If someone is coming into the intersection ahead of us, we just stop and wait for the green light again. If no one is coming we can cautiously proceed through the intersection and resume driving.

The diagram captures these rules and shows us how to behave under various conditions. For most places this is a robust set of rules. In some larger cities we are all familiar with "aggressive drivers" who seem to follow different rules:

The diagram captures these rules and shows us how to behave under various conditions. For most places this is a robust set of rules. In some larger cities we are all familiar with "aggressive drivers" who seem to follow different rules:

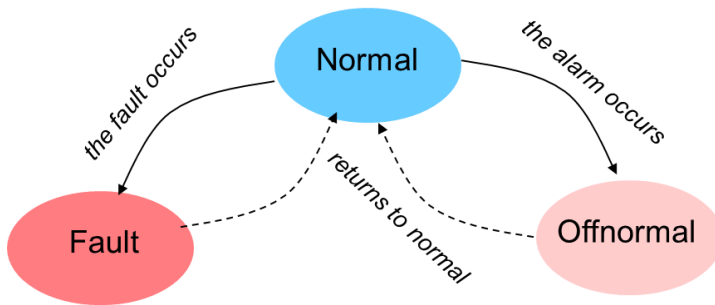


Sometimes the time of day dictates alternative rules also, such as driving home at "rush hour":

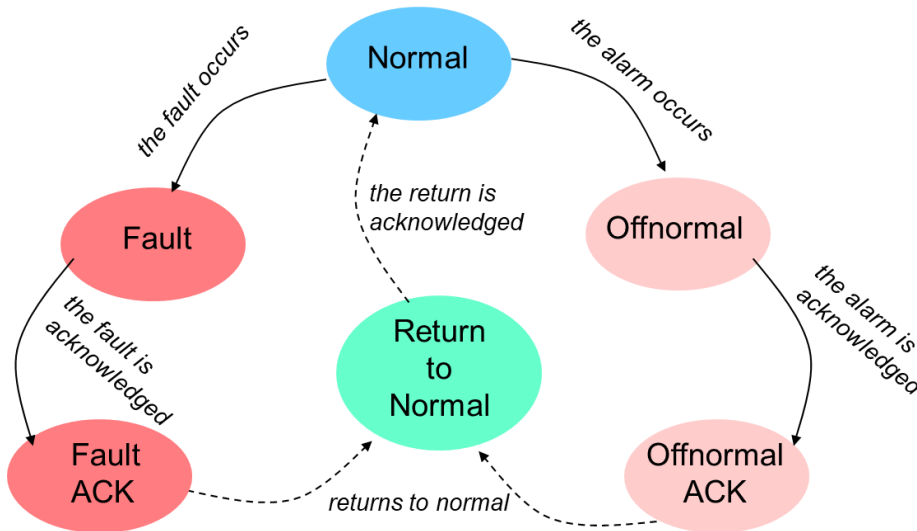


State diagrams are handy for explaining how BACnet alarming is supposed to work.

BACnet Alarm States



In the big picture, a critical concept in BACnet alarming is that an alarm source can only be in one of three fundamental states at any time: NORMAL, OFFNORMAL or FAULT. The transition from one state to another can trigger the notification process. As a further complication, each transition can independently require human acknowledgement. In effect, with the acknowledgement criteria, as many as six states may be needed: NORMAL, OFFNORMAL, OFFNORMAL ACKNOWLEDGED, FAULT, FAULT ACKNOWLEDGED, and RETURN TO NORMAL.



BACnet standard objects that support intrinsic alarming must provide several properties that control and indicate the state of alarm detection. The Event_Enable property is a three bit bitstring with one bit for each of these three principal states. If the bit is one then that corresponding transition generates a notification. In the simple case when no human acknowledgement is required, there are three basic states. BACnet requires the state to transition through Normal.

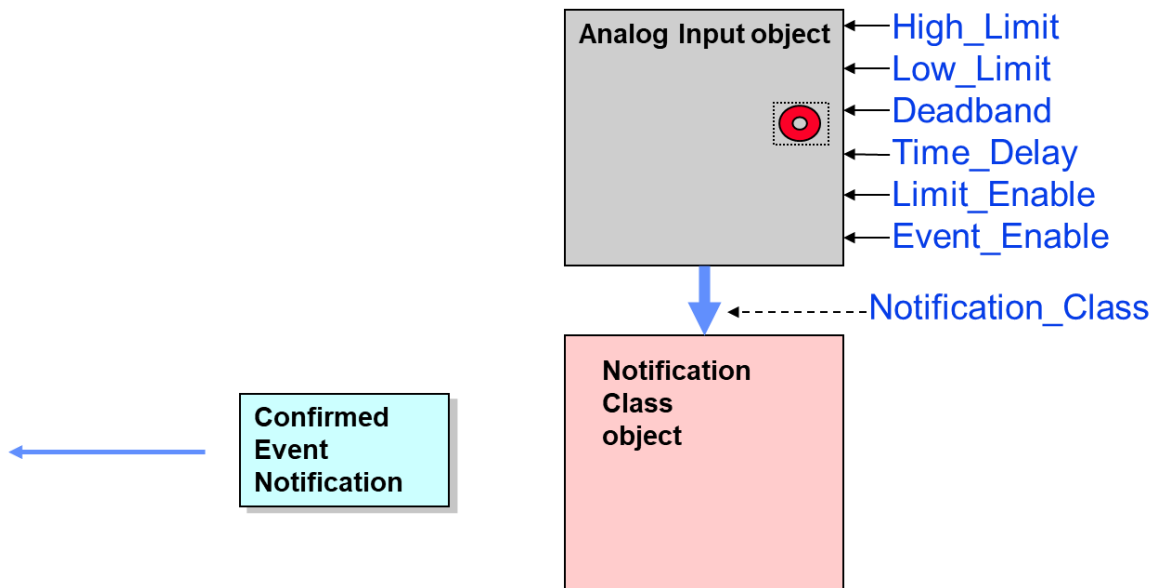
When human acknowledgement is required for one or more of these transitions, the state diagram can become much more complex.

The intermediate states occur while waiting for human acknowledgement to arrive via an AcknowledgeAlarm service. Note that return to normal is also a transition that can be configured to require acknowledgement.

Each object that can generate event notifications keeps track of the state and the acknowledgements using two standard properties. The Event_State property is an enumerated value that indicates the state: NORMAL, FAULT and OFFNORMAL. There are several special

event states including non-standard ones, but any event state that is not NORMAL and not FAULT is considered to be OFFNORMAL for state-tracking purposes. The Acked_Transitions property is a three bit bitstring that indicates for each state whether the transition to that state has been acknowledged or not.

Intrinsic Alarming



Standard objects that implement built-in alarm and event management according to the standard use intrinsic alarming. The Analog Input, Output and Value, Binary Input, Output and Value, Multi-state Input, Output and Value, Loop and various other value objects all specify intrinsic alarm handling behavior. When these objects implement intrinsic alarming, a number of their otherwise optional properties become required.

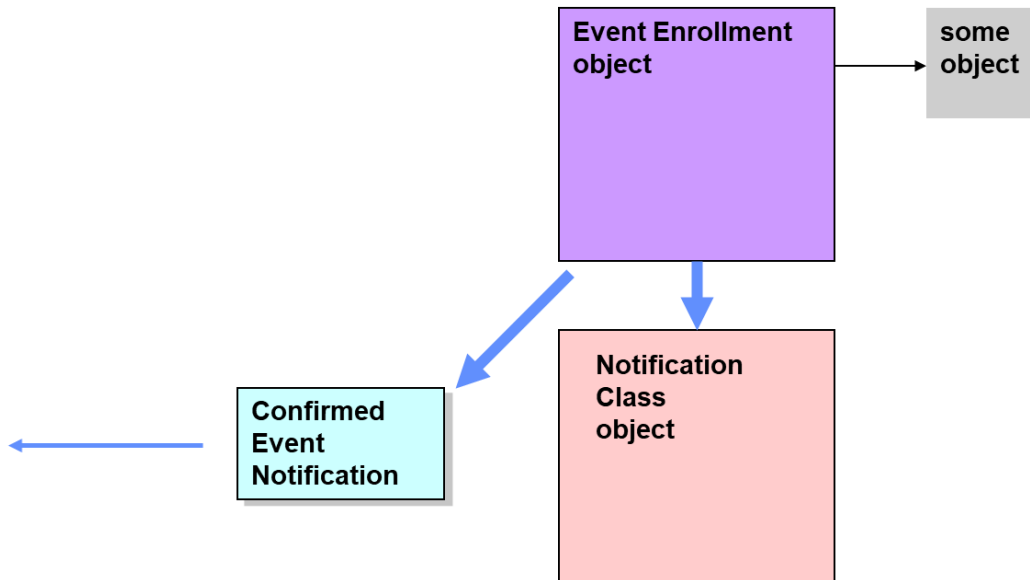
Consider the case of the Analog Input object shown above. The High_Limit, Low_Limit and Deadband properties specify a range of acceptable operating values for Present_Value. Once outside these limits the object is *offnormal* and may generate alarm notifications. The Limit_Enable property determines which limits (high, low or both) can generate notifications. The Event_Enable property determines which types of events (TO-NORMAL, TO-OFFNORMAL, TO-FAULT) should be reported. The Notify_Type property (not shown) serves to determine whether the object is an alarm or event generating object.

There is no difference in BACnet between "alarms" and "events". By convention, alarms are intended for humans and events are intended for monitoring programs but otherwise they behave exactly the same.

The Notification_Class property indirectly refers to a Notification Class object in the same device by virtue of the class number. The Notification Class object may be shared among several alarm-generating objects. Its purpose is to determine which BACnet devices should be recipients of the alarm notification. Every alarm-generating object has a notification class. Why do it this way? The reason is that typically there are many more *sources of alarms* than there are unique *destinations* for alarm notifications. By organizing destinations into notification class objects, we simplify the setup and configuration of alarm-generating objects, as well as their ongoing

maintenance. Each individual device that can be a destination for alarm notifications is called a *recipient*.

Algorithmic Alarming



Some kinds of BACnet devices, although they contain standard objects, may not implement intrinsic alarming, or they may implement it in a different way. Even if intrinsic alarming is implemented, there are some kinds of alarms that may require a somewhat different algorithm than the standard ones. In any of these cases, a BACnet device may choose to implement *algorithmic alarming*. In this scheme, an Event Enrollment object is used to monitor a property of another object using one of many standard monitoring algorithms, to determine whether an offnormal condition exists or not.

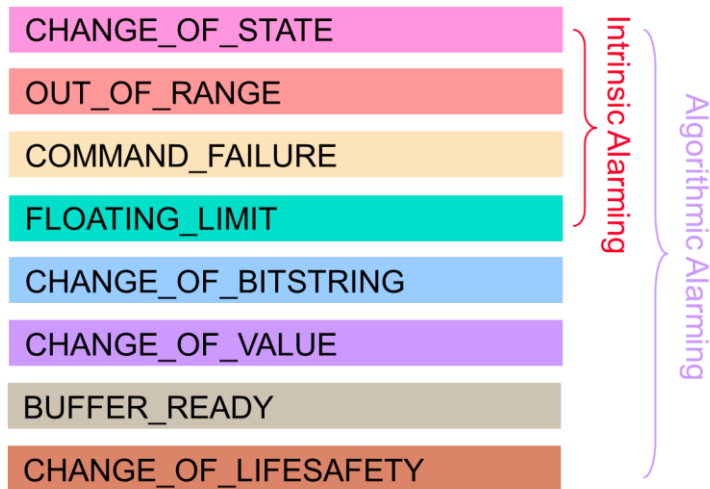
Each Event Enrollment object refers to only one property of one object. Unlike the case of intrinsic alarming, algorithmic alarming may be applied to any property of any object. Moreover, multiple Event Enrollment objects can apply different algorithms to the same property of an object.

As with intrinsic alarming, the Event Enrollment object use a Notification_Class to implicitly refer to a Notification Class object.

There are 23 standard algorithms that are specified for BACnet alarming, only four of which are used by the intrinsic alarming features in standard objects. It is also possible for devices to implement Event Enrollment objects that use non-standard algorithms, but the parameters that are used by the algorithm may have a more limited network visibility.

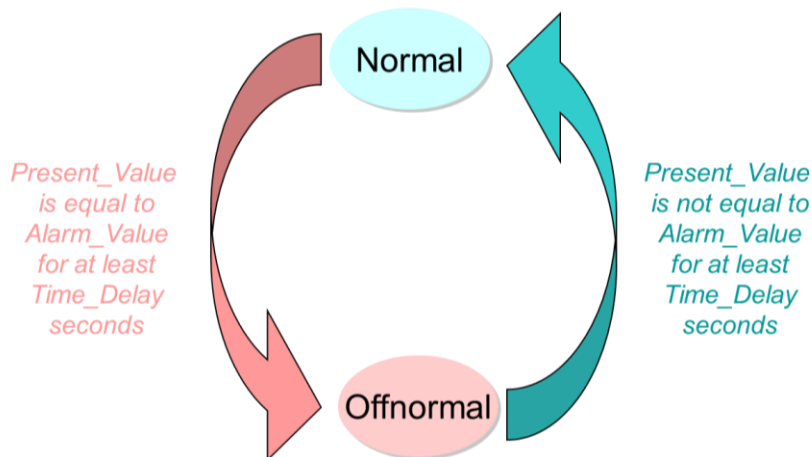
We'll examine each of the standard algorithms in some detail in the following section.

Standard Algorithms



CHANGE_OF_STATE

applies to Binary Input, Binary Value, Multi-state Input, Multi-state Value, Event Enrollment



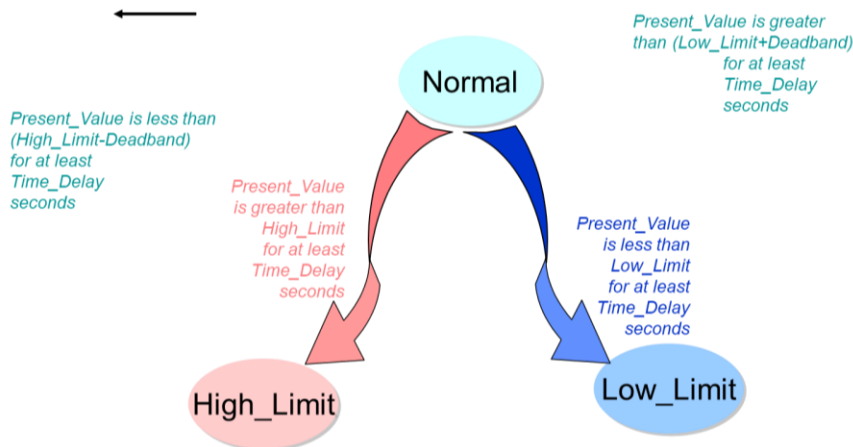
The Change-Of-State algorithm applies to Binary Input, Binary Value, Multi-state Input, Multi-state Value and Event Enrollment objects. The diagram shows the Binary Input object as an example.

The concept is that the object's Present_Value is continuously compared with an Alarm_Value. If the Present_Value remains equal to the Alarm_Value for Time_Delay seconds, then the object changes from the *normal state* to the *offnormal state*. Once the two values remain not equal for the same period, then the object returns to the normal state.

In the case of Multi-state Inputs and Values, there can be several values that are Alarm_Values.

OUT_OF_RANGE

applies to Analog Input, Analog Output, Analog Value, Event Enrollment

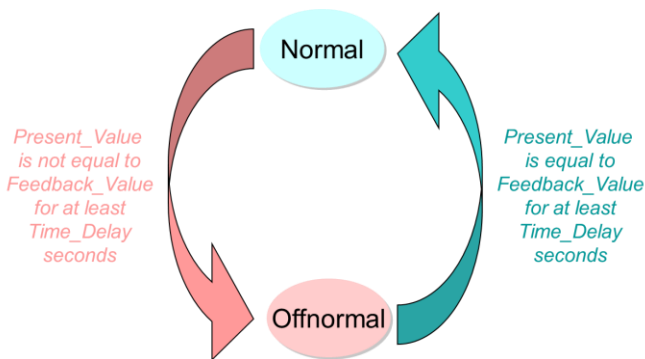


The *Out-Of-Range* algorithm applies to Analog Input, Analog Output, Analog Value, and Event Enrollment objects. The diagram shows the Analog Input object as an example.

The concept is that the object’s *Present_Value* is continuously compared with *High_Limit* and *Low_Limit*. If the *Present_Value* remains less than the *Low_Limit*, or greater than the *High_Limit*, for *Time_Delay* seconds, then the object changes from the *normal state* to the *high limit* or *low limit state*. Once the value remains greater than the $(Low_Limit+Deadband)$ or less than the $(High_Limit-Deadband)$ for the same period, then the object returns to the normal state.

COMMAND_FAILURE

applies to Binary Output, Multi-state Output, Event Enrollment

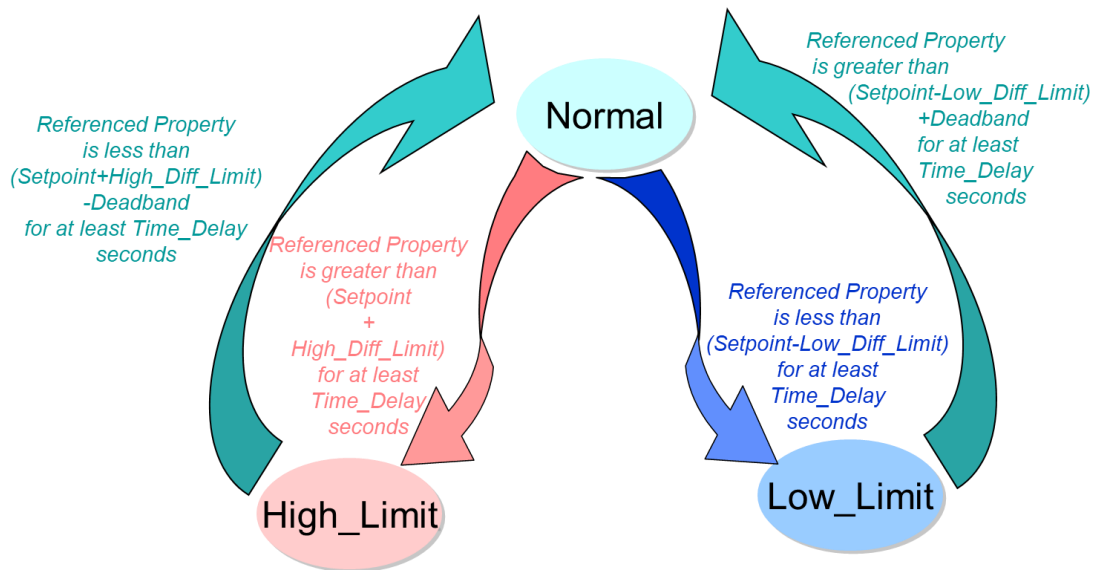


The *Command_Failure* algorithm applies to Binary Output, Multi-state Output and Event Enrollment objects. The diagram shows the Binary Output object as an example.

The concept is that the object’s *Present_Value* is continuously compared with a *Feedback_Value*. If the *Present_Value* remains not equal to the *Feedback_Value* for *Time_Delay* seconds, then the object changes from the *normal state* to the *offnormal state*. Once the two values remain equal for the same period, then the object returns to the normal state.

FLOATING_LIMIT

applies to Loop, Event Enrollment

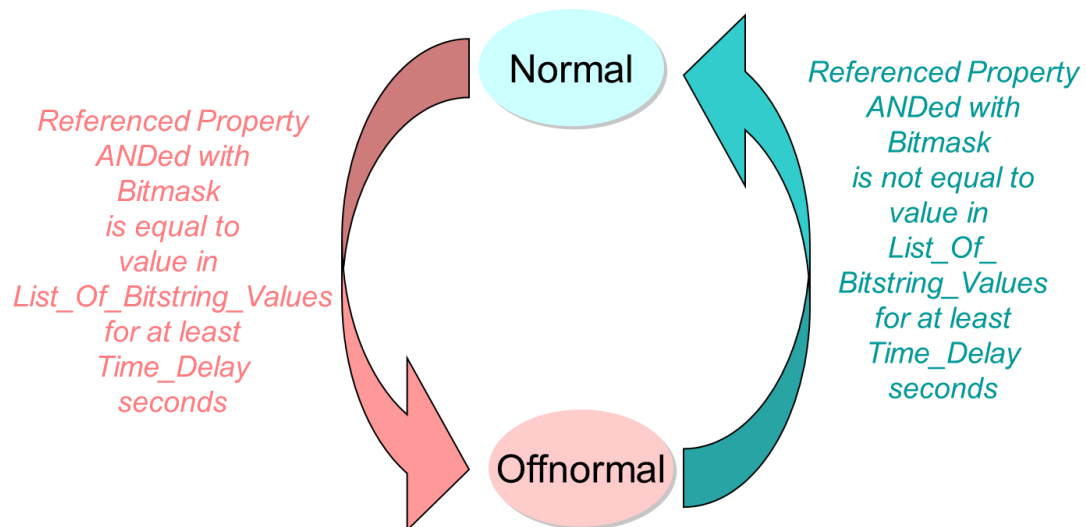


The *Floating Limit* algorithm applies to Loop and Event Enrollment objects. The diagram shows the Event Enrollment object as an example.

The concept is that the object's Referred_Property is continuously compared with a Setpoint, High_Diff_Limit and Low_Diff_Limit. If the Referred_Property remains less than the $(\text{Setpoint} - \text{Low_Diff_Limit})$, or greater than the $(\text{Setpoint} + \text{High_Diff_Limit})$, for Time_Delay seconds, then the object changes from the *normal state* to the *high limit* or *low limit state*. Once the value remains greater than the $((\text{Setpoint} - \text{Low_Diff_Limit}) + \text{Deadband})$ or less than the $((\text{Setpoint} + \text{High_Diff_Limit}) - \text{Deadband})$ for the same period, then the object returns to the normal state.

CHANGE_OF_BITSTRING

applies to Event Enrollment

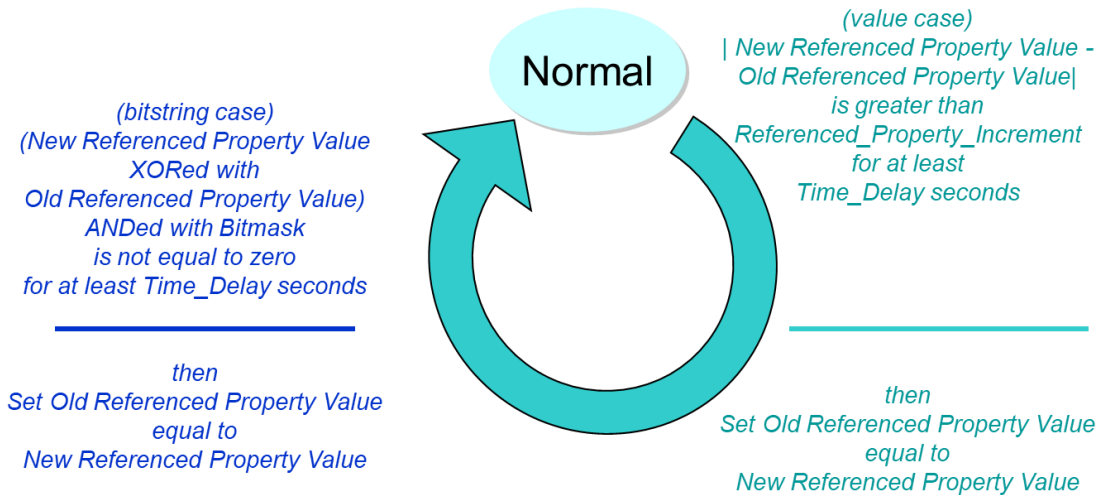


The *Change-Of-Bitstring* algorithm applies to Event Enrollment objects.

The concept is that the object's *Referenced_Property* is continuously ANDed with a *Bitmask*, and the result is compared with each bitstring in a *List_Of_Bitstring_Values*. If the (*Referenced_Property* AND *Bitmask*) remains equal to one of the values in the *List_Of_Bitstring_Values* for *Time_Delay* seconds, then the object changes from the *normal state* to the *offnormal state*. Once the (*Referenced_Property* AND *Bitmask*) remains not equal to one of the values in the *List_Of_Bitstring_Values* for the same period, then the object returns to the normal state.

CHANGE_OF_VALUE

applies to Event Enrollment



The *Change-Of-Value* algorithm applies to Event Enrollment objects.

Don't confuse the Change-Of-Value algorithm with ChangeOfValue (COV) reporting which is a different mechanism in BACnet! We have a separate tutorial on COV.

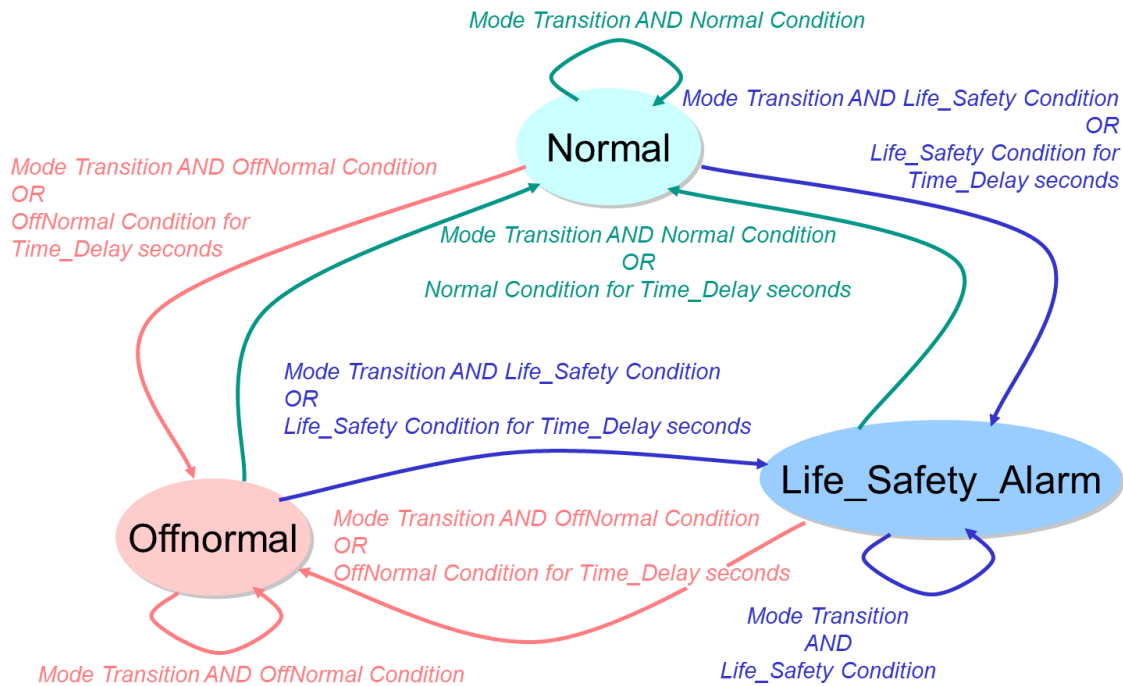
This algorithm is never “offnormal.” It is only able to generate TO-NORMAL and TO-FAULT events. There are two possible forms for the COV algorithm: bitstring and value.

The bitstring concept is that the object’s New Referenced_Property Value is continuously *XORed* with the previous (Old) Referenced_Property Value, then the result is *ANDed* with a Bitmask. If the ((New XOR Old) AND Bitmask) remains not equal to zero for Time_Delay seconds, then a TO-NORMAL transition is generated, and the Old Value is updated with the New Value.

The value concept is to continuously take the absolute value of the object’s New Referenced_Property Value minus the previous (Old) Referenced_Property Value. If the result is greater than the Referenced_Property_Increment for Time_Delay seconds, then a TO-NORMAL transition is generated, and the Old Value is updated with the New Value.

CHANGE_OF_LIFE_SAFETY

applies to LifeSafetyZone, LifeSafetyPoint and Event Enrollment



The *Change_Of_Life_Safety* algorithm applies to LifeSafetyZone and LifeSafetyPoint objects.

Unlike the other algorithms, lifesafety objects have both a *state* and a *mode*. So, for example a smoke detector can be normal/fault/offnormal in a "test mode" or an "active mode".

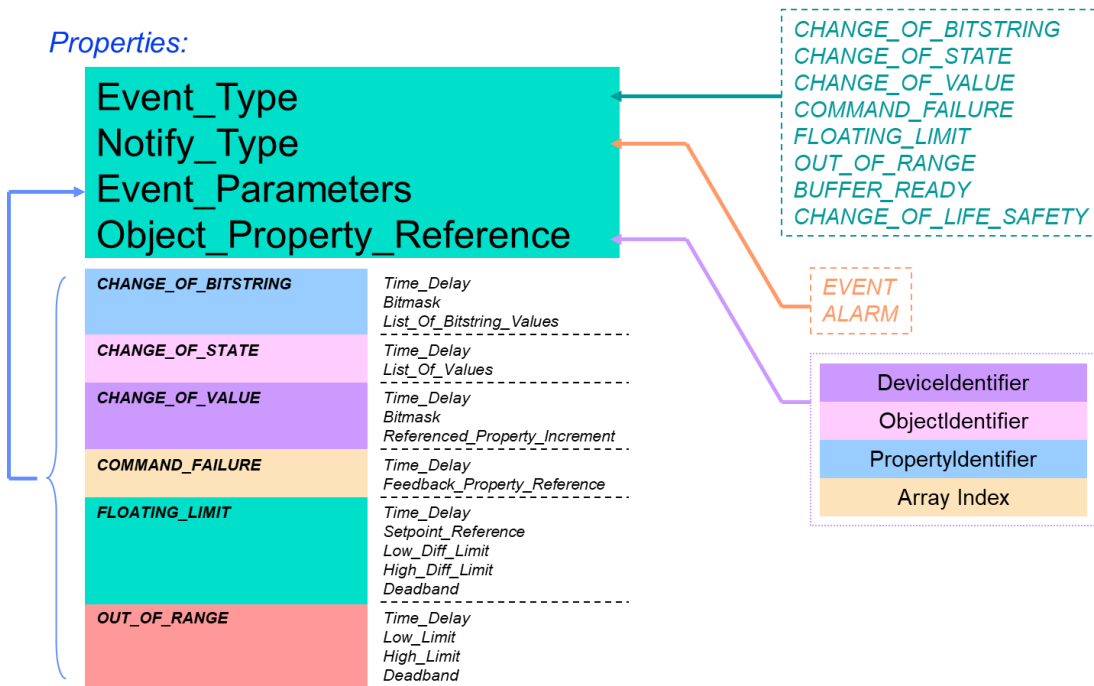
Other Standard Algorithms

Although we won't detail them here, there are many other standard algorithms whose behavior is similar in some respects to those we have already discussed. For example, unsigned-range, double-out-of-range, signed-out-of-range, unsigned-out-of-range are similar to out-of-range except for the datatype of the limits.

Event Enrollment Object

The Event Enrollment (EE) object is used to monitor an object-property value in another object, then to apply one of the standard algorithms to the monitored value in order to detect changes of state. The EE object has various properties that control event detection, the current state of the monitored object and the notification class to be used to send notifications.

Event Detection Properties



The *event detection* properties of an Event Enrollment object determine what object property is to be monitored using what algorithm and what parameters.

The *Event_Type* property specifies which of the standard algorithms should be applied when monitoring.

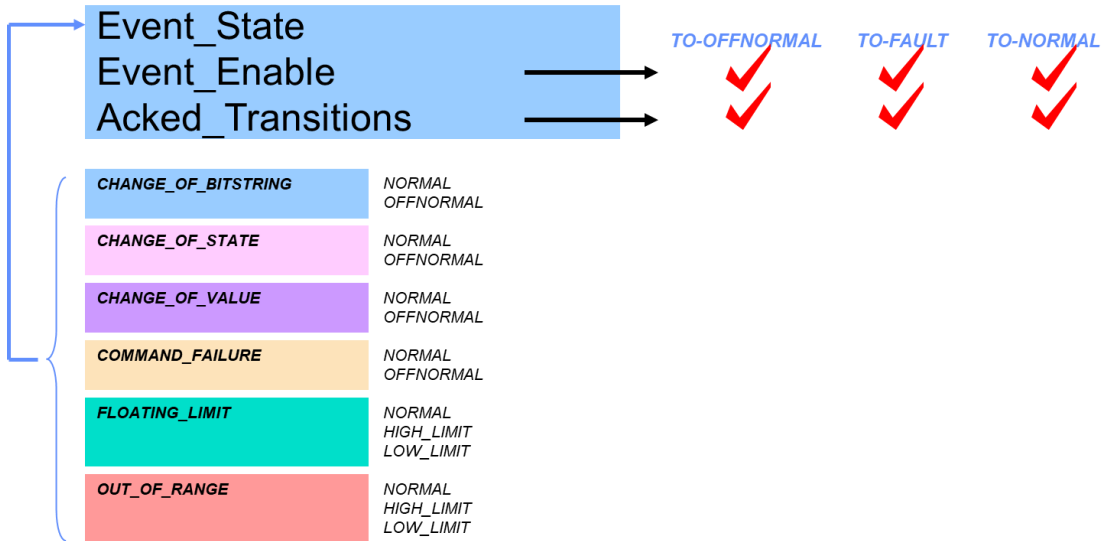
The *Notify_Type* property serves only to distinguish any transitions as events or alarms.

The *Object_Property_Reference* property defines that specific object property that is to be monitored by this Event Enrollment object. You'll notice that this property is what is called a *DeviceObjectPropertyReference*. Some BACnet devices are only capable of monitoring properties of objects in the same device as the EE object. These so-called *internal EE* objects do not include the *DeviceIdentifier* component of the reference. More capable devices that include a BACnet client are able to use the *DeviceIdentifier* to talk to and monitor object properties *in other BACnet devices* and are so-called *external EE* objects.

The *Event_Parameters* property defines a collection of parameters that is different depending on the *Event_Type*. These parameters are used by each standard algorithm to determine if an event transition has occurred.

Current Event State

Properties:



The current state of an Event Enrollment object is reflected in these properties.

The *Event_State* indicates one of several possible states based on the *Event_Type*.

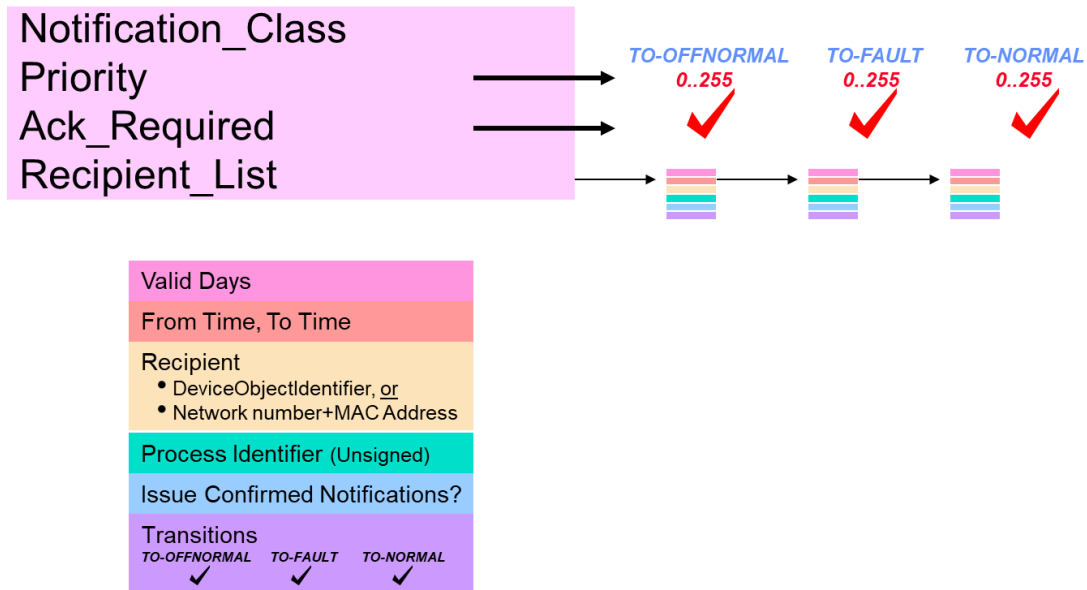
The *Event_Enable* indicates which types of event transitions should generate notifications. Of course, at least one of the flags should be present in *Event_Enable* or no notifications can be generated. This means that you can "configure" which transitions the EE object should pay attention to, i.e. any combination of to-offnormal, to-fault or to-normal.

The *Acked_Transitions* indicates which events that may have already taken place have been acknowledged by the receipt of an AcknowledgeAlarm service request. Don't confuse the receipt of a 'Result(+)' confirmation of a ConfirmedEventNotification, with an AcknowledgeAlarm service request! When an event is detected, for example a to-offnormal transition, the corresponding to-offnormal bit in *Acked_Transitions* is affected. If the notification class specifies that to-offnormal transitions require human acknowledgement then that bit is cleared in *Acked_Transitions*. Otherwise, it is set since we don't need a human acknowledgement, so we pretend that it has been acknowledged already.

Notification Classes

The Notification Class (NC) object is used to define recipients who should receive notifications when any object that uses a given notification class has detected an event.

Properties:



Normally there will be many sources of alarms and very few recipients. It is also typical that there are only a small number of different combinations of parameters that are needed. For efficiency, Notification Class objects are used to define potential recipient devices.

The *Notification_Class* property defines the class number for this Notification Class object. This will be the same as the instance of the Notification Class object.

The *Priority* specifies a value from 0..255 (0 being most important, 255 least important). This number is used by the recipient device(s) to sort in-coming event notifications in order of importance. Notice that you can specify a different priority for each of the three types of event transitions.

The *Ack_Required* specifies for each event transition whether a human acknowledgement is required. In this context, “acknowledgement” means the receipt of an AcknowledgeAlarm service request referencing an event notification generated with this Notification Class.

The *Recipient_List* property specifies a list of what are called *BACnetDestinations*. A single Notification Class object may have one or many destinations. Each one specifies six parameters for qualifying and routing event notifications. The destination is only valid to be used if the event transition occurs on one of the Valid Days, between the From Time and To Time. At any other time, the destination is ignored, possibly resulting in no notification being sent. As with Event Enrollments, the *Recipient*, *Process Identifier*, *Issue Confirmed Notifications* and *Transitions* flags specify who the intended recipient device and process are, the type of notification to send, and the transitions that should result in a notification being issued.

More specifically, *Valid Days* is a bitstring of seven bits that correspond to the seven days of the week. You can specify which day a given recipient is valid. For example, you might have an NC that sends certain alarms to a security guard desk, but only on Saturday and Sunday.

The *From Time* and *To Time* specify the time of day as an inclusive range. The recipient is only valid during those times (on those days).

The *Recipient* can be in one of two forms. You can specify a Device object identifier (an object identifier whose object type is DEVICE and whose object instance is the destination device instance), or you can specify a (network number, MAC address) pair. Why does NC provide this option? If the device form is used (this is most common) then it is up to the NC object to locate the device when an alarm notification needs to be sent. There are various ways to do this. Some schemes would initiate a Who-Is(device,device) and wait for some seconds for a reply I-Am to arrive. At that point, the device would know the network number and MAC address from the I-Am response. This doctrine could delay the issuance of the notification by several seconds or more. More typically, when a device that has NC objects starts up, it can issue those Who-Is messages immediately in anticipation of alarms occurring later. The results can be cached in RAM for later use. On the other hand, using the (network,MAC) form of recipient avoids this altogether because the destination is provided in the Recipient. The downside of this approach is that if a device must ever be relocated, for example to a different network or MAC address, then you must manually locate all NC objects that are using the (net,MAC) form of recipient and change them manually. This is a labor-intensive, or high-maintenance issue when there are a lot of alarm-generating devices so this second form is rarely used.

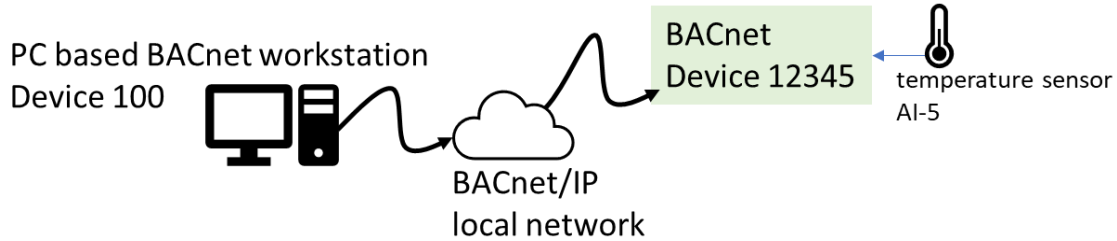
The *Process Identifier* portion of a recipient is a number that is meaningful to the recipient device which indicates what "process" the alarm is intended for. Some BACnet alarm clients totally ignore this and accept and process all notifications in one place. However, some alarm clients use this as a way to sort incoming notifications into buckets based on the process identifier. For example, maybe process identifier 1 means maintenance alarms, 2 means critical alarms and 3 means life safety alarms etc. When configuring an NC object, one must understand the capabilities of the alarm client(s) in order to set this up correctly.

Issue Confirmed Notifications is true if the event notification should be sent as a confirmedEventNotification and false if sent as an unconfirmedEventNotification. Unconfirmed notifications are just sent when they occur. There is no "confirmation" that the message was received by the recipient. Some applications want to have more assurance and prefer to use confirmed notifications instead. A confirmed notification gets sent to a recipient and the recipient is obliged to "confirm" that receipt by sending back a handshake message. If the NC object does not receive that handshake within a short period of time it will retry the sending. Typically, the time that the NC object waits for a reply, as well as the number of retries it may use are configuration parameters in the device. While not specified by BACnet, typically the time it waits is 3-5 seconds and there are one or two retries.

The *Transitions* portion of the recipient is another bitstring with three bits representing which transitions this is a valid recipient for. For example, when sending notifications to the security guard desk, perhaps we only care about to-offnormal and to-fault events but returns to normal are not important on the weekend.

Step-by-Step Example

Let's take a simple example and go through it step-by-step highlighting details about what happens. Here is our example setup:



Our test device is device instance 12345. It has a temperature sensor represented as analog input instance 5 (AI-5). From moment to moment, device 12345 is measuring the temperature and updating AI-5;PRESENT_VALUE to indicate the temperature. Let's say that nominally it is measuring about 70°F. We want to generate an alarm whenever the temperature is outside of the range 65 to 80. We want to enable both high and low limit alarming. When alarms occur, we want to send the notifications to our BACnet workstation device 100, any time of day and any day of the week. We want to send confirmed notifications. Our workstation has only one alarm process ID #1. The alarms do not require human acknowledgement and they are lower priority. We will treat offnormal, fault and return to normal equally but we want to report any of those conditions.

To setup device 12345, we need to first set some of the properties for AI-5:

Time_Delay	0	there is no delay after a limit is exceeded before we send a notification
Notification_Class	0	use NC-0 for recipients
High_Limit	80	
Low_Limit	65	
Deadband	1	must be <=79 and >=66 to return to normal
Limit_Enable	{11}	we want both high and low limits
Event_Enable	{111}	we want offnormal, fault and return to normal events
Notify_Type	0	alarm
Event_Detection_Enable	true	

If we read our status properties we would see:

Status_Flags	{0000}	all normal
Event_State	0	normal
Acked_Transitions	{111}	we start out as all normal

We also need to setup NC-0:

Priority	{200,200,200}	all transitions are lower priority
Ack_Required	{000}	none require human acknowledgement
Recipient_List		
Valid Days	{1111111}	all days of the week
From Time	00:00:00	
To Time	23:59:59	all day long
Recipient	DE-100	device 100
Process Identifier	1	
Issue Confirmed Notifications	true	
Transitions	{111}	all transitions

Here is a step-by-step timeline of what happens:

>temperature rises to 81.
 >Event_State changes to HIGH_LIMIT
 >Status_Flags "alarm" bit is set {1000}
 >Because NC-0;Ack_Required<alarm> is 0 we do not require human acknowledgement so AI-5:Acked_Transitions<alarm> is set to 1 (it is "auto-acked")
 >Because today is one of the Valid Days, and the time is between 00:00:00 and 23:59:59 and Transitions<alarm> is set then this is a valid recipient.
 >Device 12345 issues a ConfirmedEventNotification and sends it to Device 100.
 >Device 100 receives the ConfirmedEventNotification and replies with Simple ACK (SACK).
 >Event notification contains:

Process Identifier	1
Initiating Device Identifier	DE-12345
Event Object Identifier	AI-5
Time Stamp	2021-Jun-04 12:34:56.00
Notification Class	0
Priority	200
Event Type	out-of-range
Message Text	"High Limit was exceeded"
Notify Type	Alarm
AckRequired	false
From State	normal
To State	high limit
Event Values	{exceeding-value: 81.0, status-flags: {1000}, deadband: 1.0, exceeded-limit: 80.0}

Some time later the temperature drops to 79:

>Event_State changes to NORMAL
 >Status_Flags "alarm" bit is cleared {0000}
 >Because NC-0;Ack_Required<normal> is 0 we do not require human acknowledgement so AI-5:Acked_Transitions<normal> is set to 1 (it is "auto-acked")
 >Device 12345 issues a ConfirmedEventNotification and sends it to Device 100.
 >Device 100 receives the ConfirmedEventNotification and replies with Simple ACK (SACK).

>Event notification contains:

```

Process Identifier      1
Initiating Device Identifier  DE-12345
Event Object Identifier  AI-5
Time Stamp             2021-Jun-04 12:34:56.00
Notification Class     0
Priority               200
Event Type             out-of-range
Message Text           "Returned to normal"
Notify Type            Alarm
AckRequired            false
From State             high limit
To State               normal
Event Values           {exceeding-value: 79.0,
                      status-flags: {0000},
                      deadband: 1.0,
                      exceeded-limit: 80.0}

```

When we require human acknowledgement, the steps are similar but there are several key differences. Suppose we change NC-0 so that acknowledgement is required:

```
Ack_Required {111} all require human acknowledgement
```

Let's look at how the timeline changes:

>temperature rises to 81.

>Event_State changes to HIGH_LIMIT

>Status_Flags "alarm" bit is set {1000}

>Because NC-0;Ack_Required<alarm> is 1 we do require human acknowledgement so
AI-5:Acked_Transitions<alarm> is set to 0 (not yet acknowledged)

As before:

>Device 12345 issues a ConfirmedEventNotification and sends it to Device 100.

>Device 100 receives the ConfirmedEventNotification and replies with Simple ACK (SACK).

Keep in mind that the SACK is not the same as human acknowledgement. It is simply confirming receipt of the notification!

Note that the Event notification is the same as before except:

```
AckRequired true
```

This causes the workstation to inform its human operator that this alarm is special and asks the human to acknowledge that they have seen it and are taking steps to correct the issue. Typically they would end up clicking some kind of ACK button. That action would cause the workstation device to send device 12345 an AcknowledgeAlarm message containing:

```

AcknowledgingProcessIdentifier  1
EventObjectIdentifier           AI-5
EventStateAcknowledged          high limit
Timestamp                       2021-Jun-04 12:34:56.00
AcknowledgementSource           "Fred the operator"
TimeOfAcknowledgement           2021-Jun-04 12:35:00.00

```

Device 12345, upon receiving this will change AI-5:Acked_Transitions<alarm> to 1 (acknowledged).

In such special cases, device 12345 is also obligated to issue what are called "Ack Notifications". This is essentially the same as the ConfirmedEventNotification except that the Notify Type is AckNotification and the AckRequired, From State and Event Values are not included:

Process Identifier	1
Initiating Device Identifier	DE-12345
Event Object Identifier	AI-5
Time Stamp	2021-Jun-04 12:34:56.00
Notification Class	0
Priority	200
Event Type	out-of-range
Message Text	"High Limit was exceeded"
Notify Type	AckNotification
To State	high limit

Advanced Alarm Concepts

Although the alarming mechanism presented so far is very sophisticated, there are additional advanced features that we have avoided explaining. Let's look at each of those features now. Generally speaking, objects that support intrinsic alarming will also implement these extra properties.

Event_Time_Stamps	this is an array of three time stamps that corresponds to the three event states offnormal, fault and normal.
Event_Message_Texts	this is an array of three strings that correspond to MessageText that was sent for each event state.
Event_Message_Texts_Config	this is an array of strings where you can configure the three event messages.
Event_Detection_Enable	true or false. When event detection is disabled then Event_State is always normal. When event detection is enabled then Event_State tracks the result of the event algorithm.
Event_Algorithm_Inhibit	true or false. When the algorithm is inhibited then Event_State is always normal. This is subtle, but the idea is that one could in theory enable event detection but inhibit the algorithm. In that case you could "poll" the Event_State to see if it was normal or offnormal, but this would not generate notifications.
Event_Algorithm_Inhibit_Ref	this is a DeviceObjectPropertyReference. If it points to a valid true/false value then that value is copied into Event_Algorithm_Inhibit. So for example you could have some BACnet object that determines whether alarming is turned on or off.
Time_Delay_Normal	usually the algorithms have a single delay time that is used to delay both offnormal and return to normal the same amount. This property if present allows an object to have different time delays for offnormal and return to normal.
Reliability_Evaluation_Inhibit	usually the Reliability property (if present) determines the reliability of the value. This allows for scenarios where Out_Of_Service is true to also allow the overwriting of Reliability for testing purposes. However this can be at cross purposes with algorithms that depend on reliability, and how this affects determination of faults. Inhibiting reliability evaluation can be useful to change this behavior.
Fault_High_Limit, Fault_Low_Limit	are used to provide different limits for fault purposes than the usual limits for alarm purposes.

Conclusion

BACnet provides a rich and complex alarm detection and notification mechanism that includes many options. The designer is faced with a lot of decisions about which features to implement and this places a burden on BACnet alarm clients to accommodate many variations of alarm sources.

Legal Stuff

This paper represents the author's opinions only and does not necessarily represent the views of the author's employer, SSPC-135, ASHRAE or any other organization. While the author believes all information is factual, it is provided as-is without any guarantees of suitability for a particular purpose. The name "BACnet" and its logo are registered trademarks of ASHRAE Inc.

Contact the Author

David Fisher

president, PolarSoft® Inc.
368 44th Street
Pittsburgh PA 15201-1761 USA

+1-412-683-2018 voice
dfisher@polarsoft.com