bacnet experts

PolarSoft® Inc.
914 South Aiken Ave
Pittsburgh PA 15232-2212
412-683-2018 voice
info@polarsoft.com
www.polarsoft.com

# Commandability in BACnet

David Fisher

13-Aug-2016

TUTORIAL

## Contents



# Commandability in BACnet

13-Aug-2016

David Fisher

## Introduction

When a BACnet device contains a control object, such as Analog Output, Binary Output, Multi-state Output, etc. it doesn't matter how many other devices may be interested in finding out what the Present_Value of the object is at any moment. They can use ReadProperty or ReadPropertyMultiple services to find out. Other than traffic, it also doesn't matter which device reads the value first or second. But device(s) that want to change the control output need to use WriteProperty or WritePropertyMultiple to write to the Present_Value in order to *change the actual output*. When there are two or more potential "writers" there is the classic control problem: which writer is more important, or is it simply whoever writes last, wins?

Sometimes that's an issue and sometimes it isn't. For example, many controller devices represent their internal relationships in BACnet-visible terms, but nonetheless the relationships are fixed within the software architecture of the device.

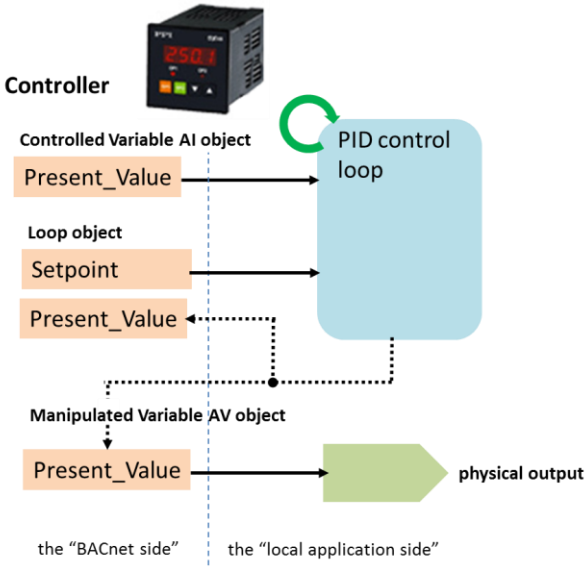



Figure 1 shows a typical controller that has a measuring input for the controlled variable, a PID loop that evaluates this measured input relative to a setpoint and a corrective output that is permanently coupled to a physical output.

Although the relationships are not alterable, in this case by design, the values known to the controller are "visible" on the BACnet side through properties of several objects that represent the internal software algorithm in action.

Even though the physical output is modeled as an AV object, generally only the PID loop internal to the controller is ever really writing to the AV;Present_Value in this example.

**Figure 1 -** The PID control loop has a fixed relationship to the physical output

In even simpler terms, Figure 2 shows a Schedule that controls a Binary Value and turns the output on and off according to a predetermined schedule.

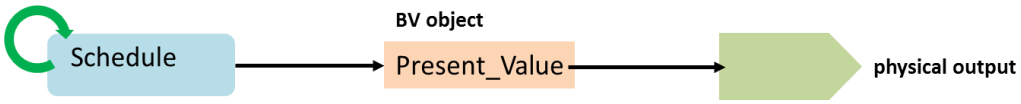



**Figure 2 -** A Schedule manipulating a BV directly

In both of these cases, there is really only one entity that needs to control the physical output, so synchronization between multiple entities isn't a requirement.

In real-world control it's not always that simple. Lot's of times, a given physical output may need to be controlled by multiple entities who have different responsibilities. Usually each responsibility has some importance relative to the other responsibilities. We might have an energy-saving algorithm that decides to turn off a fan to save energy, but a smoke evacuation procedure initiated by a smoke alarm says to turn the fan on, and that's *more important*.

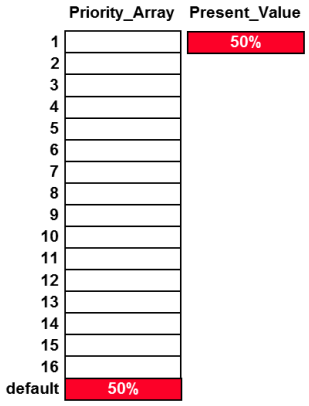
Whenever we have this kind of situation, where there are multiple control entities that need to control the same *logical output*, we need a way to *prioritize* the importance of each entity in the larger scheme. BACnet calls this feature *commandability*, which is to say the writing of property values along with the relative importance of the writer. BACnet object types that can recognize this type of writing are said to be *commandable object types*.

In addition to the Present_Value, commandable objects include two special properties: Priority_Array and Relinquish_Default.

## Behavior of Commandable Objects

Clause 19.2 in BACnet defines the expected behavior of so-called commandable objects. When a device is restarted, and more importantly when there are no entities that have commanded the object to a specific value, the Present_Value of the object is automatically set to the Relinquish_Default value. How does the object "know" when there are no entities commanding it? This is where the Priority_Array comes into play.

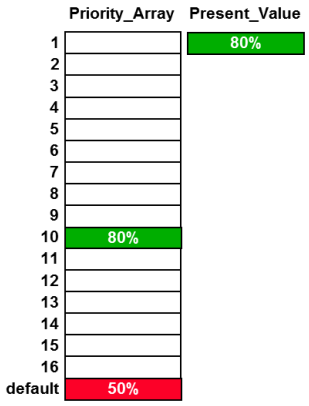
The Priority_Array is a collection of 16 numbered "slots", indexed as 1,2,3…etc. Slot 1 represents the *most important commanding* value, while slot 16 represents the *least important commanding* value. Each slot can hold either a value like on/off, or an analog value like 72.5, or the slot can be "empty" which means that it holds a special value called *null*.




If all of the 16 slots are empty (contain null) then there are no entities currently commanding the object. In this case Relinquish_Default acts like a "17th" slot.

In this example, since all of the Priority_Array slots are empty, the Present_Value takes on the Relinquish_Default value

How do these slots get filled? Each time any device sends a WriteProperty (or WritePropertyMultiple) to the Present_Value of a commandable object, the writing client includes a *priority* along with the value to be written. The commandable object saves the value in the corresponding slot in the Priority_Array, overwriting any value that may already be in that slot.




In this example, we write to Present_Value a value of 80% at priority 10.

Slot 10 is filled with the 80% value.

Then the commandable object evaluates each slot starting with slot 1 until it finds the first non-empty slot and that slot's value is transferred to Present_Value.

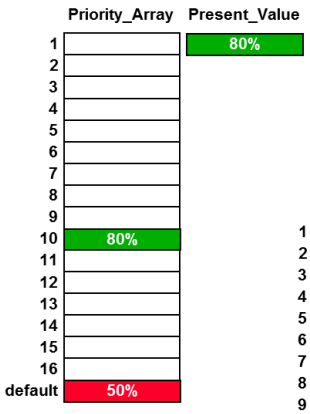
If *no priority is provided* in the WriteProperty, then the commandable object interprets this to be the same as a priority of 16 (least important).

If a priority is provided in a WriteProperty, but the object is not commandable then the priority is ignored.

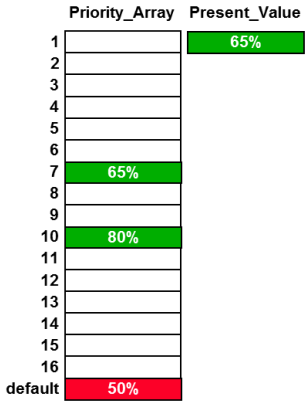
## How does this solve the problem?

In order to be effective, the control system designer and installer need to establish a hierarchy of importance between different entities in the overall control system. Then each commanding entity must be assigned or configured to know its importance in the hierarchy so that when the entity tries to write to commandable objects, it can convey its priority. But, it is up to the system designer and/or installer to ensure that the hierarchy is logical and consistent with the desired behavior for the control system. It's also important that the commanded objects, that are to be synchronized in this way, in fact provide commandability as a feature. Otherwise (obviously) they can't be controlled in a hierarchical manner.

How does the priority, once commanded, ever return back to normal? Each entity that potentially writes to a commandable object needs to realize that it's control is not absolute or forever. A controlling entity takes control (at its priority level) by writing to a commandable object at the controlling entity's priority. When the entity *no longer requires control* it writes a null value at its priority level. This has the effect of emptying the slot and "relinquishing control" at that level.




Let's use the example from earlier. A weekly schedule entity decides that some commandable output needs to be set to 80% based on the schedule. We've assigned the schedule an importance of 10. The day/time arrives and the schedule writes to Present_Value 80% at priority 10.




A separate entity is responsible for "optimal start" and its algorithm determines based on the season, outside air and occupancy time that some number of minutes of precooling is required. Then, that many minutes prior to scheduled occupancy, it writes to this output a value of say 65%, but at priority 7. Note that the original weekly schedule value is retained in the Priority_Array, but the output goes to 65% because that is the most important value at that moment.

Finally the scheduled occupancy time arrives, the optimal start routine realizes that its job is done, so it writes a null value at priority 7, relinquishing control. The Priority_Array is reevaluated and the retained weekly schedule value of 80% is restored.

## Is there a standard "meaning" for each of the 16 priorities?

With one exception, BACnet allows designers to use prioritization to define any scheme that is appropriate for a given facility. Having said that, BACnet does *recommend* a particular use for some of the 16 priorities.

| *slot* | *recommended usage* | *example explanation* |
|---|---|---|
| 1 | Manual LifeSafety | When a Fire Captain arrives at a facility and wants to command an output to a particular state, taking priority over all other considerations. |
| 2 | Automatic LifeSafety | When a smoke detector detects smoke and activates an automated smoke evacuation procedure. |
| 3, 4 | none | |
| 5 | Critical Equipment Control | The access panel on the side of an air handler has a microswitch that detects the panel has been removed causing the fan to be commanded OFF. |
| 6 | Minimum On/Off | A fan is turned off, and is then commanded off (internally) at this priority for say two minutes preventing "normal" writes at higher (less important) priorities from taking effect.<br>*This priority may only be used for minimum on/off time!* |
| 7 | none | |
| 8 | Manual Operator | Normal human operation of the control system, a human changes some output. In common BAS parlance this is often called *override* which is not to be confused with BACnet's definition of override which is something completely different. |
| 9..16 | none | |

**Table 1 -** Recommended uses for Priority Levels

## Some (not obvious) things about Prioritization

The mechanism of BACnet prioritization is very powerful, and seemingly simple. But there are a number of not-so-obvious side effects and understated assumptions that deserve discussion.

### Support All Priorities

Devices that implement commandable objects must support all of the 16 possible priorities. That means that if you have a commandable object, I can write to it at any priority from 1 to 16 and it should retain that value and restore that value even after writing a different value at a different priority. You can't elect to only use one or two priority values, or always fix certain behavior at certain priorities (except for priority 6 which is always and only used for minimum on/off).

BACnet devices that serve as clients who may potentially write to commandable objects, must be configurable to use any priority, even if their application doesn't need that flexibility.

### Don't Write Continuously

BACnet devices that serve as clients who may potentially write to commandable objects, must not write a command value at priority X, and then some seconds later rewrite the same value at priority X, repeatedly. This is unnecessary and just wastes network bandwidth.

### Volatility

BACnet devices with commandable objects are not required to maintain the Priority_Array across restarts.

### LifeSafety Doctrine

The special priorities 1 and 2 are most commonly used only for Fire and LifeSafety applications where commands from local fire-fighting authorities must take precedence over all other considerations. BACnet's recommended doctrine is that human-based control should always be more important than automation-based control. But there are some situations where implementors may feel the need to reverse that doctrine. In doing so, keep in mind that the overwhelming majority of BACnet devices don't work that way, so changing this doctrine requires careful evaluation of those devices that are in use and their proper configuration.

### Internalized Usage of Prioritization

Although we commonly think about commandability as an "external" issue, meaning how do we behave when other devices write to our properties, this is a restrictive view. BACnet's design and intent is to use commandability to control behavior both externally visible and internal to the device.

There are many examples, but consider the classic case of a binary style output perhaps controlling a fan or large motor. Because of mechanical characteristics of the controlled device, it is unwise to repeatedly turn the motor on and off in a short period of time. This can cause overheating and wear on belts and other mechanical parts. Typically these negative effects are mitigated by imposing a minimum on time and/or off time. For example after the motor is turned on, we may require a minimum of two minutes of run time before allowing the motor to be turned off. Of course, not all binary output applications have this requirement, but some do and therefore minimum on/off is an optional but common feature.

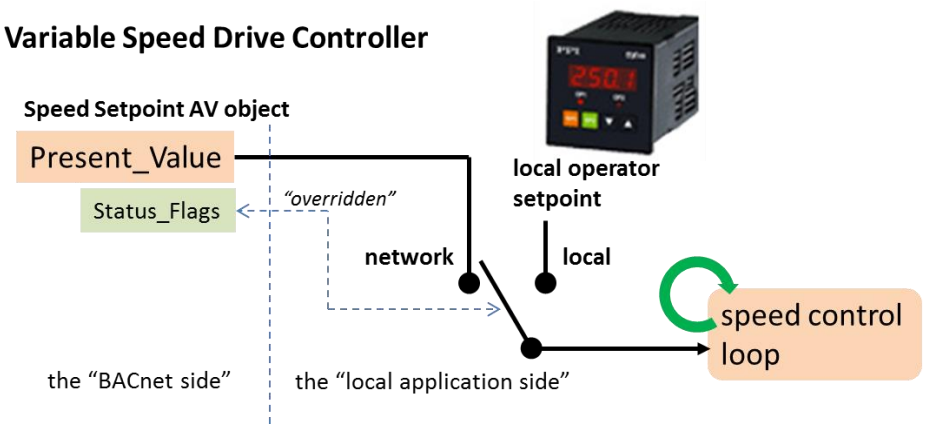
For commandable objects that implement this feature, the idea is that whenever the Present_Value is written with a value that is *different from* the existing (pre-write) Present_Value, the object internally also writes the new value into the Priority_Array slot 6 and starts a timer to keep track of the minimum time. When that timer expires, the device again internally rewrites Priority_Array slot 6 with a null value, relinquishing control at the minimum on/off priority. This extra writing to Priority_Array takes place regardless of the incoming priority.

### Internal Overrides

Some implementers have an arms-length view about commandability. They implement commandable objects that have Priority_Array and Relinquish_Default and the "correct" behavior, but when it comes to their own internal-to-the-device procedures they feel free to "lock out" BACnet-side access to some outputs whenever they feel it is needed.

First we should say that this is not best practice and to a degree flys in the face of what commandability was intended to do. A better doctrine would be to decide (and make configurable) a priority level that is used by these internal algorithms to command to "when

needed." The internal process(es) would use the same mechanism that gets invoked when WriteProperty is received, so that the internal entities are treated the same as external devices.




But, many developers don't take this approach, and instead throw an internal software switch that just locks out the BACnet stack. If one is going to take this approach, then as a minimum it should be reflected in the OVERRIDDEN bit in the Status_Flags property for the output. At least this will provide a BACnet-visible mechanism for humans to realize why the output is not responding to BACnet-side commands.

## What BACnet objects are Commandable?

One of the characteristics of BACnet's three main output objects (Analog Output, Binary Output and Multi-state Output) is that they are *required* to be *commandable*. In this context, commandability means that the Present_Value is required to be writable AND the Priority_Array and Relinquish_Default properties must be present AND writes to Present_Value must follow the rules of Clause 19.2. On the other hand, all of the value objects may be implemented as either input-like or output-like, and when output-like are NOT required to be commandable. However, they may be implemented as commandable at the implementer's discretion.

This means that the Priority_Array and Relinquish default properties of a value object might be implemented, and if so must behave in the manner described in 19.2.

Among the standard BACnet object types, those that are required to implement commandability are Access Door, Analog Output, Binary Lighting Output, Binary Output, Lighting Output, and Multi-state Output. Table 2 shows the other standard object types that are optionally commandable. By "optional" we mean that the implementer may choose, on an object-instance basis, whether a given instance of a given object type implements commandability or not. That's a subtle but important flexibility for implementers. For a given object type, e.g. AV, some instances of that could be commandable and some not.

Channel objects are special in the way that they handle commandability. Rather than the Channel object itself being commandable and having a Priority_Array and Relinquish_Default, the Channel object *forwards the priority,* that may be included in the write to the Channel object Present_Value, on to the constituent referenced objects that are part of the Channel. The Channel object Present_Value is required to be writable, and you may write to it as if it was commandable, but any priority that is included in the write is just passed on when writing the channel value to each reference.

| *object* | *commandability* | *notes* |
|---|---|---|
| Access Door | required | |
| Analog Output | required | |
| Analog Value | optional | |
| Binary Lighting Output | required | |
| Binary Output | required | |
| Binary Value | optional | |
| BitString Value | optional | |
| Channel | no | Present_Value (see 19.2.1.6) |
| CharacterString Value | optional | |
| Date Value | optional | |
| Date Pattern Value | optional | |
| DateTime Value | optional | |
| DateTime Pattern Value | optional | |
| Integer Value | optional | |
| Large Analog Value | optional | |
| Lighting Output | required | |
| Multi-state Output | required | |
| Multi-state Value | optional | |
| OctetString Value | optional | |
| Positive Integer Value | optional | |
| Time Value | optional | |
| Time Pattern Value | optional | |

**Table 2 -** Object Types that are Commandable