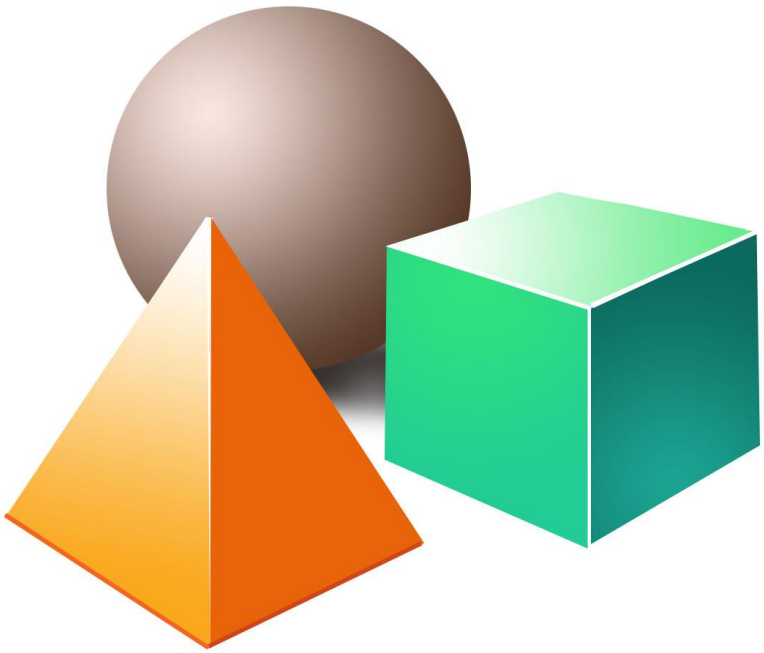
Value Objects In BACnet

David Fisher

5-Mar-2016



TUTORIAL

Contents

Value Objects in BACnet

5-Mar-2016

David Fisher

Introduction

BACnet provides a robust *object-oriented* mechanism for representing data and control information. Generally speaking, a BACnet object is a network-facing collection of *properties* each of which is an (*identifier, name, value*) tuple. Although a typical object has many properties, a lot of the standard BACnet object types include a property called Present_Value that represents the most important, or key value. For example, in an Analog_Input object type, the Present_Value represents the value of some input that is being measured. As a consequence, it is common to think about objects as if they were individual data points where the Present_Value is the point itself. Of course, the other properties are useful, and in some cases also critical, to the use and operation of the object. But sometimes it is convenient to think about objects from a broader perspective.

Objects that represent sensed or measured values are typically called *input objects* in BACnet, such as Analog_Inputs and Binary_Inputs. Objects that represent control outputs for relays, or modulating actuators are typically called *output objects* in BACnet such as Analog_Outputs and Binary_Outputs. But there are many kinds of points that are not specifically inputs or outputs, or that don't necessarily represent physical connections, for example calculated values such as Enthalpy. In BACnet these are more generically called *values*. Although they can be used to actually represent physical input/output, often they are used when there is no obvious physicality.

The subject of this paper is to explore each of the BACnet value objects and to explain their required and optional functionality.

Humble Beginnings

The original 1995 BACnet standard defined only two value objects: Binary_Value (BV) and Analog_Value (AV). The BV Present_Value has an enumerated datatype that equates a zero or "off" condition with the neutral value *inactive*, and a one or "on" condition with the value *active*. The AV Present_Value has a datatype of REAL and can represent *analog* or a range of values that typically change over time. Since 1995, many new *value objects* have been introduced into the standard.

The first thing to realize about value objects is that we tend to switch back and forth between the big picture view of the object (only its Present_Value) and a detailed view of the object that looks at all of its properties. When we say *the value of the object* we usually mean the value of its Present_Value property. The second thing to keep in mind is that the value has an associated *datatype*, e.g. enumerated, REAL, unsigned etc.

The original BV and AV objects provide a means for representing only two kinds of values:

- binary, or on-off values
- analog, or a continuum of values within some range

In practical terms this meant that everything in BACnet that needed to use a value object had to be represented by only these two datatypes. That turned out to be too limiting.

Building Automation Needs More Datatypes

It is very common to have control signals, inputs and outputs that have more than only two *states*. For example, I might have a fan that can be OFF, LOW SPEED and HIGH SPEED. That would be three states. In the very early days of BACnet, implementers had to represent these kinds of values as AV, with a fixed set of allowed values like 1.0, 2.0, 3.0 etc. While workable, this was not an optimum solution. In particular there is the problem of what each state is called. Humans greatly prefer thinking in terms like OFF/LOW/HIGH rather than 1/2/3. The specific correspondence between a state value and the name of that state has to be looked up in some user manual and is disassociated from the object itself.

In BACnet, these kinds of values typically are represented by *multi-state objects*. Although the 1995 standard included Multi_State_Input (MSI) and Multi_State_Output (MSO) objects, it didn't include a multi-state value object. This was added shortly after the original standard was published.

But for many years there was a serious limitation for value objects which could only provide binary, REAL and unsigned multi-state value types. The 135-2008w Addendum changed this by introducing so-called *simple value objects*. As a result, today there are 15 value object types summarized in Table 1 below.

object type	*datatype of Present_Value*
Analog Value	REAL
Binary Value	Enumerated 0/1
BitString Value	BITSTRING
CharacterString Value	CharacterString
Date Value	Date
Date Pattern Value	Date with wildcarding
DateTime Value	DateTime
DateTime Pattern Value	DateTime with wilcarding
Integer Value	Integer
Large Analog Value	Double
Multi-state Value	Unsigned > 0
OctetString Value	OCTETSTRING
Positive Integer Value	Unsigned
Time Value	Time
Time Pattern Value	Time with wildcarding

Table 1 - Simple Value Object Types in BACnet

The Behavior of Value Objects: Input or Output?

The simplest kind of value object has a Present_Value property that can be read by BACnet clients. This Present_Value is updated periodically by some process internal to the BACnet device that contains the value object. The updating gets the value through some local procedure and then retains this value typically in a "holding bucket" for subsequent easy access. However, some implementations may forego this and proactively fetch the value any time it is required, for example during a ReadProperty execution to read that Present_Value.

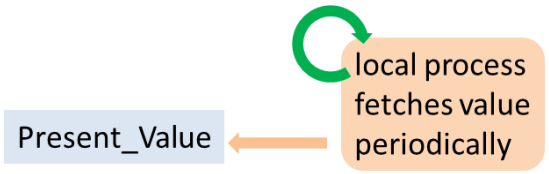


Figure 1 - An Input-like Value Object

We call this kind of value object behavior *input-like* because it is similar to the way that BACnet input objects, like Analog Input, are specified to behave. It's called "input" because from the object's perspective the value is an input to the object. More importantly, from a BACnet client's perspective the Present_Value of that object provides input to the device and client. There is an implied periodicity to the behavior because the process repeatedly does the "fetching" of the input value and refreshes the Present_Value accordingly. If a BACnet client reads the Present_Value repeatedly it should track changes in the measured or calculated input value, regardless of *how* that value is derived.

It's also possible to use value objects in another way. In some circumstances, the device may need to receive instructions or parameters from other BACnet client devices. For example, a control loop may wish to provide a BACnet-visible *setpoint parameter* that is implemented as a value object. In these cases, although the Present_Value may of course be read by the client, usually the client *writes to the Present_Value* as a way of conveying a new setpoint or adjustment. When a value object is implemented in this way, its behavior is said to be *output-like* because it is similar to BACnet output objects such as Analog Output. It's called "output" because from the object's perspective the value is something that the object uses or consumes in order to produce some change in behavior, so the object is an output for the device.

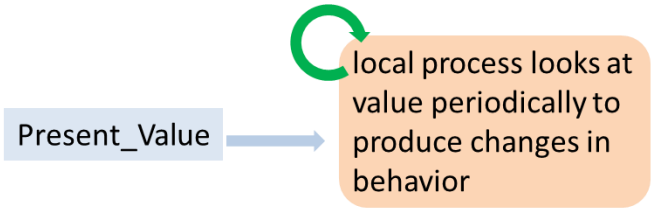


Figure 2 - An Output-like Value Object

The distinction between input-like and output-like behavior is important because it changes how we think about the application of a particular value object. Keep in mind that the object type itself is not tied to the *inputness* or *outputness* as far as the behavior that is expected for value objects. Also, in a given BACnet device and a given value object type, some instances of the value object might be input-like and some might be output-like.

Commandability

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. The details of how commandability is required to behave are discussed in another paper.

Intrinsic Reporting

Most of the value objects are allowed to support *intrinsic reporting* with or without fault detection. Specifically, the Analog Value, Binary Value, CharacterString Value, Large Analog Value, BitString Value, Integer Value, and Positive Integer Value objects may optionally implement intrinsic reporting (alarming). The details and properties that control Intrinsic Reporting apply equally to value object and other standard object types. These are discussed in another paper.

Out_Of_Service

Like straight input and output objects, value objects have a concept called *out of service*. Normally a value object is "in service" meaning that it is operating as an input value or output value as previously described above. In particular there is a periodic process that is measuring or calculating the value that is reflected in Present_Value when it is read, or that is producing and refreshing some physical output or value that is periodically grabbed from Present_Value and used internally by other processes in the device. These periodic processes usually occur continuously without interruption. However, there are situations when it is desirable to block or interrupt these periodic processes.

As an example, consider a value object that represents a complex value like Enthalpy. This is a calculation based on temperature and relative humidity. In some devices this might be implemented as an input-like value object, say an Analog Value. The temperature and humidity come from other sources the device knows about, perhaps as physical inputs. Some process continuously acquires these values, does the appropriate calculations and saves the result in Present_Value.

In one scenario, let's say that one of the physical input sensors is broken or miscalibrated to the extent that the Enthalpy calculation is compromised. There may be other processes or BACnet clients that depend on this AV for a reasonable value. It might be desirable to temporarily disable the periodic refreshing of Present_Value until such time as the sensor can be replaced or repaired. In this case we can take the AV object "out of service" and replace its Present_Value with a nominal temporary value.

Another scenario might be that we are trying to test or validate some procedure that refers to the AV, and we want to simulate certain conditions or specific values to see how the procedure performs. In those cases, again we would like to take the value object out of service and put test values in Present_Value by writing to it.

BACnet handles these kinds of examples with the Out_Of_Service property. When Out_Of_Service is FALSE, the object is normal and "in service". When Out_Of_Service is TRUE then the object behaves as described above and allows writing to Present_Value by BACnet clients in order to replace the value.

It is important to note that the Out_Of_Service property is NOT required to be writable by BACnet clients. In most cases it is up to the device implementer to decide if the concept of out of service should be supported in the device or not. If not, then Out_Of_Service would not be writable and in effect the object would therefore always be "in service." However, if the implementer decides to support out of service, then the device would allow writes to Out_Of_Service and if it is set to TRUE then the object's behavior would be affected.

In the context of input-like value objects, the internal operation would look like Figure 1 when Out_Of_Service is FALSE, and instead would look like Figure 3 when Out_Of_Service is TRUE.

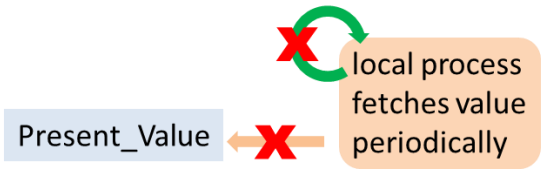


Figure 3 - An Input-like Value Object when Out_Of_Service is TRUE

BACnet does not make the distinction between implementations that halt the fetching process as opposed to halting the internal update of Present_Value.

The concept of out of service also applies to output-like value objects, but the details are more subtle. Figure 2 shows a somewhat naïve diagram of the internal operation of an output-like value object. It is over-simplified because it does not show the downstream consumer of the value and whether that consumption produces physicalities such as changing output signals or actuation. In the context of output-like value objects the concept of out of service means to block or interrupt that downstream consumer, so that any systemic effects or physical output effects, remain static while Out_Of_Service is true. This allows programs that look at and use the value object's Present_Value to be tested, without changing physicality. In order to understand this idea, we have to think of the value object's components as separate from the internal consumers and interface software.

Reliability

Like many of the standard BACnet objects, Value objects have an optional property called Reliability. This property, if present, indicates the "health" or reliability of the object as far as the object and device can determine. This is an enumerated value that enumerates various possible reasons for unreliability, or NO_FAULT_DETECTED. For objects that have a direct or indirect connection to some physical input or output, often there is a combination of hardware and software that can detect various conditions or *faults*. However, for value objects, there may not always be an obvious purpose or method for determining reliability. In those cases, the Reliability property is typically not implemented. When it is implemented it is the implementer's choice to determine how to detect faults or reliability issues and reflect those as changes in the Reliability property enumeration.

Just because a given object supports the Reliability property, it doesn't mean that in all cases it's desirable to react to changes in reliability though. For that reason, those implementations must also support another property called Reliability_Evaluation_Inhibit. When the Reliability_Evaluation_Inhibit is TRUE, the object does not even try to detect reliability changes. However, when the value of Reliability_Evaluation_Inhibit is FALSE then it is expected that reliability will be evaluated accordingly.

The Reliability property interacts with Out_Of_Service. When Out_Of_Service is FALSE (the object is "in-service"), there is a local process that evaluates reliability when Reliability_Evaluation_Inhibit is FALSE and saves the result periodically in the Reliability property. When Out_Of_Service is TRUE (the object is "out of service") this periodic updating of Reliability is blocked, similarly to the updating of Present_Value, AND the Reliability property becomes writable by BACnet clients. The purpose of this mechanism is to allow testing of the behavior of other processes in the device in response to changes of reliability by writing specific reliability enumerations to Reliability. Keep in mind that this writing to Reliability by external BACnet clients is only allowed when Reliability_Evaluation_Inhibit is FALSE.

Regardless of the state of Out_Of_Service, changes to Reliability may be considered as *fault conditions* that trigger event notification when intrinsic reporting is implemented.

Status_Flags

All of the value objects are required to support a property called Status_Flags. This is a BitString that reflects several conditions simultaneously in a compact datatype. There are four bits in the Status_Flags BitString:

bit	name
0	IN_ALARM
1	FAULT
2	OVERRIDDEN
3	OUT_OF_SERVICE

The IN_ALARM bit is TRUE (1) only when two conditions are met:

- The Event_State property is implemented and present, and
- The Event_State property does NOT have the value NORMAL

The second condition has a subtle additional side effect. If the FAULT bit is TRUE (1) then the IN_ALARM bit will also have a value of TRUE (1). The reason for this is that faults always set the Event_State to FAULT, which is NOT NORMAL.

The FAULT bit is TRUE (1) only when two conditions are met:

- The Reliability property is implemented and present, and
- The Reliability property does NOT have the value NO_FAULT_DETECTED

Again, the second condition has a subtle additional side effect. If the Reliability_Evaluation_Inhibit is TRUE then Reliability will have a value of NO_FAULT_DETECTED unless Out_Of_Service is also TRUE and some alternative value has been written to Reliability.

The OUT_OF_SERVICE bit is TRUE (1) only when the Out_Of_Service property is present AND has a value of TRUE, otherwise OUT_OF_SERVICE is FALSE (0).

Status_Flags: OVERRIDDEN

In BACnet the term *overridden* has a special meaning that is often misunderstood. This is further compounded by differences between input-like and output-like value objects' behaviors. The short definition for overridden is that a BACnet object is in an "overridden condition" when the physical input or output, or the process that derives the input-like value, or the process that consumes the output-like value, is decoupled from the object in a manner that BACnet cannot control, AND the object can detect that this decoupling is in effect.

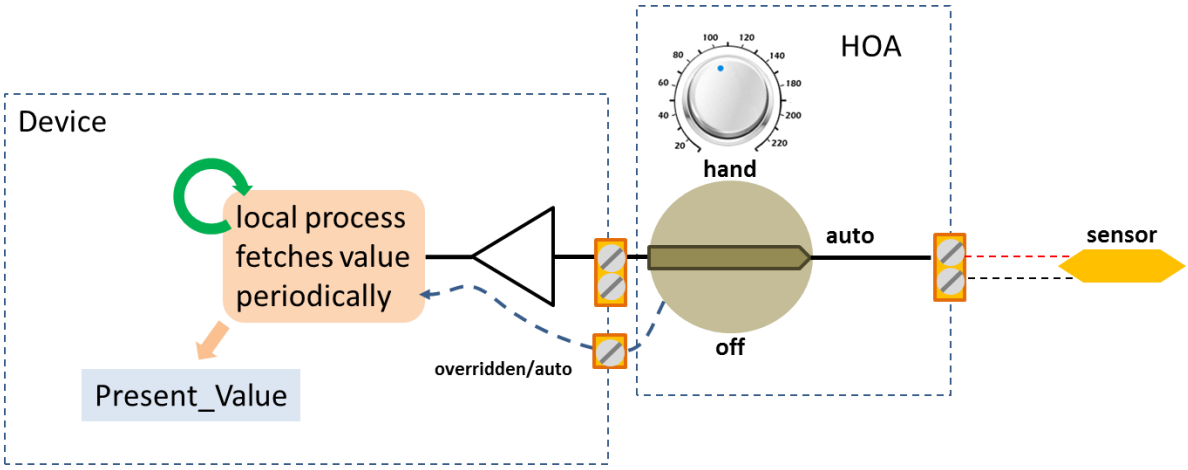


Figure 4 - An Input-like Value Object that can detect Override

In the example above, we imagine some kind of measuring input that normally connects to a sensor. However, in this case the input passes through a hand-off-auto (HOA) assembly. When the rotary switch is in the "auto" position, the sensor connects directly to the input of the Device on the left. The HOA also produces a binary signal that we call "overridden/auto" in Figure 4. Our device is able to detect this signal. In this example when the switch is auto, that binary signal is false, meaning that the main input is NOT overridden. However, if the switch is rotated to the "hand" position, instead of passing through the sensor to the device input, it passes a voltage controlled by a local adjustment knob.

The point of this example is to show that in this scenario, the BACnet device has no *control* over the position of the HOA assembly. A human can rotate the switch to a position that *overrides the sensor* producing a static value. The BACnet device doesn't have any choice, the input is set to whatever the human wants when in the hand or off positions. But the device CAN detect that the input has been overridden or not. In BACnet devices that do not have the additional inputs for detecting override, they simply don't know and must always assume that the input is NOT overridden. Generally, this same type of *external override* might be used for output-like value objects also.

For value objects that represent inputs or outputs with some kind of physicality, the concept of override is clear. To be "overridden" means that the BACnet side of the value object is unable to affect the "real" side that produces an output or is unable to read the "real" input end device. The external override is reflected in the OVERRIDDEN bit in Status_Flags.

Where it becomes more subtle is in situations where the physicality doesn't exist or is too indirect to actually detect override. Typically, this can occur when the value comes from a software process or goes to a software process.

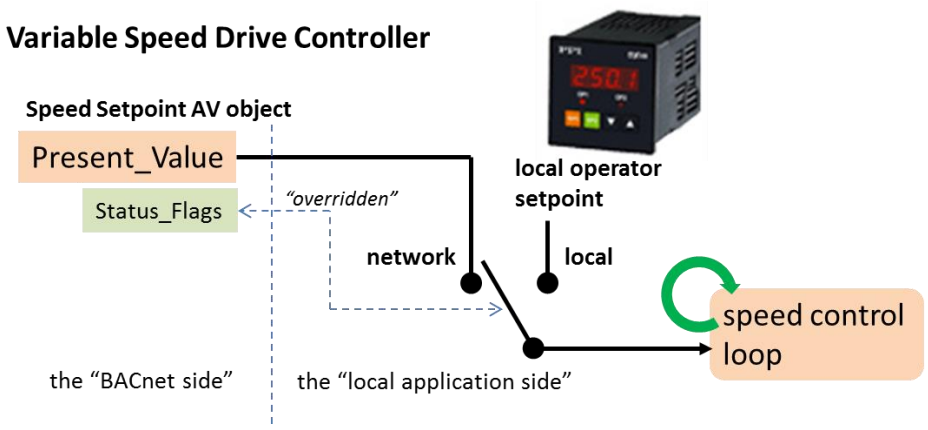


Figure 5 - A "soft decision" Override

In this example a variable speed drive has a local control loop to control drive speed. In its BACnet implementation, the setpoint for the control loop is represented as an Analog Value (AV) object. Normally whatever value is written to this AV Present_Value will be used by the control loop as the setpoint. However, the VSD controller also has a local user interface (UI) display that can be used to override the setpoint. When this happens, a software "switch" within the controller is flipped to a "local" mode that causes the control loop to read its setpoint instead from the UI rather than the Present_Value. BACnet clients can still read and write the Present_Value but in this mode it has no effect on the control loop. In this example, the software switch mode is reflected in the OVERRIDDEN bit in Status_Flags.

Modular Packages of Features for Value Objects

Like other standard BACnet object types, value objects provide implementers with a lot of choices for functionality and features in the objects themselves. One way to think about this is to imagine a *base object* that has zero or more groups of functionality (sub-classing) that may be overlaid onto an instance of a given value object type. We say "instance" because BACnet does not require all of the objects of a given type in a given device to have identical functionality. For example, a device that has four Analog Value objects might treat two of them as input-like and two as output-like. Perhaps only one of the outputs implements commandability, and only one input implements intrinsic reporting. These packages of optional functionality are:

- Writable Out_Of_Service
- Reliability Detection
- Override Detection
- Commandability
- Intrinsic Reporting
- COV Reporting

Writable Out_Of_Service

This package means that an implementer has decided to support a writable Out_Of_Service property and the corresponding logic in the behavior of the object, Reliability and Status_Flags.

Reliability Detection

This package means that the implementer has decided to support the detection of reliability, the Reliability property, and the corresponding logic in the behavior of the object, Reliability and Status_Flags.

Override Detection

This package means that the implementer has decided to support the detection of override, and/or has the hardware inputs needed to detect external override and has implemented the logic needed to reflect this status in the Status_Flags property.

Commandability

This package means that the implementer has decided to implement output-like value objects and has included the logic necessary to implement the behavior of the Priority_Array and Relinquish_Default properties, and Present_Value.

Intrinsic Reporting

This package means that the implementer has decided to implement intrinsic reporting features, accordingly to the input or output likeness of the object. This includes not only a collection of properties for managing event detection and reporting, but also support for Notification Class objects and EventNotification services.

COV Reporting

This package means that the implementer has decided to implement change-of-value reporting features, accordingly to the input or output-likeness of the object. This includes not only a collection of properties for managing change detection and reporting, but also support for Notification Class objects, SubscribeCOV and COVNotification services.

Numeric Value Objects

As a group it is helpful to think of Analog Value, Integer Value, Large Analog Value and Positive Integer Value objects as simply *numeric value objects*. They have nearly identical behavior but differ from each other mostly in the datatype of their Present_Value properties. All of the numeric value objects represent the same concept: a range of values between some minimum and maximum that can represent input-like or output-like behavior.

Why do we need different datatypes? Mostly this is a matter of precision. If we have some value, such as a temperature, we may choose to implement the conversion of a sensor's reading into whole degrees (72, 73, 74 degrees etc.) or into floating point (REAL) numbers (74.3 degrees etc.) Depending on the accuracy of the sensor and measurement hardware, we may not be able to achieve more than a specific amount of precision. For example, we may only be able to be accurate to a tenth of a degree, but not a hundredth of a degree. If we are counting pulses from a kilowatt-hour meter and converting them into kilowatt-hours, we may have limited precision depending on the magnitude of the converted number. For example, REAL numbers, although they can convey a huge range of magnitude (from 10^{-38} to 10^{+38}) their *precision* is limited to between 6 and 9 digits depending on the number. Usually this is more than enough, but there are scenarios where a REAL isn't precise enough and we need instead to use a DOUBLE. Integer values can represent positive and negative numbers, but sometimes we want to explicitly restrict a value to only zero or positive values.

In BACnet the implementer can choose which datatype is most appropriate for each application and match that need to an appropriate value object.

Don't confuse the BACnet client-visible representation of a number value with the internal storage for that number! It is common and perfectly allowed in BACnet to store numbers internally say as 16 bit words but convert them to REAL when they are read as object properties. You could even have an output-like Analog Value that stores the Present_Values internally as bytes. If you write to that Present_Value with 43.52 it might get stored internally as 43 instead. When read back, you would read a 43.00 and that's OK in BACnet.

The object type of a value object determines the datatype that it can accept and return for its Present_Value. Analog Value uses datatype REAL. Large Analog Value uses datatype DOUBLE. Integer Value uses datatype Integer. Positive Integer Value uses datatype Unsigned.

object type	*PV datatype*
Analog Value	REAL
Large Analog Value	DOUBLE
Integer Value	Integer
Positive Integer Value	Unsigned

All of the numeric value objects have some properties in common. The Min_Pres_Value and Max_Pres_Value properties represent the minimum and maximum values respectively that Present_Value can return based on the scaling and conversion in effect for that object. The Low_Limit and High_Limit properties represent a range of values that are normal for the object and values outside that range may cause event notifications when intrinsic reporting is implemented and enabled for the object. For output-like numeric value objects, the Relinquish_Default is the default value to use when Present_Value has not been commanded yet, or all slots of its Priority_Array are NULL. The Resolution property indicates the minimum resolution for step changes in Present_Value. This group of properties all share the same

datatype as the Present_Value. The Deadband property used in intrinsic reporting, and the COV_Increment property used in Change Of Value reporting, would also share this datatype. The exception are Integer and Positive Integer value objects where Deadband and COV_Increment are always Unsigned.

Dates, Times and DateTimes

There are many scenarios in building automation where we need to convey date and time values. For example, scheduling activities for certain dates, or changing a value or setpoint throughout the day at certain times. Sometimes we need both a date and a time. For these purposes, BACnet provides value objects that can take on Date, Time or combined DateTime values.

In BACnet, a Date is composed of four parts:

- the year from 1900 to 2154
- the month of the year from 1=January to 12=December
- the day of the month from 1 to 31
- the day of the week from 1=Monday to 7=Sunday

Yes, BACnet intentionally has a "Y2154" problem. The common thinking is that if one is still using BACnet 138 years from now, it would be a good time to switch to something new whether you need to or not. Generally speaking, these four-part dates are called *specific dates*.

In BACnet, a Time is composed of four parts:

- the hour from 00 to 23
- the minute from 00 to 59
- the second from 00 to 59
- the hundredths of a second from 00 to 99

These four-part times are called *specific times*.

A DateTime is simply a Date and a Time packaged together. This can also be called a *specific datetime*.

BACnet provides three value objects who's Present_Value can represent these kinds of values: Date Value, Time Value and DateTime Value. It's important to note that these object types may only be used for "specific" values.

Date and Time Patterns

Sometimes it's useful to be able to have components of Dates or Times that are *unspecified*. This is particularly useful in Schedules. An unspecified value is also sometimes called "Any". For example, the first day of every month in 2016 might be a Date whose components are (2016,Any,1,Any). The "top of the hour" might be a Time whose components are (Any,00,00,00). Dates, Times and DateTimes that contain unspecified components are called *Date Patterns, Time Patterns and DateTime Patterns* because they specify patterns of periodic repetition.

In addition to being able to use unspecified components, Date Patterns and DateTime Patterns may also make use of what are called *special date values*.

In the encoding of Dates, including Dates that are part of DateTimes, the value 0xFF (255) is used to indicate that a particular component of the Date is unspecified. For the Month component of any Date, the special value 13 means odd numbered months, and the special value 14 means even numbered months. For the Day of Month component of any Date, the special value 32 means the last day of the month regardless of the number of actual days in the month. The special value of 33 means odd numbered days of the month and the special value 34 means even numbered days of the month. Taken together any Date, or DateTime, that contains either unspecified components and/or special date values is considered to be a Date Pattern and any Time containing unspecified components is considered to be a Time Pattern.

Date Value, Time Value and DateTime Value objects are required to only use specific values and may not contain patterns, i.e. unspecified or special date value components.

Date Pattern Value, Time Pattern Value and DateTime Pattern Value objects may contain patterns.

Feature Packages for Date, Time, DateTime, Date Pattern, Time Pattern and DateTime Pattern Objects

None of these objects may support Intrinsic Reporting or COV Reporting.

String Value Objects

The numeric, date, time and datetime values have a common characteristic: they are all *scalar values*. However, there are many circumstances when we need to represent more than one discrete number. For example, names of things require a sequence, or *string*, of letters, numbers and symbols. The card number of an access control card requires a string of numbers. Representing a group of binary statuses might require a string of bits, etc. In BACnet there are datatypes for representing *characterstring, octetstring and bitstring* data. Although properties of BACnet objects may use these datatypes, there are also many use cases for value objects that can represent these datatypes as well.

OctetString Value Objects

Perhaps the simplest of the string value objects is called *octetstring*. It is used to represent a sequence of *octet* values. A very common concept in all computer-based devices is the grouping together of eight binary bits into a single entity known as a *byte*. This term hasn't always meant a group of eight bits, although it is rare today to see it used otherwise. Within the community of international communications standards, rather than relying on made-up words that have no basis in any language, the term byte has been replaced with the term *octet* from the Latin root *oct* meaning "eight." If we accept the fact that byte nearly always means eight bits, then conceptually, byte and octet represent the same idea.

A single octet can therefore represent values in the decimal range 0 to 255. The odd looking value 255 comes from the fact that with eight binary bits, the largest number that can be represented would be eight "ones" in a row 11111111_2 which is the value $(2^8\text{-}1)$ or 255_{10}.

An *octetstring* contains some sequence of octet values that are treated together as a group. Technically, BACnet Date, Time and DateTime datatypes are octetstrings whose four and eight octets are always interpreted in a specific manner.

An OctetString value has a Present_Value whose datatype is an octetstring.

BitString Value Objects

A *bitstring* value object is used to represent a sequence of single bit true/false values that are always taken together as a group. A BitString value has a Present_Value whose datatype is a bitstring. In much the same way that Multi-state values have both numeric state values and a characterstring "name" for each state, BitString values can have a name for each bit. The Bit_Text property of BitString Value objects is an array of characterstrings each representing the name for the corresponding bit. The relationship between the bits and the Bit_Text array entries may seem a bit counter-intuitive and deserves further explanation.

In BACnet, bitstring bits are "backwards" from the way that programmers usually think of them. Consider the decimal value 13_{10} which in binary would be 1101_2. The "bottom-most" or right hand bit represents 2^0 or the value 1. The "top-most" or left hand bit represents 2^3 or the value 8. When programmers think of these bits they usually use the power of 2 to number the bits, so the right hand bit is "bit 0" and in this example the left hand bit is "bit 3". However, in BACnet bitstrings are organized so that bit 0 is the "bottom-most" but appears "first" in the stream of bits in the string. Another way of thinking about this is that in BACnet bit 0 is the left hand bit and occupies the most significant bit within an octet:

bit #	0	1	2	3
	1	0	1	1

As a practical matter, bitstrings are always conveyed on the wire in a sequence of octets. One way to think about this is to imagine that the bitstring is layed out from left to right with the left-most bit representing bit 0. This bit is also "shifted" to be the most significant bit of the first octet (the hexadecimal 0x80 bit of the octet value). When the bitstring length is not an even multiple of 8 bits, the right-most (last) octet contains U unused bits in the least significant bit positions of the octet, where U=(8-(bitstring length in bits)) modulo 8. The encoding of bitstring values conveys the number of unused bits U as well as the bitstring itself.

The "bit #" is important because it has a relationship to the Bit_Text array property. The name for bit #N is contained in the Bit_Text[1+N] array slot. In the example above, suppose that bit 0 has the name AAA, bit 1 has the name BBB, bit 2 has the name CCC and bit 3 has the name DDD. The Bit_Text array would look like this:

Bit_Text[0]	4
Bit_Text[1]	"AAA"
Bit_Text[2]	"BBB"
Bit_Text[3]	"CCC"
Bit_Text[4]	"DDD"

CharacterString Objects

A *characterstring* value object is used to represent a string of text composed from letters, digits, symbols etc. The text is stored in a sequence of octets where generally speaking each octet represents a code for a particular character symbol. The original 1995 BACnet standard proposed that characterstrings could contain symbols from one of several possible *character sets* defined by national and international standards. The most common of these was the so called "ANSI" character set from ANSI x3.64 which is a superset of the venerable ASCII character encoding. Another option was so-called "Unicode" or ISO10646 and various others. In Addendum k to 135-2008 this was changed to allow the most common character set to be UTF-8. The UTF-8 encoding allows single octet representation of many popular letters and symbols,

but also allows multi-octet encoding to accommodate Asian symbols, mathematical symbols and many other less common symbols (even ancient Sanskrit and Klingon!).

The key point is that BACnet characterstrings can represent even the most obscure kinds of text and symbology. In practice of course mostly common human language text will be used. But the single difference between characterstrings and octetstrings is that characterstrings use the first octet as a code to represent the type of character set that is used in the string that follows. Overwhelmingly today, this octet is always zero (meaning UTF-8 characterstring). It's worth noting that several of the older character sets used 2 or 4 octets *per symbol* to encode characters.

Feature Packages for OctetString, BitString and CharacterString Value Objects

None of these objects support COV Reporting.

OctetString value objects do not support Intrinsic Reporting.

BitString and CharacterString value objects optionally support Intrinsic Reporting. In this case the objects include Alarm_Values and Fault_Value properties that define specific bitstrings or characterstrings that represent "Alarms" or "Faults."

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