OBJECT ORIENTED FORTH AND BUILDING AUTOMATION CONTROL

Bradford J. Rodriguez, Ph.D.

T-Recursive Technology, Collingwood, Ontario, Canada

ABSTRACT

Building Automation and Control networks are widely adopting BACnet(tm), a standard developed by the American Society of Heating, Refrigerating, and Air-Conditioning Engineers. This standard describes an object-oriented model for control devices, and a network protocol for communicating with these devices. We have implemented the functionality of BACnet in a cross-compiled object-oriented Forth, by allowing nested objects and removing encapsulation. Our system greatly simplifies the task of the application programmer wishing to define new control objects or data types for the network environment.

BACnet(tm)

BACnet(tm) refers to the Building Automation and Control Network standard [ASH95]. The American Society of Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE), envisioning a new generation of distributed and networked electronics in commercial buildings, has promulgated this standard to promote connectivity and interoperability between different manufacturers' control systems and networks.

The control devices employed by this industry range from simple light switches to complicated HVAC (Heating, Ventilation, and Air Conditioning) controls and security systems. BACnet attempts to encompass the full range of control devices by modeling them as *objects*, each of which may have an assortment of *properties*. For example, a temperature sensor may be considered an object of type *Analog Input*. Such an object will have many properties, such as *Present Value, Units, Resolution,* and *Status*.

Several standard object types are defined in the BACnet standard. For each object type, many standard properties are defined: some required, some optional. (All *Analog Inputs*, for example, are required to have a *Present Value*.) All BACnet object types and properties are assigned a numeric identifier: e.g., *Binary Output* is object type 4, and *Present Value* is property number 85. Additional manufacturer-specific object types may be defined, and manufacturer-specific properties may be added to standard objects.

While the standard does not speak in such terms, it seems likely that the designers of BACnet had in mind a "frame-based" object-oriented programming model. The properties of each object are intended to be visible. In fact, the properties of each object are specifically required to be accessible over a network. Much of the BACnet standard is devoted to specifying exactly how properties are to be transmitted over the network, and what services are available to act upon properties.

Property types and services

Properties may have simple data types, such as *Signed Integer, Unsigned Integer, Real* (floating-point), *Text, Octet String*, or *Bit String*. A given property may also contain a *list* of a given data type, or an *array* of a given data type. In addition to enumerating these primitive data types, the BACnet standard specifies the presentation of these data types over the network. This uses the ASN.1 "Abstract Syntax Notation," in which every data item is transmitted with a tag (identifying the data type), a length, and a value. For example, the unsigned integer 1234 hexadecimal is formatted as bytes 22, 12, 34, where the first byte contains both the tag and length.

BACnet also specifies which services are available over the network. The most common of these -- and the most significant to this paper -- are the *ReadProperty* and *WriteProperty* services, which allow property values to be remotely examined or altered. Supervisory services allow other actions, such as creating a new object (in controllers which allow this). All of the BACnet services are identified by assigned numeric codes; additional codes are reserved for manufacturer-specific services.

The physical network

BACnet does **not** define the network medium used to communicate between control devices. The standard's sole concern is the presentation and interpretation of control data. It essentially addresses the layers above the Network layer in the ISO 7-layer networking model (although BACnet rejects the 7-layer model in favor of its own "condensed" model).

The specification does give recommended mappings of BACnet to Ethernet and LONtalk networks. It also describes a protocol for a new low-cost network, called the MS/TP (Master-Slave Token-Passing) network, which can be carried on simple asynchronous serial ports using RS-485 multidrop drivers. Although a detailed description of this protocol (including a state machine implementation) is supplied, it is emphasized that support for this protocol is not required in a BACnet implementation.

MAPPING BACnet TO OBJECT ORIENTED FORTH

For our networked control project, we wished to adapt from prior art as much as possible. But most previous implementations of object-oriented Forth embodied the "traditional" object model, rather than a frame-based model. [ROD96] In the traditional model (of which Pountain [POU87] is a representative example), objects are instances of a class, and the "instance variables" of an object are *encapsulated*. In other words, the instance variables are available **only** to the methods of the object, and are hidden from external view. This conflicts with the BACnet model in which the properties of an object are available both over the network, and to the application programmer.

Rather than develop a frame-based object-oriented-Forth (OOF) from scratch, we set out to modify the "traditional" model so that it would be suitable to represent BACnet objects. We found that only two extensions were required:

> **1. Instance variables are not hidden.** We "defeat" the encapsulation offered by most OOFs, by allowing the application program to directly access the instance variables of an object. I.e., it is not necessary to use one of the object's methods, in order to fetch or store an instance variable. This weakens one of the protections offered by object-oriented programming, in that access to instance variables is now relatively uncontrolled (and undesirable side effects may creep in).
>
> **2. Instance variables may be objects.** In other words, rather than being a primitive data item, an "instance variable" may itself be an object of another class. This is not unique to our OOF implementation; other OOFs have supported "nested objects" (objects within objects) [POU87]. But given that direct access to instance variables (properties) is permitted, this provides a "clean" way for the application program to reference the instance variables: by invoking *their* methods.

Representing properties as nested objects helps to restore some of the protection lost when encapsulation was discarded: application programs may directly access instance variables, but that access is limited to a small and controlled set of methods. This also helps to preserve the "programmer's world view" of object and method.

Syntax

Thus a BACnet *property*, in our programming language, is an object with methods, that also happens to be an instance variable of another object. A BACnet control object is *also* an object with methods, whose instance variables are its properties. In each case, the methods are associated with the *class* of the object (all objects of the same class share the same methods). The programmer may apply a method *selector* to an object, or to a property of an object.

To apply a method to an object, the syntax is

```
object selector
```

and to apply a method to a property of an object (an object within an object), the syntax is

```
object property selector
```

This syntax is extensible to any level of nesting (objects within objects within objects), e.g.,

```
object property ... property selector
```

Thus the selector delimits the list (nest) of properties and sub-properties. This was the compelling reason for our choice of "object selector" syntax for our OOF, rather than the equally common (and hotly debated) "selector object" syntax. (Indeed, this is the first compelling argument we have encountered to favor one syntax over another; the "selector object" syntax reads confusingly, and is awkward to implement.)

Implementation of this syntax is straightforward. When an object is referenced, it leaves a data address (of its instance data) and a class identifier on the stack. A method selector will search that class for the requested method, and call that method (a Forth word) with the data address. When a property is referenced, it expects to find a data address and class on the stack. It modifies that address as needed to point to the address of the particular instance variable (nested object); then substitutes its own class identifier on the stack. I.e., the stack effects are

Object: `( -- data class )`

Property: `( data class -- data2 class2 )`

Selector: `( i*x data class -- j*x )`

This of course allows nested properties to successively modify the data address (and successively replace the class identifier), as deeply as required. The method selector is bound using the last class left on the stack, that is, the most deeply nested property. Note that the method may consume a number of items, and/or leave a number of items, on the stack.

Overloading and binding

Selector names for methods may be overloaded. Indeed, it is a requirement that all classes support a certain standard set of methods.

The default is to use late binding. The Forth word corresponding to a selector is determined at run time, when the selector word is executed. In the current implementation, each class maintains a linked list of selector codes and method addresses, and the method is found by a simple linear search. This is less efficient than early binding, since each method execution requires a search; but the overhead is minimized through the use of a very fast search routine (five machine instructions per list entry) and relatively short method lists (most classes have fewer than eight methods).

Early binding *is* supported for application programs. However, in the present implementation, the programmer must explicitly specify the class from which he wants the method to be bound. The syntax for this is (in the general case)

```
object property ... property BIND class selector
```

For example, if the programmer knows that a given property is of class `INT` (integer), he can write `BIND INT FETCH` to force early binding of the `FETCH` method. The disadvantage is that, should the property be changed to a different class (say, a `LONG` integer), **every** occurrence of this phrase for this property must be changed to `BIND LONG FETCH`. It would be vastly preferable to have `BIND` automatically determine the class of the most recently specified object or property.

Property names may **not** be overloaded. Because they are globally visible, and they may be defined differently in different classes, we currently require that all properties have unique names. This is usually accomplished with a prefix, such as `AI.PRESENT.VALUE` for the *Analog Input* class.

STANDARD CLASSES

We have defined a library of standard classes for the application programmer. They are of two kinds: *primitive* classes, which are used to define properties, and *application* classes, which are used to define the actual control objects.

Primitive classes

The primitive classes correspond roughly to the *data types* defined by the BACnet standard. Among the primitive classes are

`BYTE` 8-bit (unsigned) integer

`INT` 16-bit (signed) integer

`LONG` 32-bit (signed) integer

`REAL` 32-bit floating-point number

`TEXT` ASCII character string

These are normally used within the data declaration portion of an application class, to define its instance variables (properties). However, they can certainly be used to define actual objects, such as an integer or real "variable," independent of any other object definition.

All primitive classes are expected to have `FETCH` and `STORE` methods, as well as the `READ` and `WRITE` methods required for network access (more on this shortly).

Application classes

Application classes correspond to the *object types* defined by the BACnet standard. Among the application classes are

`ANALOG.INPUT` e.g., an analog-to-digital converter input

`ANALOG.OUTPUT` e.g., a digital-to-analog converter output

`ANALOG.VALUE` a computed or derived analog value

`BINARY.INPUT` a bit input

`BINARY.OUTPUT` a bit output

Some of the application objects are used for application programming; others have only a network presence (i.e., they are intended to be read remotely, but have no local function). The supported methods depend upon the object type; e.g., an `ANALOG.INPUT` will have a `FETCH` but not a `STORE` method.

STANDARD METHODS

As noted above, expected behaviors are defined for several "standard" methods. These methods appear in most (if not all) classes.

STORE and FETCH

`STORE` and `FETCH` are supported for application classes (control objects) and for properties. These are used for application programming, and their stack effect depends upon the particular class. For example, the `FETCH` for class `INT` will leave one cell on the stack, while the `FETCH` for class `LONG` will leave two cells on the stack.

READ and WRITE

These are the key methods used to make objects -- specifically, the properties of objects -- visible on the network. They embody the functions of the BACnet *ReadProperty* and *WriteProperty* requests, and accordingly are defined only for properties (i.e., for primitive data types). These methods are used exclusively by the network service code; not for application programming.

The function of the `READ` method is to fetch the value of the property, and to encode that value into the BACnet ASN.1 notation described earlier. The network service routines can then return that formatted value to the external device that requested it. Similarly, the function of the `WRITE` method is to decode a value which has been supplied in the BACnet format, and to store the result into the given property. `READ` and `WRITE` translate from internal representation to the "network" ("external", or BACnet) representation.

`READ` and `WRITE` illustrate the power of representing properties as objects. As long as the application programmer restricts himself to the predefined data types -- `INT`, `BYTE`, etc. -- he can define an entirely new class of control object, and that class is *automatically* accessible on the network. The programmer need do no network programming at all! All of the code required to remotely access the properties of the new object are contained within the methods of the primitive classes.

Similarly, should the application programmer need to define a new data type (primitive class), he can make that data type network-visible by simply writing a `READ` and `WRITE` method. No other network programming is involved. The network service code requires no modification: any property having these methods can be read and written over the network.

Thus the `READ` and `WRITE` methods greatly facilitate the purpose of this object-oriented Forth, which is to support, simplify, and encourage the creation of networked control objects, in the spirit of the BACnet specification.

SCRUB and INITIALIZE

These two methods solve the problem of object initialization. This is a particular concern in control systems, where a processor reset or a power failure may leave an output in an incorrect or undefined state.

`SCRUB` is a "cold start" method. It is executed for all primary objects when the system is first activated, or anytime a processor reset occurs and the RAM contents are invalid. Its primary purpose is to initialize RAM and all outputs to a known "safe" state.

`INITIALIZE` is a "warm start" method. It is executed for all primary objects when a processor reset occurs and the contents of RAM are still valid. This may be due to a watchdog reset, a manual reset, or a power outage short enough to preserve the battery-backed RAM. This method does not initialize RAM, since the program wishes to resume from its previous state. But it should reinitialize outputs, and re-output the last known value, since these are generally volatile when power is lost.

Again, these methods are provided to ease the task of the application programmer. He never needs to need modify the system initialization code; he only needs to ensure that his control objects (or other program objects) have a suitable `SCRUB` and/or `INITIALIZE` method.

OBSERVATIONS

As noted previously, this object-oriented system was developed for a *cross-compiled* application. We used Forth Inc.'s SwiftX(tm) cross-compiler for the Motorola 68HC12, and we have been very pleased with this compiler. (It is this author's personal opinion that this "cross-OOF" could have been written in **no** other commercially-available Forth cross-compiler).

We have encountered the limitations of cross-compiled Forth a few times. One limitation is that only limited object services are available at compile time. SwiftX has a very flexible memory model, allowing the target memory to be examined and altered even when acting as a pure cross compiler (with no target hardware connected). But the stack effect of objects is not properly simulated in "cross-compiler" mode, so objects can only be examined interactively when connected to the target hardware (via the SwiftX cross-target link). Forth Inc. has been very responsive to our suggestions, and we hope to rectify many of these minor annoyances in a future revision of this system.

REFERENCES

[ASH95] American Society of Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE). BACnet(tm): A Data Communication Protocol for Building Automation and Control Networks, ANSI/ASHRAE Standard 135-1995. Atlanta, GA: American Society of Heating, Refrigerating, and Air-Conditioning Engineers, Inc., 1995.

[POU87] Pountain, Dick. Object-Oriented Forth. London: Academic Press, 1987.

[ROD96] Rodriguez, Bradford J., and W. F. S. Poehlman. "A Survey of Object-Oriented Forths." ACM SIGPLAN Notices 31, no. 4 (April 1996): 39-42.