

NIST Technical Note
NIST TN 2388

BACnet Device Emulator Users' Guide

Michael A. Galler
Amanda J. Pertzborn
Parastoo Delgoshaei

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.TN.2388>

NIST Technical Note
NIST TN 2388

BACnet Device Emulator Users' Guide

Michael A. Galler
Building Energy and Environment Division
Engineering Laboratory

Amanda J. Pertzborn
Building Energy and Environment Division
Engineering Laboratory

Parastoo Delgoshaei
Building Energy and Environment Division
Engineering Laboratory

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.TN.2388>

October 2026



U.S. Department of Commerce
Howard Lutnick, Secretary

National Institute of Standards and Technology
Arvind Raman, NIST Director and Under Secretary of Commerce for Standards and Technology

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

The opinions, recommendations, findings, and conclusions in this publication do not necessarily reflect the views or policies of NIST or the United States Government.

NIST Technical Series Policies

[Copyright, Use, and Licensing Statements](#)

[NIST Technical Series Publication Identifier Syntax](#)

Publication History

Approved by the NIST Editorial Review Board on 2026-09-30

How to Cite this NIST Technical Series Publication

Galler M. A., Pertzborn A. J., Delgoshaei, P. (2026) BACnet Device Emulator Users' Guide. National Institute of Standards and Technology, Gaithersburg, MD, NIST Technical Note (TN) NIST TN 2388.
<https://doi.org/10.6028/NIST.TN.2388>

Author ORCID IDs

Michael A. Galler: 0000-0003-2303-8274

Amanda J. Pertzborn: 0000-0002-1473-7500

Parastoo Delgoshaei: 0000-0003-1182-4905

Contact Information

Michael A. Galler: mikeg@nist.gov

Abstract

The BACnet Device Emulator (BDE) was created at the National Institute of Standards and Technology to support the development of digital twins for the built environment. The BDE creates emulated BACnet subnets and controllers that can act as a stand-in for real physical BACnet devices in a variety of applications. At NIST, the Intelligent Building Agents Laboratory (IBAL) contains real heating, ventilating, and air conditioning (HVAC) equipment, but all control signals and sensor data are connected to a laboratory data acquisition system. The BDE allows the IBAL to implement control logic in a manner that is more consistent with real buildings, and to enable the demonstration of digital twin technologies more consistently with how they would be used in real buildings.

The BDE also enables benchtop development and testing to ease the deployment of a digital twin. For example, historical data can be used in place of the physical twin (e.g., a building), and the BDE can be used in place of the real physical controllers, to provide data to the software components of the digital twin that are under development. When the digital twin is later deployed in the real building, the transition from the BDE to the real physical controllers will be simple.

This report describes the BDE, how to access and use it, and provides examples that illustrate its use in two different parts of a digital twin application.

Keywords

HVAC, digital twins, building emulator, BACnet, semantic knowledge graphs

AI Disclosure

Some of the Python code snippets in this report were produced with the assistance of Gemini 3.6 and Claude Opus 4.7.

Table of Contents

1. Introduction to the BACnet Device Emulator	1
2. Installing the BDE	4
2.1. Install Npcap:.....	4
2.2. Using the included executable file:	4
3. Overview of the BDE Graphical Interface.....	5
4. Creating and Configuring Emulated Controllers	6
4.1. Determining the Configuration	6
4.2. About Templates	6
4.3. Sequence of Operations Logic Library	6
4.4. Adding Subnets	6
4.5. Adding Controllers	7
4.6. Saving Templates	8
4.7. Deleting Controllers	8
4.8. Adding Objects to Controllers	9
4.9. Viewing Controller Data	10
4.10. Modifying Property Values	11
4.11. Saving and Loading Configuration Files	11
4.12. Finishing Steps Before Use.....	11
5. Example Use Cases	13
5.1. IBAL Use Case with LabVIEW.....	13
5.2. Use Case with Python.....	15
5.3. Use Case with Knowledge Graphs.....	16
6. Conclusions and Future Work	18
Appendix A.	19
A.1. IBAL Example with LabVIEW	19
A.1.1. AdapterInit	19
A.1.2. SetDefaultAdapterSLI	19
A.1.3. ReadPVPropertyByDeviceSLI	20
A.1.4. WritePVPropertyByDeviceSLI	21
A.2. Use Case with Python	22
A.2.1. Load the DLL	22
A.2.2. Initialize and Set the Adapter	23
A.2.3. Create the Read and Write Objects	24
A.2.4. Functions for Read and Write	25

A.2.5. Execute the Code	26
A.3. Use Case with Knowledge Graphs	27
References.....	29

List of Tables

Table 1. List of LabVIEW function calls.	14
Table 2. Description of the key parameters for the read and write functions.	15

List of Figures

Fig. 1. NIST’s conception of a digital twin.	2
Fig. 2. The Npcap Installation Options Dialog	4
Fig. 3. The BACnet Device Emulator interface.....	5
Fig. 4. The Add New Network Window	6
Fig. 5. The Add Controller Window	7
Fig. 6. Adding Multiple Controllers.....	8
Fig. 7. Adding a Template	8
Fig. 8. The Delete Controller Window	9
Fig. 9. The Add New Object Window.....	9
Fig. 10. Changes to New Object	9
Fig. 11. Objects Added to Controller	10
Fig. 12. Object and properties shown in Focus Window	10
Fig. 13. The Modify Property Value Window	11
Fig. 14. The Modified Model Name	11
Fig. 15. Sending a BACnet I-Am message.....	12
Fig. 16. Data flow for the digital twin to physical twin interaction in the IBAL.	13
Fig. 17. Steps to create the emulated BACnet network for the IBAL use case.....	14
Fig. 18. LabVIEW screenshots of the AdapterInit function call.....	19
Fig. 19. LabVIEW screenshots of the SetDefaultAdapterSLI function call.	20
Fig. 20. LabVIEW screenshots of the ReadPVPropertyByDeviceSLI function call.....	21
Fig. 21. LabVIEW screenshots of the WritePVPropertyByDeviceSLI function call.....	22

1. Introduction to the BACnet Device Emulator

Digital twins (DT) technology is being developed across a wide range of industries [1][2]. According to the National Academies of Sciences, Engineering, and Medicine definition [3], a DT has several key features:

- a virtual representation of a physical system,
- dynamic updating of the virtual representation using data from the physical twin (PT),
- contextual behaviors and structural representation, and
- a predictive capability.

The DT can assist with decision-making, and the bidirectional flow of information between the PT and DT is essential. In the building domain, DTs can be used for a variety of purposes, including optimal control of HVAC equipment, fault detection and diagnostics, improved grid operation, and urban planning. However, the widespread development and deployment of DTs is limited by some significant challenges. At the National Institute of Standards and Technology (NIST), we focus on helping industry develop the tools required to create DTs to improve the built environment.

Fig. 1 presents the details of a DT as currently conceptualized at NIST; it includes the key features outlined above. Microservices are tools that provide predictive capability and either automated or assistive decision-making. The microservices interact with the virtual representation, which is composed of various models (e.g., EnergyPlus, machine learning, Building Information Model (BIM)), a procedure to dynamically update the models from the PT (i.e., ongoing model calibration), application programming interfaces (APIs) to interact with microservices and the PT, and semantic knowledge graphs. The details of a specific DT application will determine which microservices are required and what services they provide to the PT and/or the user.

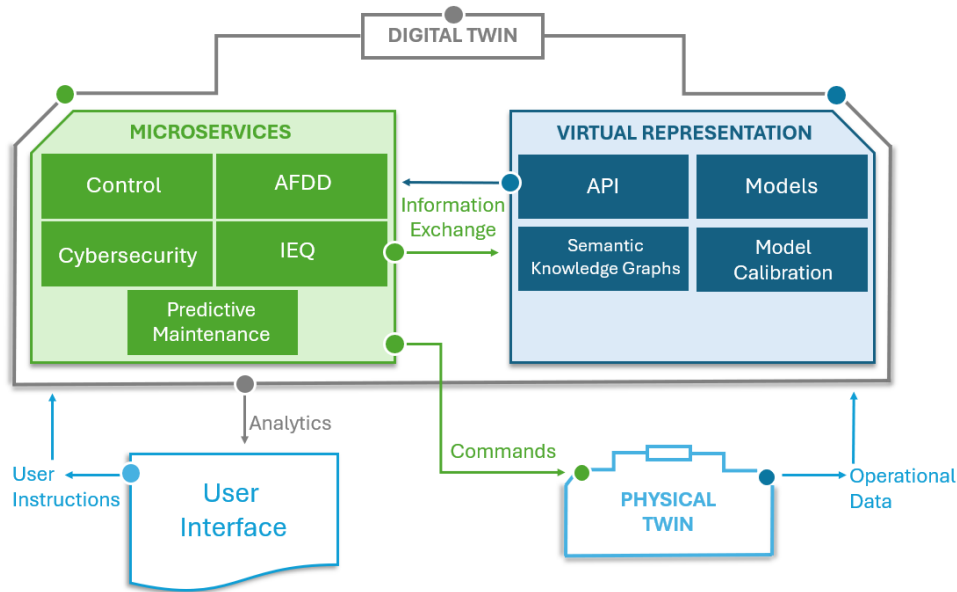


Fig. 1. NIST's conception of a digital twin.

Semantic knowledge graphs are at the core of enabling DTs. They present information about the building systems in machine-readable, portable, and scalable graphs that can grow and be updated as information changes. ASHRAE Standard 223P [4] (S223), *Semantic Modeling for Building Monitoring and Control Applications*, standardizes these semantic graphs for building systems. It includes definitions of equipment (e.g., air handling unit, chiller) and sensors (e.g., temperature, pressure), how those equipment and sensors relate to one another, and information about how to access data. The graph is queryable and can be formally validated for compliance checking. S223 enables interoperability between disparate systems by providing a common language, and it connects with existing ontologies such as BRICK and Real Estate Core.

The Intelligent Building Agents Laboratory (IBAL) is being used at NIST to develop and demonstrate the technologies and tools for DTs in the built environment. The IBAL (the PT) contains a mix of heating, ventilating, and air conditioning (HVAC) equipment, including air handling units (AHUs), variable air volume terminal units (VAVs), and chillers. In a real building, a typical configuration is for a Building Automation System (BAS) to be connected to BACnet controllers. These controllers send control signals to the equipment and collect sensor data, and all this information can be accessed from the BAS. However, the equipment in the IBAL uses a laboratory data acquisition system instead of a BAS and BACnet controllers. Since we want to emulate the real world as closely as possible in the IBAL, we have created the BACnet Device Emulator (BDE) to play the role of BACnet controllers. The BDE will act as the interface between the simulation (i.e., model) of the laboratory, control logic (i.e., microservice), and the equipment (i.e., PT). See Section 5.1 for additional details on this application.

In general, the BDE creates an emulated BACnet network of one or more controllers that can be used in place of a real system. Other software (e.g., Python, LabVIEW, commercial BAS) or controllers (e.g., real BACnet controllers) on the physical network can interact with the emulated controllers as if they were real. For the IBAL DT, the BDE plays the role of a BACnet

system that does not exist. The BDE can also be used for benchtop testing of a DT or microservice before it is deployed in a real system. It allows the software to be tested and developed offline and then readily moved to the real system.

The BDE works within an ecosystem of software tools:

- **BACnet Communications DLL (BCD):** A Dynamic Link Library (DLL) containing functions to read from and write to BACnet devices from other software programs [1]. It is the basis for communications between the other tools.
- **BACnet Data Source (BDS):** User interface to view and interact with BACnet devices on a network [4][6].
- **BDE Test Library (BDE-TL):** DLL containing templates for different BACnet HVAC controllers.

The remainder of this report introduces the BDE design and explains how to use it in detail. Later sections provide specific use cases in the IBAL (Sections 5.1 and 5.2) and with a semantic graph (Section 5.3).

2. Installing the BDE

The BDE requires three components to run:

- The BDE executable “BACnet Device Emulator.exe”;
- The BACnet Communications DLL (BCD); and
- The Npcap library.

The BDE and BCD can be installed from GitHub at <https://github.com/usnistgov/BACnet-Device-Emulator>. The BDE only runs on Windows 10 or Windows 11 operating systems.

2.1. Install Npcap:

Npcap provides packet capture and transmission capabilities to the BDE and BCD by allowing capture and transmission of raw network traffic. The access to the network adapters provided by Npcap is the basis for BACnet communications in the BCD. Follow these steps to install Npcap:

- Download the latest installer from <https://www.npcap.com>
- Run the installer executable. On the “Installation Options” dialog select “Install Npcap in WinPcap API-compatible Mode”

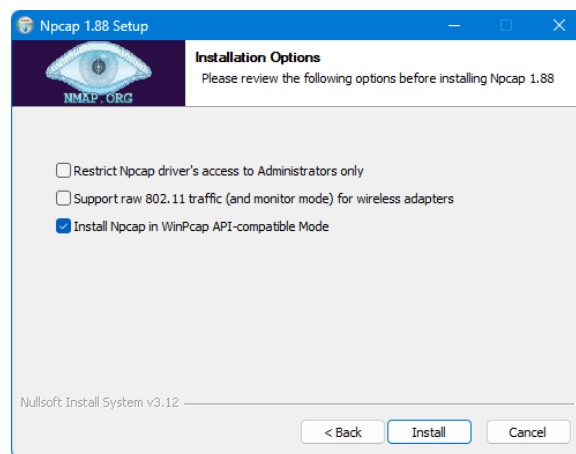


Fig. 2. The Npcap Installation Options Dialog

- Follow the remaining instructions to install Npcap.

2.2. Using the included executable file:

The BDE does not have an installer. Create a new folder on your local drive, for example, “C:\Tools\BDE”. Copy the BDE and the BCD to the new folder. The BDE may be run from this location.

3. Overview of the BDE Graphical Interface

The graphical interface to the BDE is shown in Fig. 3. It is split into four main areas:

- The **Net Tree Window** is a tree control that displays information about the host system, controller templates, and emulated controllers.
- The **Net Tree Status Window** provides status updates on events such as user actions and network activity.
- The **Focus Tree Window** displays detailed information about the controller selected in the Net Tree Window.
- The **Focus Tree Status Window** provides a history of user actions, such as which controllers have been displayed and changes to property values.

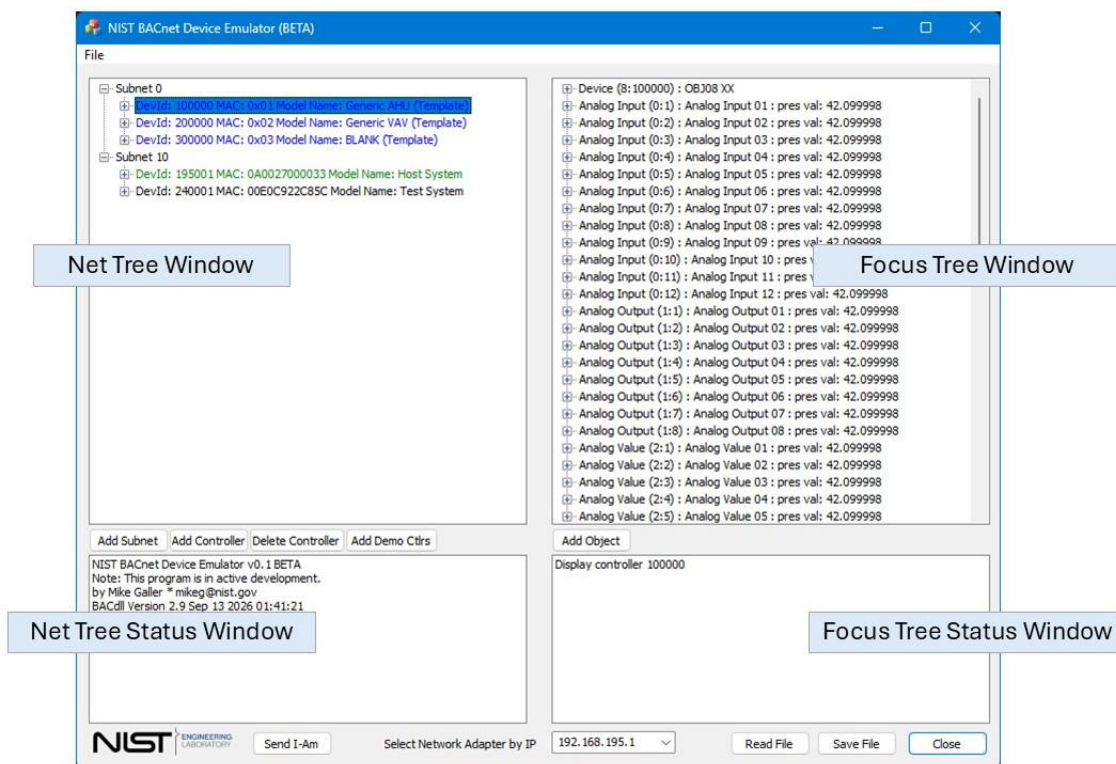


Fig. 3. The BACnet Device Emulator interface

Under the Net Tree window are controls that allow the user to modify the emulated network by adding or deleting emulated subnets and controllers. Under the Focus Tree is a button to add BACnet objects to a controller. On the bottom are controls relating to network communications, including a button to send an I-Am BACnet message and a drop-down menu to select the network adapter.

4. Creating and Configuring Emulated Controllers

4.1. Determining the Configuration

This section describes the steps necessary to use the BDE to build the network and controller configuration you want to emulate. This includes the subnet(s), the number and types of controllers, the objects and property values on each type of controller, and optionally any sequence of operations (SOO) logic.

4.2. About Templates

The BDE has default template controllers that can be used when creating emulated controllers. These are intended to provide an approximate typical representation of common controllers and are populated with various numbers of inputs and outputs depending on the equipment they are intended for. A blank template is also provided, which has only a Device object as is required for all controllers.

4.3. Sequence of Operations Logic Library

The BDE can apply a sequence of operations (SOO) logic library to the emulated controllers. The logic library can be selected when a controller is created. The BDE is provided with a few basic logic libraries for common types of HVAC equipment. Users may develop their own logic libraries using the procedure outlined in [7].

4.4. Adding Subnets

When you have determined the configuration you want to emulate, the next step is to add the subnet(s). Select the “Add Subnet” button under the Net Tree. The “Add Subnet” window shown in Fig. 4 will open. Enter the subnet number in the “Enter new network number” edit box. The interface will not allow duplicate network numbers. The “OK” button will be enabled when a valid subnet number is entered. The Host system is the router for all subnets.

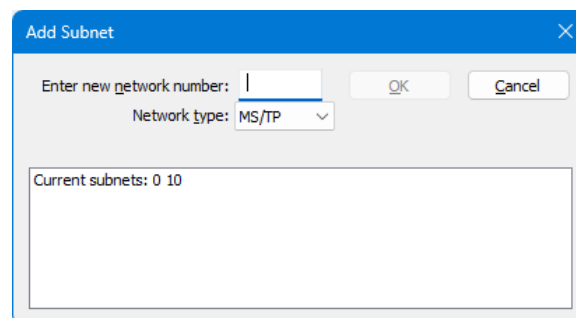


Fig. 4. The Add New Network Window

4.5. Adding Controllers

The next step is to add controllers. Select the “Add Controllers” button. The “Add Controller” window shown in Fig. 5 will open. In this window, the Device Object ID, device MAC address, controller template, subnet, and optional SOO logic library can be selected.

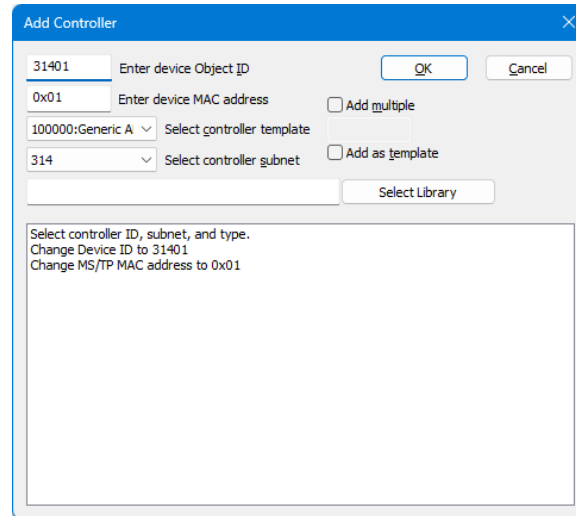


Fig. 5. The Add Controller Window

Enter device Object ID: The Device Object ID must be unique on the emulated network. The BDE will not accept a value that is already used by any other virtual controller. Starting each Object ID with the subnet number helps to keep them unique.

Enter device MAC address: The MAC address must be unique on the subnet, but controllers on different subnets can have the same MAC address.

Select controller template: Select the template to use for the new controller. There is a BLANK option if none of the templates are appropriate.

Select controller subnet: The subnet must be selected from those already created. The interface defaults to the highest numbered existing subnet.

Select Library: This will open a file selection window that allows the user to pick the library file.

Add Multiple: The interface allows multiple instances of a controller to be added in one step. Select the “Add Multiple” checkbox, and an edit box will be enabled. Modify the number of controllers to add from the default of 5 if needed. The Device Object ID and MAC address will be incremented for each additional controller. This is demonstrated in Fig. 6 where five controllers are added. The first figure shows the configuration choices, and the second figure shows the new controllers in the Net Tree display.

Add as template: This will add a new controller template. It can be based on an existing template or blank. It may be modified after it is created. The modifications will be applied to any controllers using this template.

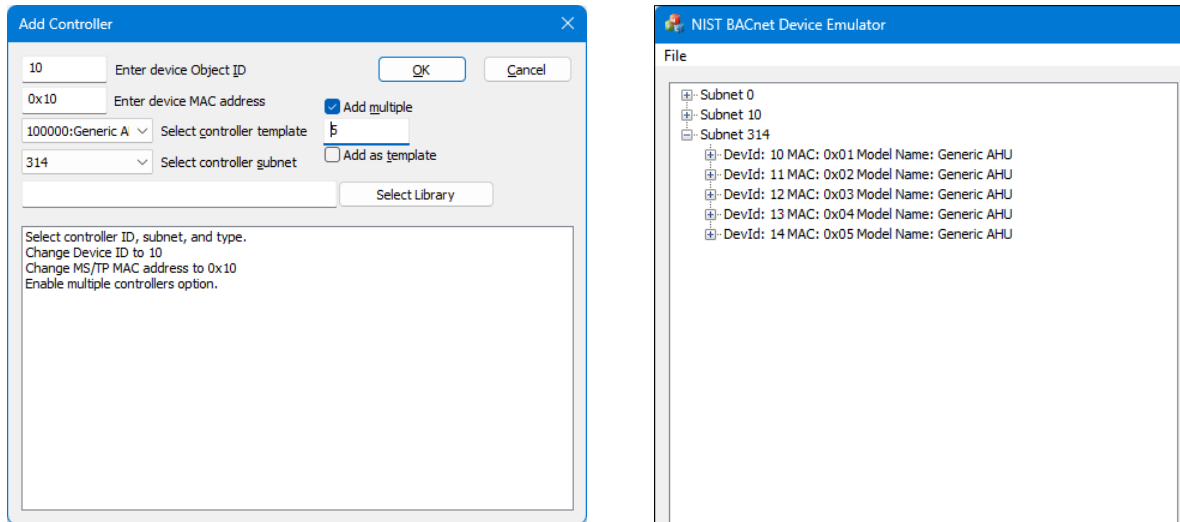


Fig. 6. Adding Multiple Controllers

4.6. Saving Templates

New controllers may be saved as a template by selecting the “Add as template” checkbox. Template controllers are only on subnet 0. Fig. 7 shows how to add a template controller. New template controllers may be modified and used to add new controllers. The model name and other information can be modified as described in Section 4.10.

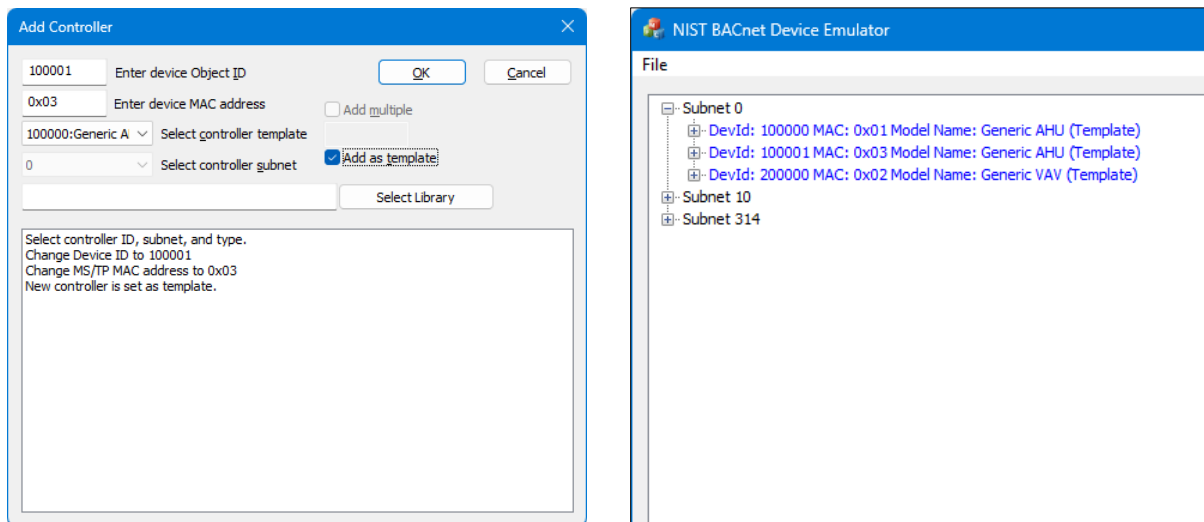


Fig. 7. Adding a Template

4.7. Deleting Controllers

Controllers may be removed by selecting the “Delete Controller” button. This will open a window with a drop-down box with the Device ID numbers from the controllers in the BDE,

excluding Templates and the Host System, as shown in Fig. 8. Select the identifier for the controller to be deleted and select the “Delete” button to remove the specified controller.

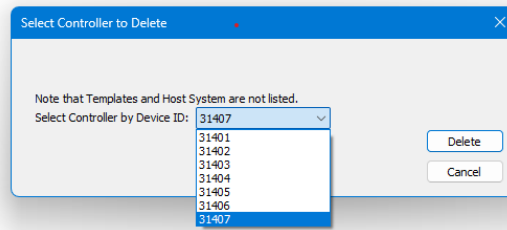


Fig. 8. The Delete Controller Window

4.8. Adding Objects to Controllers

BACnet objects can be added to an existing controller. Select the “Add Object” button to open the “Add New Object” window, shown in Fig. 9. Select the controller to add the new object to. The default selection is the highest numbered device ID. Select the object type from the drop-down menu. Enter the object instance and the number of objects to add. The object instance number will increment by one for each object that is added. The status window at the bottom of the dialog will update to reflect the most recent selections, as shown in Fig. 10.

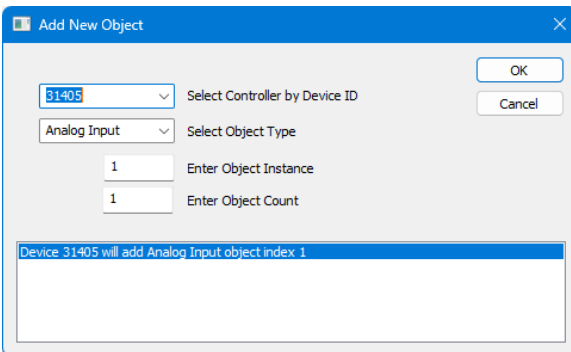


Fig. 9. The Add New Object Window

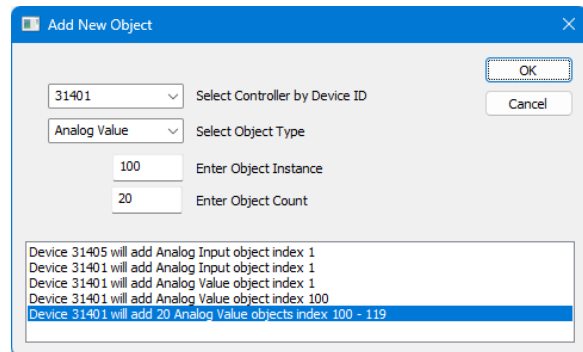


Fig. 10. Changes to New Object

When the configuration is complete press the OK button. The new objects will be added to the controller as indicated.

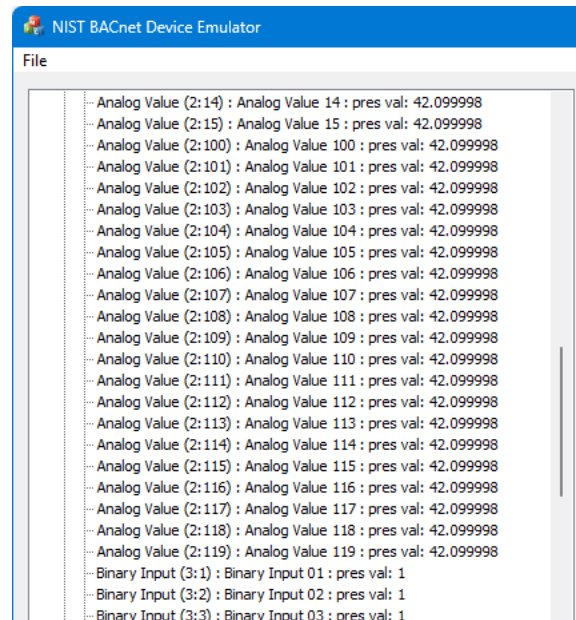


Fig. 11. Objects Added to Controller

4.9. Viewing Controller Data

The objects in the BDE are created with a default subset of properties, each of which is assigned a default value. The values of the properties can be viewed in the Focus Tree Window by selecting a controller in the Net Tree Window as shown in Fig. 12. The objects in the selected controller will be displayed in the Focus Window. Expanding an object will display the properties and values of the object.

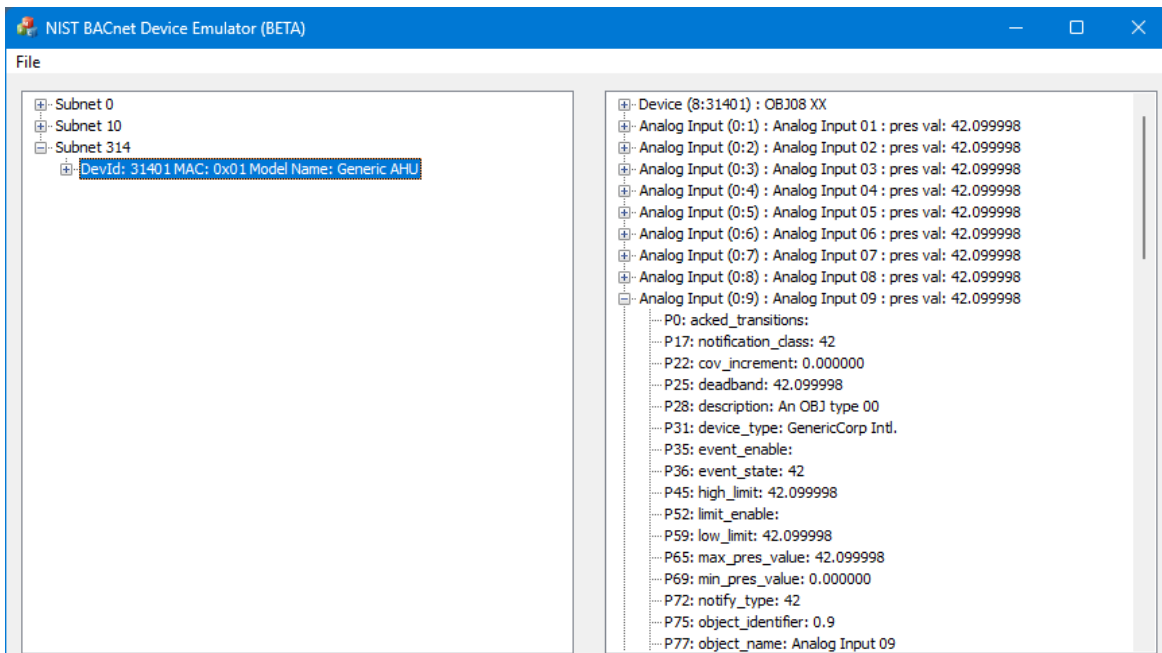


Fig. 12. Object and properties shown in Focus Window

4.10. Modifying Property Values

Object pre-set default property values need to be updated to match the desired configuration. The value of any property can be modified by double-clicking on the property in the Focus Tree window. This will open the Modify Property Value window. To verify the correct property data is being modified, this window displays the Device ID, BACnet object type and instance, and type of property being modified as shown in Fig. 13. It also displays the current value of the property. Below the verification information is an edit box where the new value for the property can be entered. The updated value is shown in Fig. 14.

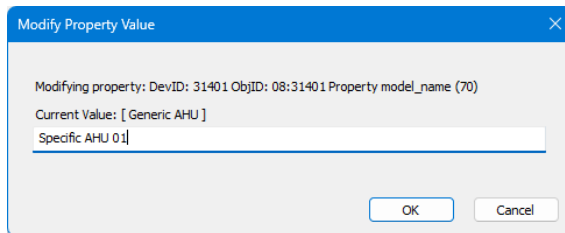


Fig. 13. The Modify Property Value Window

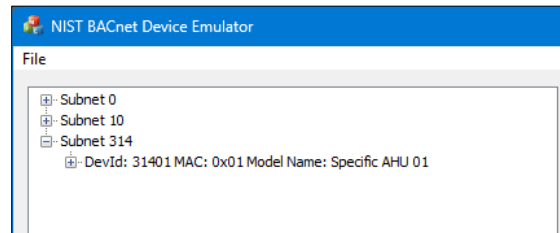


Fig. 14. The Modified Model Name

4.11. Saving and Loading Configuration Files

The configuration of the virtual controllers in the BDE can be saved to a file by selecting the “Save File” button. This will open a dialog allowing selection of the name and location of the file. Configuration files are saved in XML format and may be modified in a text editor. Files may be loaded into the BDE by selecting the “Read File” button or by selecting the file in Windows Explorer and dropping the file on the BDE interface. If the configuration file has been modified check the Net Tree Status Window for any report of an error. Note that the Host System from a saved configuration file and template controllers that are already loaded will not be added to the emulated network.

4.12. Finishing Steps Before Use

When all the controllers are entered and configured, there are a few steps to take before using them. If the computer the BDE is running on has more than one network adapter, the correct one must be chosen in the “Select Network Adapter by IP” drop-down menu at the bottom center of the BDE interface. This would typically be found on a computer with a virtual machine or Windows emulator installed. If there is only one network adapter, the drop-down menu will not be displayed.

Next, optionally select the “Send I-Am” button to broadcast the presence of the emulated network and controllers. This information should be visible to real controllers and to BACnet-enabled software (such as the FPTM or BDS) on the local network and allow them to interact

with the emulated controllers as if they were also real controllers. This is demonstrated in

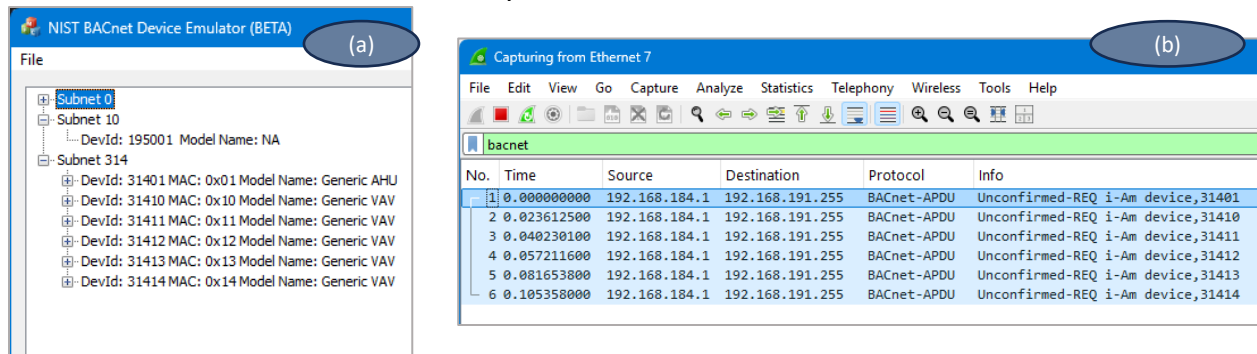


Fig. 15 which shows (a) the list of emulated controllers in the BDE, (b) The I-Am messages received by Wireshark (an open-source network packet analyzer) after the “Send I-Am” button was pressed. Note that the destination subnet is configured with a /20 subnet mask, which the BDE detected and accounted for when sending messages.

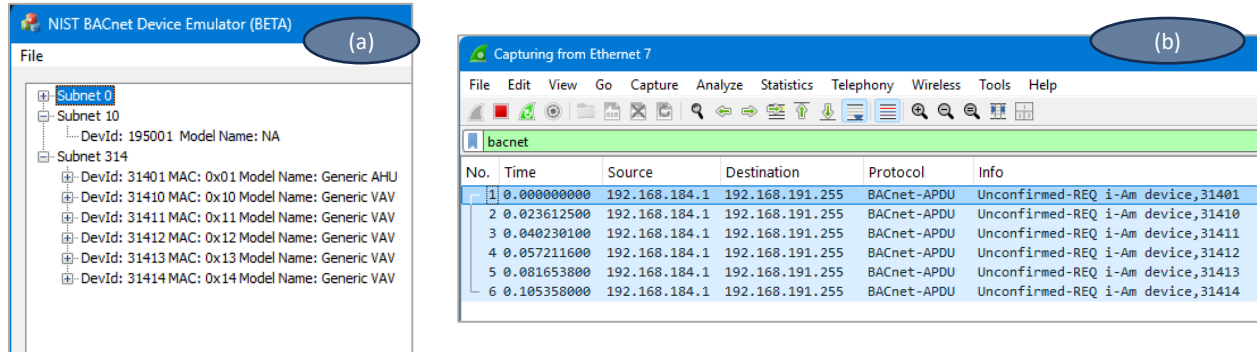


Fig. 15. Sending a BACnet I-Am message. (a) the list of emulated controllers in the BDE, (b) The I-Am messages received by Wireshark

5. Example Use Cases

This section provides examples of BDE usage with LabVIEW and Python.

5.1. IBAL Use Case with LabVIEW

Fig. 16 is a schematic of the DT infrastructure under development in the IBAL. In the current configuration, the IBAL Equipment receive control signals (e.g., V or A) from the lab data acquisition program (IBAL DAQ), which also contains the control logic for operating the equipment. The BDE will provide an interface between the IBAL DAQ, the DT, and a future controller microservice. Control logic will be removed from the IBAL DAQ to enable testing of more advanced control approaches and align the IBAL configuration more closely with a real building. This example will focus on how the BACnet network was created in the BDE and how the IBAL DAQ reads and writes data to the BDE using API calls [8] to BACdll2012.dll (BCD) from LabVIEW.

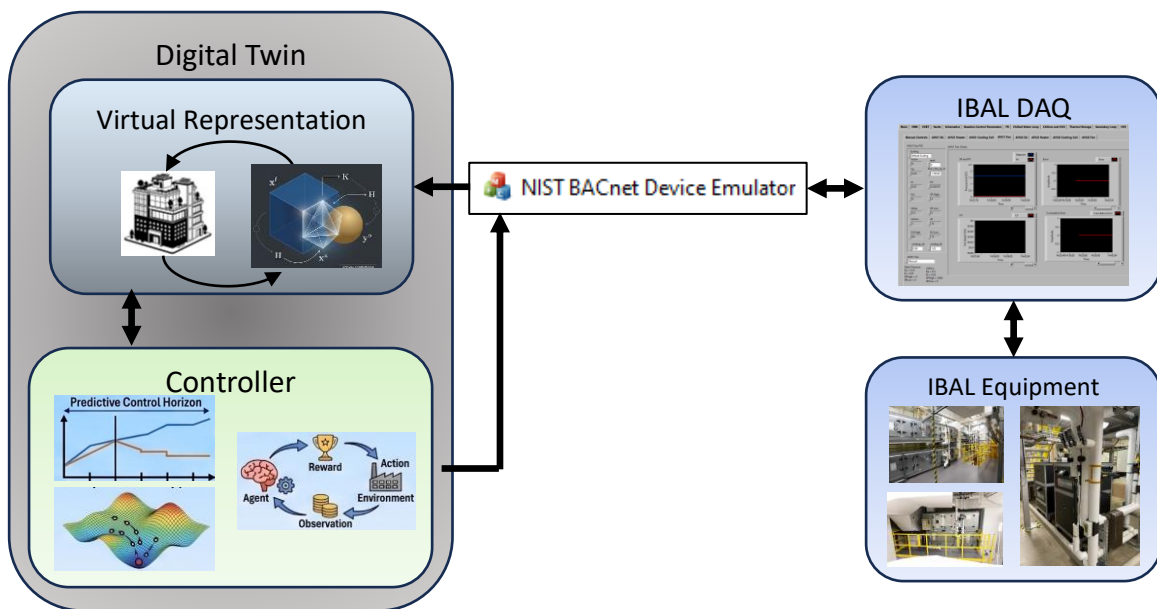


Fig. 16. Data flow for the digital twin to physical twin interaction in the IBAL.

The first step was to create the emulated network. In 2025, the IBAL was connected to controllers in the Virtual Cybernetic Building Testbed (VCBT) to compare the performance of commercial implementations of ASHRAE Guideline 36 sequences of operation and a legacy control approach in the air system [8]. For this example, the points list from that test was converted from a CSV file (1) to an XML file (2) and loaded into the BDE (3) (see Fig. 17).

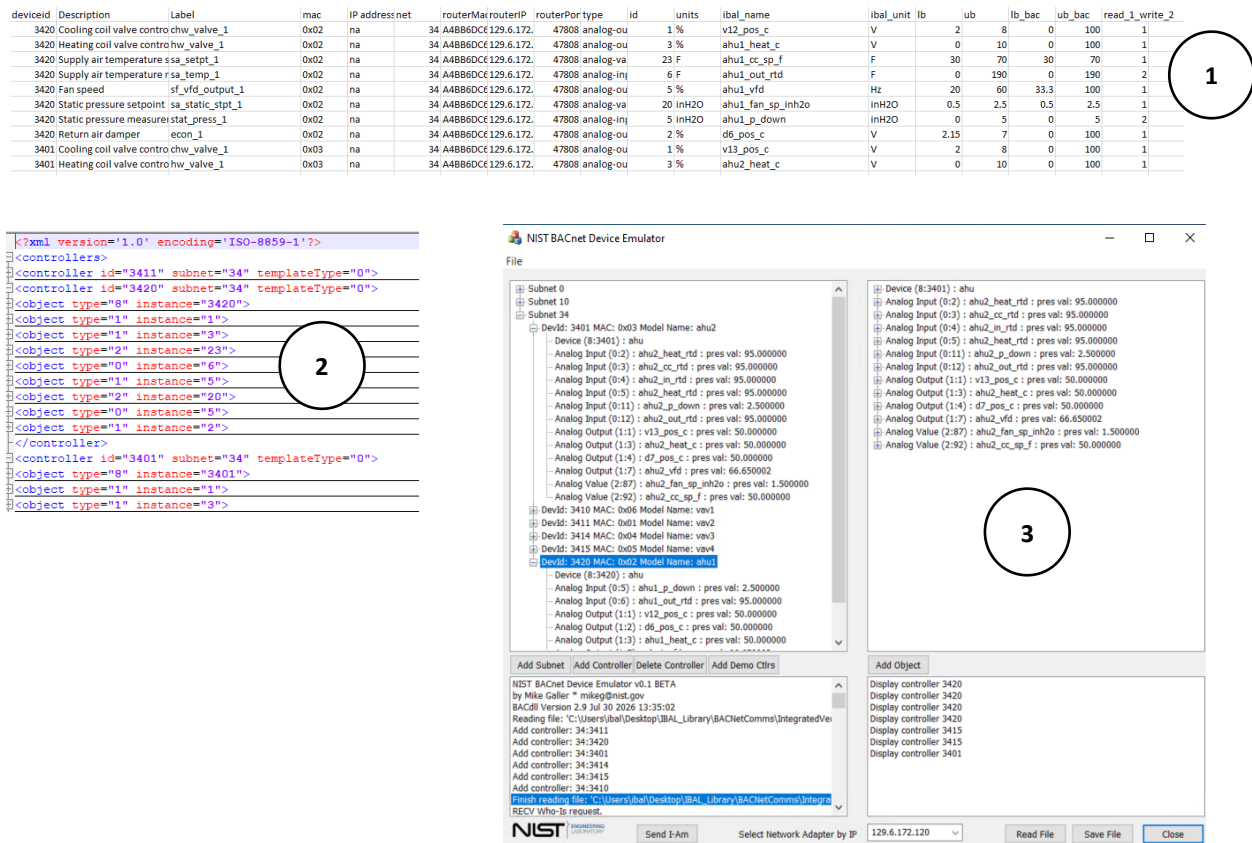


Fig. 17. Steps to create the emulated BACnet network for the IBAL use case. 1) csv file; 2) xml file; 3) the emulated network in the BDE.

This network was then accessed from LabVIEW using API calls to BCD using the “Call Library Function Node”; see Appendix A.1 for screenshots of the LabVIEW code. Table 1 lists the functions that are called from LabVIEW and Table 2 describes the key parameters that are passed to the functions.

Table 1. List of LabVIEW function calls.

Function prototype	Purpose
int32_t AdapterInit(void);	Initialize the connection to the network
int32_t SetDefaultAdapterSLI(CStr id);	Set the network adapter number
int32_t ReadPVPropertyByDeviceSLI(const CStr devid, const CStr obj, const CStr inst, CStr bv, CStr fault);	Read from the device
CStr WritePVPropertyByDeviceSLI(CStr devid, CStr obj, CStr inst, CStr bv, CStr fault);	Write to the device

Table 2. Description of the key parameters for the read and write functions.

Parameter	Description	Read Value	Write Value
devid	Device ID – a unique number that the device is identified by on the network. Typically begins with the subnet number (e.g., 3420 is on subnet 34)	3420	3420
obj	BACnet type of the data object. For example, analog input (0), analog output (1), analog value (2).	2	0
inst	Instance of the object. For example, the 20 th analog value	20	6
bv	This is the return value, in this case the present value	Empty string	75
fault	Returns a fault flag. 0 or no flag indicates no fault. It is initialized to an empty string.	Empty string	Empty string

5.2. Use Case with Python

In this example, the same emulated network from Section 5.1 was accessed from Python. The basic procedure is the same as that used in LabVIEW: load the BCD, initialize the adapter, set the adapter number, and then read and write to the channels. The key functions are shown here, and the entire code is provided in Appendix A.2. The bitness (64 or 32 bit) and the bit order of the BCD and Python must match.

```

Load the DLL
bacnetDLL = ctypes.CDLL('BACdll2012x64.dll')

Initialize the adapter
adapter_count = bacnetDLL.WPCInit()

Set the adapter
bacnetDLL.SetDefaultAdapter(CStr id)

Read present value
read_obj_pv = bacnetDLL.ReadPVPropertyByDeviceSLI(
    const CStr devid, const CStr obj,
    const CStr inst, CStr bv, CStr fault)

Write present value
write_obj_pv = bacnetDLL.WritePVPropertyByDeviceSLI(
    const CStr devid, const CStr obj,
    const CStr inst, CStr bv, CStr fault)

```

5.3. Use Case with Knowledge Graphs

In traditional building management systems, telemetry data (such as sensor readings) and contextual metadata (such as the location of a sensor or the equipment it is associated with) are often tightly coupled in a vendor-specific naming convention (e.g., VAV_03_DAT_AO_2). This hinders scaling analytics, building fault detection algorithms, or integrating building applications into the buildings. This example will focus on how the BDE can be used with a S223 knowledge graph to provide data from an emulated BACnet network in response to a user's query.

The S223 graph models physical topology, spatial relationships, equipment hierarchies, and rich semantic context. For example, the graph could include "Sensor A" as a node of type sensor that observes the property "Discharge Air Temperature" with units of "°F", and "VAV 3" as a node of type equipment. A graph edge connects Sensor A to VAV 3, so the graph can be used to infer that "Sensor A measures the Discharge Air Temperature of VAV 3 in °F". By serving purely as metadata rather than as a telemetry database, the S223 graph, which uses the Resource Description Framework (RDF), remains lightweight, fast, and maintainable.

The user's question is: What is the present value of the discharge air temperature from VAV 3? This question might be asked by an automated fault detection and diagnostics microservice that is attempting to diagnose the cause of a fault, or it may be a query from maintenance staff who are investigating a complaint from an occupant of the zone served by VAV 3. This is converted from plain language to a SPARQL query using S223 and BACnet ontologies to map a specific BACnet point (discharge air temperature) to a physical location (downstream) or monitored equipment (VAV 3). This is a critical architectural step in many building applications because the S223 graph serves as the digital twin map and does not need to store live or trend telemetry. Instead, S223 defines semantic properties, such as a VAV discharge air temperature, and links them to BACnet network addresses in a standardized manner.

1. Query the Graph: Search for the specific equipment (IBAL:VAV 3) and traverse its properties (Discharge Temperature) to extract the external BACnet reference (device-identifier and object-identifier).

```
query_str = """
SELECT ?property ?extRef ?dev_id ?obj_id
WHERE {
    # 1. Target IBAL:vav3 and its specific property context
    ?eq s223:hasProperty ?property .

    # 2. Filter property by aspect (Role-Discharge), quantity kind (Temperature), and unit (DEG_F)
    ?property s223:hasAspect s223:Role-Discharge ;
    qudt:hasQuantityKind quantitykind:Temperature ;
    qudt:hasUnit unit:DEG_F ;
    s223:hasExternalReference ?extRef .

    # 3. Extract BACnet external reference details
    ?extRef bacnet:device-identifier ?dev_id ;
    bacnet:object-identifier ?obj_id .
}
"""
```

2. Fetch Live Data: Pass the extracted BACnet device, object, and property IDs to the BDS driver function to retrieve the real-time present-value from the BDE device.

```
# call BDS with data obtained from graph  
retval = bacnetDLL.ReadPVPropertyByDeviceSLI(deviceId, obj_buffer, inst_buffer, bv, fault)
```

This query returns the present value of 72 °F from Analog Output 2 on the emulated device 3101. By decoupling the topological model (S223) from the dynamic data layer (BDS), the system remains lightweight, maintainable, and highly scalable. The entire code is shown in Appendix A.3.

6. Conclusions and Future Work

The BDE provides a powerful infrastructure for situations where a BACnet network does not yet exist. The entire network can be generated within the BDE, allowing different BACnet control strategies to be developed and tested in a simulated environment before deployment. This is a crucial step toward building digitization and the modernization of building systems.

The BDE is available on GitHub for download, and it will continue to be updated to add features as needed by users. We currently plan to add additional template options, improve configuration options, provide a range of SOO logic libraries, and include an option to export to a BACnet Turtle (ttl) file, a serialized representation of a graph, enabling a semantic graph to be generated semi-automatically from an emulated network for direct use with ASHRAE 223. We will also include other features requested by users of the BDE and welcome all feedback.

Appendix A.

A.1. IBAL Example with LabVIEW

The following figures show screenshots of individual “Call Library Function” configuration interfaces for wrapping BCD functions, enabling VI based read and write access to the BDE. Each figure shows details of the API function call and the LabVIEW settings for each parameter in that function call.

A.1.1. AdapterInit

Fig. 18 shows both the details of the function call and the parameters, in this case just a return of either -1, which indicates an error, or ≥ 0 , which indicates the number of network adapters initialized.

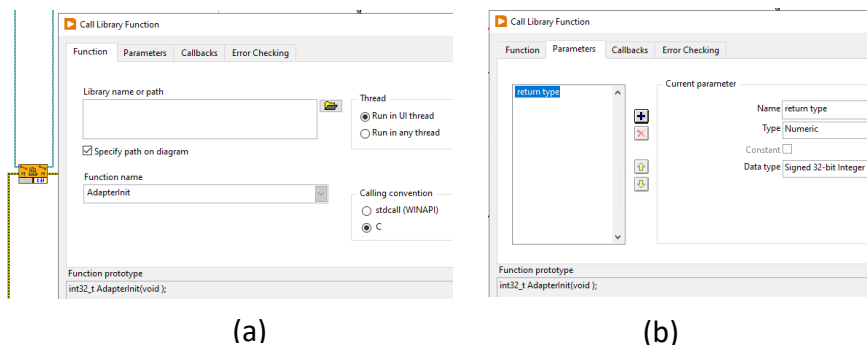


Fig. 18. LabVIEW screenshots of the AdapterInit function call. a) Definition of the function call. b) Definition of the return type parameter.

A.1.2. SetDefaultAdapterSLI

The computer’s network adapter that sees the BACnet objects is set to an integer value based on the computer. Fig. 19 shows the function call and parameters.

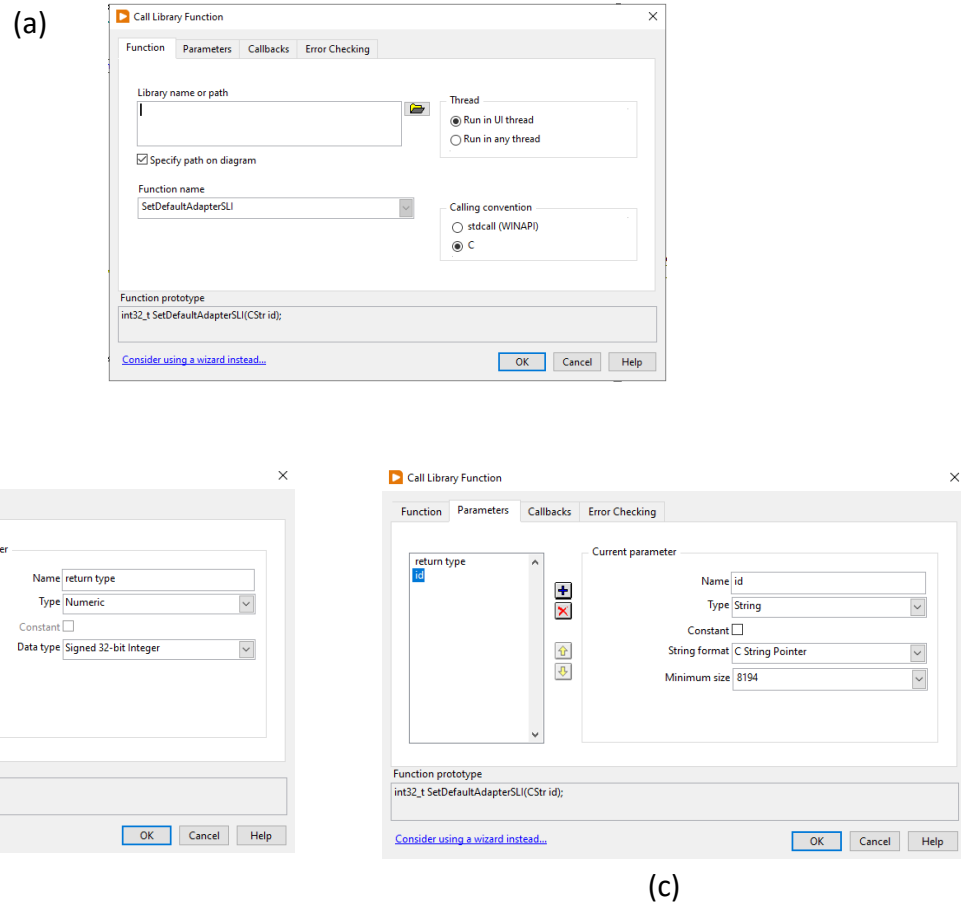


Fig. 19. LabVIEW screenshots of the SetDefaultAdapterSLI function call. a) Definition of the function call. b) Definition of the return type parameter. c) Definition of the id parameter.

A.1.3. ReadPVPropertyByDeviceSLI

The function call to read a present value to the emulated BACnet device and its parameters are shown in Fig. 20.

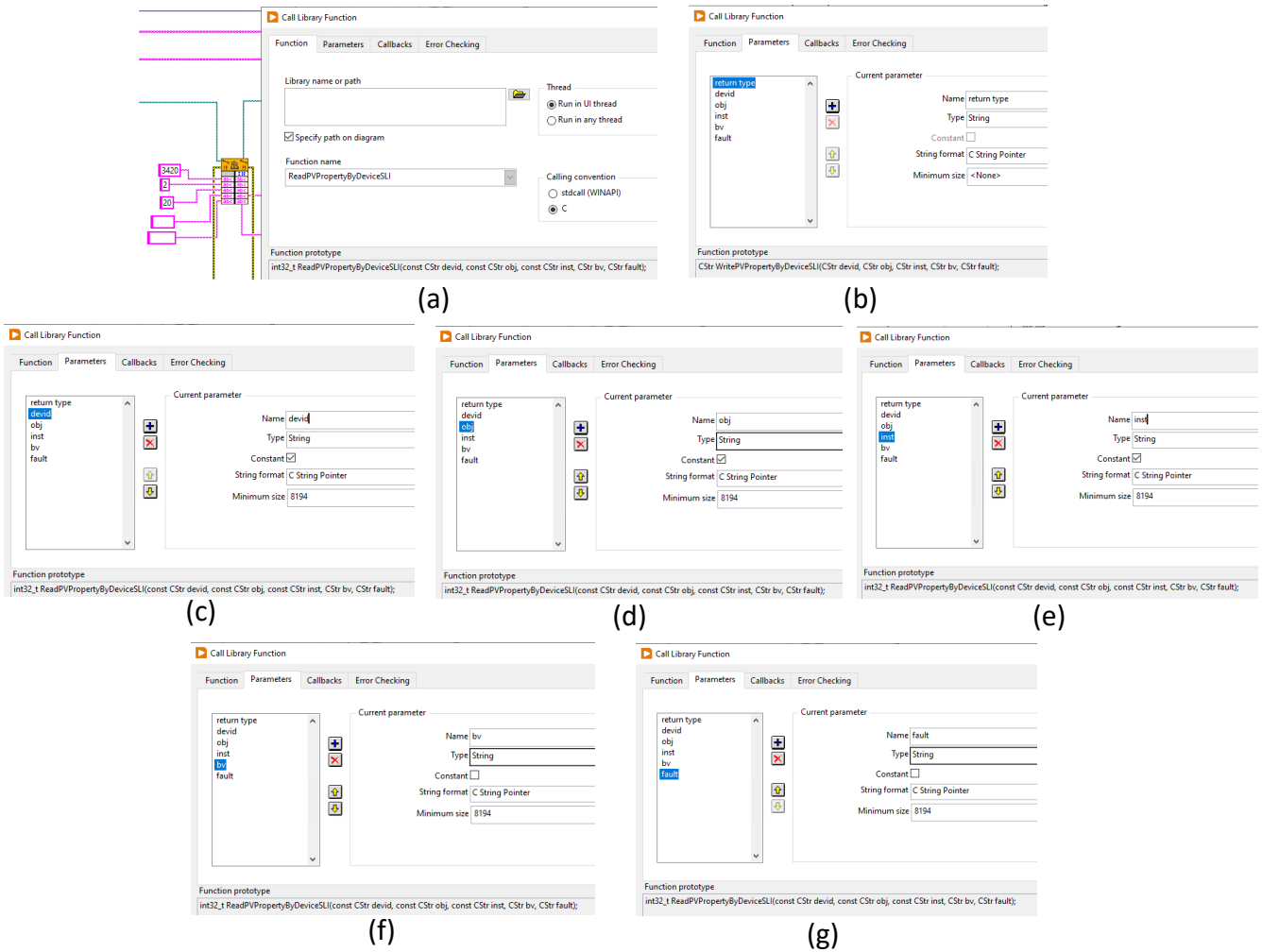


Fig. 20. LabVIEW screenshots of the ReadPVPropertyByDeviceSLI function call. a) Definition of the function call. b) Definition of the return_type parameter. c) Definition of the id parameter. d) Definition of the obj parameter. e) Definition of the inst parameter. f) Definition of the bv (present value) parameter. g) Definition of the fault parameter.

A.1.4. WritePVPropertyByDeviceSLI

The function call to write a present value to the emulated BACnet device and its parameters are shown in Fig. 21.

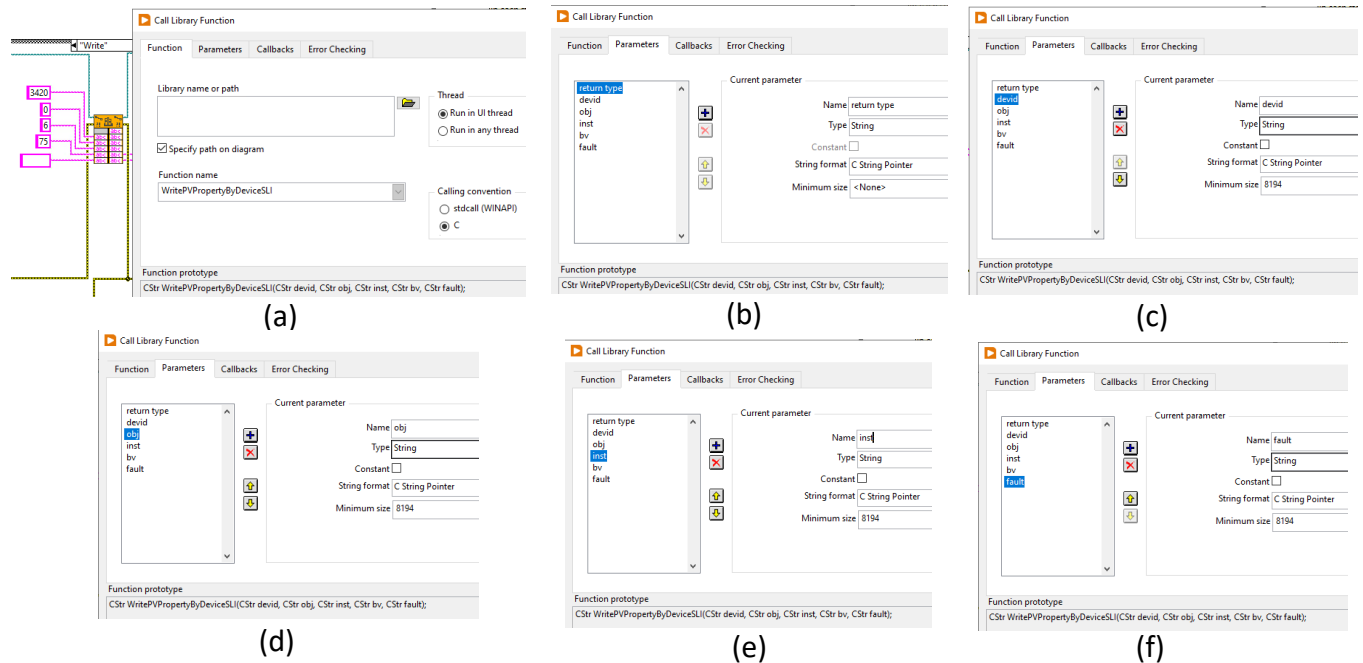


Fig. 21. LabVIEW screenshots of the WritePVPropertyByDeviceSLI function call. a) Definition of the function call. b) Definition of the return type parameter. c) Definition of the id parameter. d) Definition of the obj parameter. e) Definition of the inst parameter.

A.2. Use Case with Python

This section contains the Python script used to read and write to the emulated BACnet network. Much of the code in this section was written with the assistance of Gemini 3.6 and Claude Opus 4.7.

A.2.1. Load the DLL

```
import ctypes

try:
    bacnetDLL = ctypes.CDLL('BACdll2012x64.dll')
except OSError as e:
    print(f'Failed to load DLL (check bitness / path): {e}')
    raise
```

A.2.2. Initialize and Set the Adapter

```
# int WPCInit(void) -> <0 error, >=0 adapter count
bacnetDLL.WPCInit.argtypes = []
bacnetDLL.WPCInit.restype = ctypes.c_int

# int SetDefaultAdapter(int index) -> 0 error, 1 ok
bacnetDLL.SetDefaultAdapter.argtypes = [ctypes.c_int]
bacnetDLL.SetDefaultAdapter.restype = ctypes.c_int

# int GetAdapterIPSLI(int index, char* ipOut)
bacnetDLL.GetAdapterIPSLI.argtypes = [ctypes.c_int, ctypes.c_char_p]
bacnetDLL.GetAdapterIPSLI.restype = ctypes.c_int

# --- Initialization ---
TARGET_IP = 'XXX.X.XXX.XXX'
MAX_ADAPTERS = 16

def initialize_bacnet(target_ip: str = TARGET_IP) -> int:
    """Initialize WinPcap and bind the DLL to the NIC with the given IP.

    Returns the adapter index that was selected.
    Raises RuntimeError if init fails or the target IP isn't found.
    """
    adapter_count = bacnetDLL.WPCInit()
    if adapter_count < 0:
        raise RuntimeError(f'WPCInit failed with code {adapter_count} '
                           '(WinPcap/Npcap installed? Running as admin?)')
    print(f'WPCInit OK. {adapter_count} adapter(s) found.')

    ip_buf = ctypes.create_string_buffer(64)
    selected = None

    # Loop through potential adapters until you find the correct one
    for it in range(min(adapter_count, MAX_ADAPTERS)):
        ctypes.memset(ip_buf, 0, ctypes.sizeof(ip_buf))
        bacnetDLL.GetAdapterIPSLI(it, ip_buf)
        ip_str = ip_buf.value.decode('utf-8', errors='replace').strip()
        print(f' adapter {it}: {ip_str!r}')
        if ip_str == target_ip:
            selected = it
            break

    if selected is None:
        raise RuntimeError(f'Adapter with IP {target_ip} not found. '
                           'Check the machine is on the BACnet subnet.')
```

```
if bacnetDLL.SetDefaultAdapter(selected) == 0:
    raise RuntimeError(f'SetDefaultAdapter({selected}) failed.')

print(f'Bound to adapter {selected} ({target_ip}).')
return selected
```

A.2.3. Create the Read and Write Objects

```
# Set up the read and write objects
# Use the more general read function; this function requires the user to
# specify which property they want to read. Present value is the number 85.
read_obj_prop = bacnetDLL.ReadObjPropertyByDeviceSLI
read_obj_prop.argtypes = [
    ctypes.c_char_p, ctypes.c_char_p, ctypes.c_char_p,
    ctypes.c_char_p, ctypes.c_char_p, ctypes.c_char_p,
]
read_obj_prop.restype = ctypes.c_int

# Use the more general write function; this function requires the user to
# specify which property they want to write to. Present value is the number 85.
write_obj_prop = bacnetDLL.WriteObjPropertyByDeviceSLI
write_obj_prop.argtypes = [
    ctypes.c_char_p, ctypes.c_char_p, ctypes.c_char_p,
    ctypes.c_char_p, ctypes.c_char_p, ctypes.c_char_p,
]
write_obj_prop.restype = ctypes.c_int

# This is a specific read function that will only read the present value
read_obj_pv = bacnetDLL.ReadPVPropertyByDeviceSLI
read_obj_pv.argtypes = [
    ctypes.c_char_p, ctypes.c_char_p, ctypes.c_char_p,
    ctypes.c_char_p, ctypes.c_char_p,
]
read_obj_pv.restype = ctypes.c_int

# This is a specific write function that will only write to the present value
write_obj_pv = bacnetDLL.WritePVPropertyByDeviceSLI
write_obj_pv.argtypes = [
    ctypes.c_char_p, ctypes.c_char_p, ctypes.c_char_p,
    ctypes.c_char_p, ctypes.c_char_p,
]
write_obj_pv.restype = ctypes.c_int
```

A.2.4. Functions for Read and Write

```
# These functions are used to format the values correctly and call the functions
def read_device_property(devid, obj, inst, prop, buffer_size=256):
    """Read the specified property; all inputs and outputs are strings:
        devid = device identification
        obj = object; can be a string integer (e.g., 0,1,2) or the term
            (e.g., analog-input,analog-output,analog-value)
        inst = instance of the property (e.g., 0,1,2); an example is analog-input 1
        prop = property to read; 85 is present value
        buffer_size = specifies the size of the buffer for the return values
    outputs:
        bv = this is the property value
        fault = if there is an error during the read, it will be specified here."""
    bv = ctypes.create_string_buffer(buffer_size)
    fault = ctypes.create_string_buffer(buffer_size)
    result = read_obj_prop(devid.encode('utf-8'), obj.encode('utf-8'),
                          inst.encode('utf-8'), prop.encode('utf-8'),bv, fault)
    return result, bv.value.decode('utf-8'), fault.value.decode('utf-8')

def write_device_property(devid, obj, inst, bv, prop, buffer_size=256):
    """Write the specified property; all inputs and outputs are strings:
        devid = device identification
        obj = object; can be a string integer (e.g., 0,1,2) or the term
            (e.g., analog-input,analog-output,analog-value)
        inst = instance of the property (e.g., 0,1,2); an example is analog-input 1
        bv = this is the value of the property to write
        prop = property to read; 85 is present value
        buffer_size = specifies the size of the buffer for the return values
    outputs:
        fault = if there is an error during the read, it will be specified here."""
    fault = ctypes.create_string_buffer(buffer_size)
    result = write_obj_prop(devid.encode('utf-8'), obj.encode('utf-8'),
                          inst.encode('utf-8'), prop.encode('utf-8'),bv.encode('utf-8'), fault)
    return result, fault.value.decode('utf-8')

def read_device_pv(devid, obj, inst, buffer_size=256):
    """Read the present value; all inputs and outputs are strings:
        devid = device identification
        obj = object; can be a string integer (e.g., 0,1,2) or the term
            (e.g., analog-input,analog-output,analog-value)
        inst = instance of the property (e.g., 0,1,2); an example is analog-input 1
        buffer_size = specifies the size of the buffer for the return values
    outputs:
        bv = this is the present value
        fault = if there is an error during the read, it will be specified here."""
```

```
    bv = ctypes.create_string_buffer(buffer_size)
    fault = ctypes.create_string_buffer(buffer_size)
    result = read_obj_pv(devid.encode('utf-8'), obj.encode('utf-8'), inst.encode('utf-8'),
                        bv, fault)
    return result, bv.value.decode('utf-8'), fault.value.decode('utf-8')

def write_device_pv(devid, obj, inst, bv, buffer_size=256):
    """Write the present value; all inputs and outputs are strings:
        devid = device identification
        obj = object; can be a string integer (e.g., 0,1,2) or the term
            (e.g., analog-input, analog-output, analog-value)
        inst = instance of the property (e.g., 0,1,2); an example is analog-input 1
        bv = this is the present value to write
        buffer_size = specifies the size of the buffer for the return values
    outputs:
        fault = if there is an error during the read, it will be specified here."""
    fault = ctypes.create_string_buffer(buffer_size)
    result = write_obj_pv(devid.encode('utf-8'), obj.encode('utf-8'), inst.encode('utf-8'),
                        bv.encode('utf-8'), fault)
    return result, fault.value.decode('utf-8')
```

A.2.5. Execute the Code

```
if __name__ == '__main__':
    initialize_bacnet()

    status, value, fault = read_device_property(devid='3420', obj='2', inst='20',
prop='85')
    writeVal = '85'
    status_w, fault_w = write_device_property(devid='3420', obj='0', inst='6',
                        bv = writeVal, prop='85')

    status_r, value_r, fault_r = read_device_property(devid='3420', obj='analog-input',
                        inst='6', prop='85')

    print(f'Read return code: {status}; Read value: {value}; Read fault: {fault}')
    print(f'Write return code: {status_w}; Write value: {writeVal}; Write fault: {fault_w}')
    print(f'Read return code: {status_r}; Read value: {value_r}; Read fault: {fault_r}')

    status, value, fault = read_device_pv(devid='3420', obj='2', inst='20')
    writeVal = '75'
    status_w, fault_w = write_device_pv(devid='3420', obj='0', inst='6', bv = writeVal)

    status_r, value_r, fault_r = read_device_pv(devid='3420', obj='analog-input', inst='6')
    print(f'Read return code: {status}; Read value: {value}; Read fault: {fault}')
```

```
print(f'Write return code: {status_w}; Write value: {writeVal}; Write fault: {fault_w}')  
print(f'Read return code: {status_r}; Read value: {value_r}; Read fault: {fault_r}')
```

A.3. Use Case with Knowledge Graphs

This section contains the Python script used to query the IBAL graph and return the present value in response to the query: What is the present value of the discharge air temperature from VAV 3?

```
# the goal is to construct a function that takes the equipment, finds the right property,  
and returns the BACnet information (device and Object ids).  
from rdflib import Graph, Namespace, URIRef, Literal  
from rdflib.plugins.sparql import prepareQuery  
  
# Load graph  
g = Graph()  
g.parse("IBAL.ttl", format="turtle") # or .rdf, .xml, etc.  
S223 = Namespace("http://data.ashrae.org/standard223#") # Replace with real URIs  
IBAL = Namespace("http://example.org/IBAL#")  
BACNET = Namespace("http://data.ashrae.org/bacnet/2020#")  
QUDT = Namespace("http://qudt.org/schema/qudt/")  
QUANTITYKIND = Namespace("http://qudt.org/vocab/quantitykind/")  
UNIT = Namespace("http://qudt.org/vocab/unit/")  
  
# Define Query to get the bacnet info for  
query_str = """  
SELECT ?property ?extRef ?dev_id ?obj_id  
WHERE {  
    # 1. Target IBAL:vav3 and its specific property context  
    ?eq s223:hasProperty ?property .  
    # 2. Filter property by aspect (Role-Discharge), quantity kind (Temperature), and unit  
    (DEG_F)  
    ?property s223:hasAspect s223:Role-Discharge ;  
    qudt:hasQuantityKind quantitykind:Temperature ;  
    qudt:hasUnit unit:DEG_F ;  
    s223:hasExternalReference ?extRef .  
    # 3. Extract BACnet external reference details  
    ?extRef bacnet:device-identifier ?dev_id ;  
    bacnet:object-identifier ?obj_id .  
}  
"""  
  
# testing the results obtained from the graph  
q = prepareQuery(query_str, initNs={"s223": S223, "bacnet":  
BACNET, "IBAL": IBAL, "qudt": QUDT, "quantitykind": QUANTITYKIND, "unit": UNIT})  
# setting the equipment ID
```

```
eq_id = IBAL.VAV_3
print(eq_id)

# Obtaining the results
results = g.query(q, initBindings={
    'eq': eq_id
})
rows = list(results)
if rows:
    row = rows[0]
    print(str(row.property),str(row.extRef),str(row.dev_id),str(row.obj_id))

#cleaning up the format to call the function
raw_valueO = str(row.obj_id)
raw_valueD = str(row.dev_id)
obj_str, inst_str = raw_valueO.split(',')
junk_str, deviceId_str = raw_valueD.split(',')

obj_buffer = ctypes.create_string_buffer(obj_str.encode('utf-8'))
inst_buffer = ctypes.create_string_buffer(inst_str.encode('utf-8'))
deviceId = ctypes.create_string_buffer(deviceId_str.encode('utf-8'))
# initialize the return value to an empty string
bv = ctypes.create_string_buffer(b'      ')
# initialize the fault to an empty string
fault = ctypes.create_string_buffer(b'      ')

# call BDS with data obtained from graph
retval = bacnetDLL.ReadPVPropertyByDeviceSLI(deviceId, obj_buffer, inst_buffer, bv,
fault)

print("after the call")
fault_bytes = fault.value.strip()
if not fault_bytes:
    fault = 0
else:
    fault = int(fault_bytes)

print(eval(bv.value), ((fault)),(retval))

# Check retval value for error. If no error indicated use get the present value from bv.
if (retval != 0)&(fault==0):
    myValue = eval(bv.value)
    print('Present value = '+str(eval(bv.value))+ ' fault = '+str(fault))
else:
    print('There was an error! bv = '+str(bv.value)+ ' fault = '+str(fault))
```

References

- [1] Singh, M., Srivastava, R., Fuenmayor, E., Kuts, V., Qiao, Y., Murray, N., & Devine, D. (2022). Applications of Digital Twin across Industries: A Review. *Applied Sciences*, 12(11), 5727. <https://doi.org/10.3390/app12115727>
- [2] V. Hassija et al., "A Survey on Digital Twins: Enabling Technologies, Use Cases, Application, Open Issues, and More," in *IEEE Journal of Selected Areas in Sensors*, vol. 2, pp. 84-107, 2025, DOI: 10.1109/JSAS.2024.3523856.
- [3] National Academies of Sciences, Engineering, and Medicine. 2024. *Foundational Research Gaps and Future Directions for Digital Twins*. Washington, DC: The National Academies Press.
- [4] Open223. (2026, August 3). *ASHRAE Standard 223P User Documentation*. Retrieved September 1, 2026, from <https://docs.open223.info/>.
- [5] Galler, M. (2018), Programmers Guide to the BACnet Communications DLL, Technical Note (NIST TN), National Institute of Standards and Technology, Gaithersburg, MD, [online], <https://doi.org/10.6028/NIST.TN.2023> (Accessed July 31, 2026).
- [6] Galler, M. (2011), Using the BACnet Data Source v1.7, NIST Interagency/Internal Report (NISTIR), National Institute of Standards and Technology, Gaithersburg, MD, [online], <https://doi.org/10.6028/NIST.IR.7825> (Accessed July 31, 2026).
- [7] Galler, M. A. (2026), BACnet Device Emulator Developers' Guide, Technical Note (NIST TN), National Institute of Standards and Technology, Gaithersburg, MD, [online] (in development)
- [8] Galler, M.A. (2024), Using the Scripting Language Interface to the BACnet Communications Dynamic Link Library. Technical Note 2309, National Institute of Standards and Technology, Gaithersburg, MD, <https://doi.org/10.6028/NIST.TN.2309>
- [9] Pertzborn, A. J., & Galler, M. A. (2026). Experimental comparison of guideline 36 and legacy controller sequences of operation using commercial-off-the-shelf control software. *Science and Technology for the Built Environment*, 1–13. <https://doi.org/10.1080/23744731.2026.2663718>.