pycomm3

Release 1.2.7

Ian Ottoway

May 16, 2022

CONTENTS

CHAPTER

ONE

INTRODUCTION

pycomm3 started as a Python 3 fork of pycomm, which is a Python 2 library for communicating with Allen-Bradley PLCs using Ethernet/IP. The initial Python 3 port was done in this fork and was used as the base for pycomm3. Since then, the library has been almost entirely rewritten and the API is no longer compatible with pycomm. Without the hard work done by the original pycomm developers, pycomm3 would not exist. This library seeks to expand upon their great work.

CHAPTER

TWO

DRIVERS

pycomm3 includes 3 drivers:

- **CIPDriver** This driver is the base driver for the library, it handles common CIP services used by the other drivers. Things like opening/closing a connection, register/unregister sessions, forward open/close services, device discovery, and generic messaging. It can be used to connect to any Ethernet/IP device, like: drives, switches, meters, and other non-PLC devices.
- **LogixDriver** This driver supports services specific to ControlLogix, CompactLogix, and Micro800 PLCs. Services like reading/writing tags, uploading the tag list, and getting/setting the PLC time.
- **SLCDriver** This driver supports basic reading/writing data files in a SLC500 or MicroLogix PLCs. It is a port of the `SlcDriver` from `pycomm` with minimal changes to make the API similar to the other drivers. Currently this driver is considered legacy and it's development will be on a limited basis.

CHAPTER

THREE

DISCLAIMER

PLCs can be used to control heavy or dangerous equipment, this library is provided “as is” and makes no guarantees on its reliability in a production environment. This library makes no promises in the completeness or correctness of the protocol implementations and should not be solely relied upon for critical systems. The development for this library is aimed at providing quick and convenient access for reading/writing data inside Allen-Bradley PLCs.

CHAPTER

FOUR

SETUP

The package can be installed from PyPI using `pip`: `pip install pycomm3` or `python -m pip install pycomm3`.

Optionally, you may configure logging using the Python standard logging library. A convenience method is provided to help configure basic logging, see the Logging Section in the docs for more information.

CHAPTER

FIVE

PYTHON AND OS SUPPORT

pycomm3 is a Python 3-only library and is supported on Python versions from 3.6.1 up to 3.10. There should be no OS-specific requirements and should be able to run on any OS that Python is supported on. Development and testing is done primarily on Windows 10. If you encounter an OS-related problem, please open an issue in the GitHub repository and it will be investigated.

Attention: Python 3.6.0 is not supported due to `NamedTuple` not supporting default values and methods until 3.6.1

5.1 Contents

5.1.1 Getting Started

Creating a Driver

Drivers are simple to create and use, the quickest way is to use them within a context manager (`with` statement). Most of the examples in the documentation will shown them used in that way. If you are using them as part of a larger program or creating long-lived connections, you may not want to use the context manager in this case. When used outside a context manager, you will need to call the *`open()`* method first and the *`close()`* method on shutdown. Failing to close the connection could cause issues communicating with the device. Each driver opens a single connection to the device, you may use multiple instances to create multiple connections. It is also the user's responsibility to maintain the connection, the drivers do not implement any periodic handshaking. The default timeout is fairly long, but a long lived connection will need to issue a request usually at least once a minute or the PLC may close the connection.

Each driver requires a `path` argument, this is a CIP path to the destination device. The paths used in `pycomm3` are similar to how they appear in Logix.

There are three possible forms:

IP Address Only (`10.20.30.100`) Use for devices without a backplane (drives, switches, Micro800 PLCs, etc) or for PLCs in slot 0 of a backplane. Only the LogixDriver and SLCDriver will automatically add the `backplane/0` to the path if no slot is specified.

IP Address/Slot (`10.20.30.100/1`) Use for PLCs in a backplane that are not in slot 0. Only supported in LogixDriver and SLCDriver.

CIP Routing Path (`1.2.3.4/backplane/2/enet/6.7.8.9/backplane/0`) This is a full CIP route to a device, it should appear similar to how paths are shown in Logix. For port selection, use `backplane` or `bp` for the backplane and `enet` for the ethernet port. Both slash (`/`) and backslash (`\`) are supported.

Note: Both the IP Address and IP Address/Slot options are shortcuts, they will be replaced with the CIP path automatically in the LogixDriver and SLCDriver.

```
>>> from pycomm3 import CIPDriver
>>> with CIPDriver('10.20.30.100') as drive:
>>>     print(drive)
Device: AC Drive, Revision: 1.2
```

The default behavior is to use the *Extended Forward Open* service when opening a connection. This allows the use of ~4KB of data for each request, the standard is only ~500 bytes. Although this requires the communications module to be an EN2T or newer and the PLC firmware to be version 20 or newer. Upon opening a connection, the `CIPDriver` will attempt an *Extended Forward Open*, if that fails it will then try using the standard *Forward Open*.

Creating a LogixDriver

The *LogixDriver* has two additional arguments:

init_tags (default True) When true, the driver will upload all tags in the PLC and the definitions for any UDTs and AOIs. These definitions are required for the *read()* and *write()* methods to work.

init_program_tags (default True) When uploading the tag list, if `True` all program scoped tags are uploaded. Set `False` to upload controller-scoped tags only. This arg is only checked if `init_tags` is `True`.

There is some data that is collected about the target controller when a connection is first established. It will call both the *get_plc_info()* and *get_plc_name()* methods. *get_plc_info()* returns a dict of the info collected and stores that information, making it accessible from the *info* property. *get_plc_name()* will return the name of the program running in the PLC and store it in `info['name']`. See *info* for details on the specific fields.

After the controller info has been retrieved, the driver will begin uploading the tag list unless `init_tags` option has not been set `False`. Depending on the number of tags, the PLC model, and other factors, the tag list could take some time to upload. A very large tag list on an old processor with high CPU utilization could take 10-15 seconds, while a small tag list or a new processor might take <1 second. If you are setting up multiple drivers on the same PLC, startup time can be saved by uploading the tag list in the first driver and disabling `init_tags` in the others. Then you can pass the uploaded tag list from the first driver to the other drivers, shown below.

```
from pycomm3 import LogixDriver
first_plc = LogixDriver('10.20.30.100')
first_plc.open()  # uploads the tag list
second_plc = LogixDriver('10.20.30.100', init_tags=False)
second_plc._tags = first_plc.tags
second_plc.open()  # doesn't upload any tags
```

Creating a SLCDriver

Currently, there is no additional configuration for a `SLCDriver` over a `CIPDriver`.

Response Tag Object

Many methods return a *Tag* object, like *generic_message()* or the `read` and `write` methods of the *LogixDriver* or *SLCDriver*. The truthiness of a `Tag` object represents the status of a request. A successful request will have a `value` that is not `None` and the `error` attribute is `None`. Anything otherwise will be a failed request. The `error` attribute will contain either the CIP error message or exception raised during the request.

class pycomm3.**Tag**(*tag*, *value*, *type*, *error*)

__bool__()

`True` if both `value` is not `None` and `error` is `None`, `False` otherwise

property tag

tag name of tag read/written or request name (generic message)

property value

value read/written, may be `None` on error

property type

data type of tag

property error

error message if unsuccessful, else `None`

Data Types

Data types are a major component of `pycomm3`, they are classes used to represent any tag or CIP object. They are able to encode and decode to and from Python values and bytes. Atomic and structure values along with arrays of either are supported. Each elementary (primitive) data type is provided as well as some common derived (structure of elementary types) types. See the *Data Types* for all available CIP types and *Custom Types* for any `pycomm3` provided custom types. The type classes provide two class methods: `encode` and `decode`. These are *class* methods, meaning they do not require an instance of they type to be created. In fact, the only time an instance of a type is used is when added members (with a name) to a structure. The `encode` method takes a Python object and encodes it to `bytes`. The `decode` method takes `bytes` and returns the corresponding Python object.

Elementary Types

Also known as primitives, these types are the building blocks for all CIP data types. These are basic types that store a single value, like integers, floats, strings, etc. All of these types can be imported directly from `pycomm3`, for a full list of the types refer to *Data Types*.

```
>>> from pycomm3 import DINT, SHORT_STRING
>>> DINT.encode(112233)
b'i\xb6\x01\x00'
>>> DINT.decode(b'\x12\x34\x56\x78')
2018915346
>>> SHORT_STRING.encode('Hello there!')
b'\x0cHello there!'
```

(continues on next page)

(continued from previous page)

```
>>> SHORT_STRING.decode(b'\x0eGeneral Kenobi')
'General Kenobi'
```

Structure Types

Structures are complex types composed of any number of different elementary or struct member types. The `Struct()` factory is used to create new struct types. To create a new struct, a list of members is required. Members must be *DataType*, either classes (unnamed) or instance (named). Creating named members is really the only time a user would create an instance of a type. When decoding a struct, the value is returned as dictionary of `{member_name: value}`. Any unnamed members will be excluded from the return value, also since the return value is a `dict`, member names should be unique.

```
>>> from pycomm3 import Struct, DINT, STRING, REAL
>>> MyStruct = Struct(DINT('code'), STRING('name'), REAL('value'))
>>> struct_values = {
...     'code': 80,
...     'name': 'my name',
...     'value': 123.45
... }
>>> MyStruct.encode(struct_values)
b'P\x00\x00\x00\x07\x00my namef\xe6\xf6B'
```

```
>>> YourStruct = Struct(DINT, DINT('code'), DINT('type'))
>>> YourStruct.decode(your_bytes)  # assume your_bytes is an encoded YourStruct
{'code': 34, 'type': 73}  # notice the first member is unnamed and not included
```

Both dictionaries and sequences are supported for encoding structs. In the first example, we could have done: `struct_values = [80, 'my name', 123.45]` and gotten the same result. When encoding a struct with multiple unnamed members, using a list of values is the easiest solution. To use a `dict` you must include a `None` key and value to be used for the unnamed members. But, if there are multiple unnamed members of incompatible types, you will have to use a list/sequence instead.

Arrays

Arrays are a homogenous sequence of a *DataType* (either elementary or structs). Any type can be used to create an array of that type using the `[]` operator or the `Array()` factory. There are two important components for an array, the *element type* and the *length*. The *element type* is the *DataType* and the *length* specifies the number of elements. The *length* has 3 options:

Fixed Where the length is specified as an `int`, the array length is fixed to that number of elements.

```
>>> SINT[5].encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])  # notice it only encodes/
↪decodes 5 elements
b'\x01\x02\x03\x04\x05'
>>> SINT[5].decode(b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n')
[1, 2, 3, 4, 5]
```

Derived Where the length is specified as a *DataType*. When decoding an array, the length will be decoded first using the type specified and then decoded that many elements. Encoding will encode however many values are supplied, but does not add the encoded length.

```
>>> SINT[SINT].encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])  # length type is not used␣
↪when encoding
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n'
>>> SINT[SINT].decode(b'\x05\x01\x02\x03\x04\x05\x00\x00\x00')
[1, 2, 3, 4, 5]
```

Unbound Where the length is `None`. When decoding, the array will consume the entire byte buffer and decode as many elements as possible. Encoding will encode however many values are supplied.

```
>>> SINT[None].encode([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])  # length type is not used␣
↪when encoding
b'\x01\x02\x03\x04\x05\x06\x07\x08\t\n'
>>> SINT[None].decode(b'\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0A')
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Logging

This library uses the standard Python logging module. You may configure the logging module as needed. The `DEBUG` level will log every sent/received packed and other diagnostic data. Set the level to higher than `DEBUG` if you only wish to see errors, exceptions, etc. A helper method called `configure_default_logger` is provided to setup basic logging. There are three optional parameters, `level`, `filename`, and `logger`. `level` (default `logging.INFO`) is the logging level. `filename` (default `None`) if set, will also log to the specified file. By default this function only configures the `pycomm3` logger. You can also configure your own custom logger by passing the name in using the `logger` parameter. The `pycomm3` logger is always configured. To configure the root logger set `logger` to an empty string (`''`).

```
from pycomm3.logger import configure_default_logger

configure_default_logger(filename='c:/tmp/pycomm3.log')
```

Produces output similar to:

```
2021-02-26 14:37:41,389 [DEBUG] pycomm3.cip_driver.CIPDriver.open(): Opening connection␣
↪to 192.168.1.236
2021-02-26 14:37:41,393 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Sent:␣
↪RegisterSessionRequestPacket(message=[b'\x01\x00', b'\x00\x00'])
2021-02-26 14:37:41,397 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Received:␣
↪RegisterSessionResponsePacket(session=184719106, error=None)
2021-02-26 14:37:41,398 [INFO] pycomm3.cip_driver.CIPDriver._register_session():␣
↪Session=184719106 has been registered.
2021-02-26 14:37:41,398 [INFO] pycomm3.logix_driver.LogixDriver._initialize_driver():␣
↪Initializing driver...
```

`pycomm3` also uses a custom logging level for verbose logging, this level also prints the contents of each packet send and received. If submitting a bug report, this level of logging is the most helpful.

```
from pycomm3.logger import configure_default_logger, LOG_VERBOSE
configure_default_logger(level=LOG_VERBOSE, filename='c:/tmp/pycomm3.log')
```

Verbose output:

```
2021-02-26 14:42:36,752 [DEBUG] pycomm3.cip_driver.CIPDriver.open(): Opening connection␣
↪to 192.168.1.236
2021-02-26 14:42:36,765 [VERBOSE] pycomm3.cip_driver.CIPDriver._send(): >>> SEND >>>
```

(continues on next page)

(continued from previous page)

```
(0000) 65 00 04 00 00 00 00 00 00 00 00 00 5f 70 79 63     e•••••••••••_pyc
(0010) 6f 6d 6d 5f 00 00 00 00 01 00 00 00                 omm_••••••••
2021-02-26 14:42:36,766 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Sent:␣
↪RegisterSessionRequestPacket(message=[b'\x01\x00', b'\x00\x00'])
2021-02-26 14:42:36,768 [VERBOSE] pycomm3.cip_driver.CIPDriver._receive(): <<< RECEIVE <<
↪<
(0000) 65 00 04 00 02 98 02 0b 00 00 00 00 5f 70 79 63     e•••••••••••_pyc
(0010) 6f 6d 6d 5f 00 00 00 00 01 00 00 00                 omm_••••••••
2021-02-26 14:42:36,768 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Received:␣
↪RegisterSessionResponsePacket(session=184719362, error=None)
2021-02-26 14:42:36,769 [INFO] pycomm3.cip_driver.CIPDriver._register_session():␣
↪Session=184719362 has been registered.
2021-02-26 14:42:36,769 [INFO] pycomm3.logix_driver.LogixDriver._initialize_driver():␣
↪Initializing driver...
2021-02-26 14:42:36,769 [VERBOSE] pycomm3.cip_driver.CIPDriver._send(): >>> SEND >>>
(0000) 63 00 00 00 02 98 02 0b 00 00 00 00 5f 70 79 63     c•••••••••••_pyc
(0010) 6f 6d 6d 5f 00 00 00 00                             omm_••••
2021-02-26 14:42:36,769 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Sent:␣
↪ListIdentityRequestPacket(message=[])
2021-02-26 14:42:36,771 [VERBOSE] pycomm3.cip_driver.CIPDriver._receive(): <<< RECEIVE <<
↪<
(0000) 63 00 45 00 02 98 02 0b 00 00 00 00 5f 70 79 63     c•E•••••••••_pyc
(0010) 6f 6d 6d 5f 00 00 00 00 01 00 0c 00 3f 00 01 00     omm_••••••••?•••
(0020) 00 02 af 12 c0 a8 01 ec 00 00 00 00 00 00 00 00     ••••••••••••••••
(0030) 01 00 0c 00 bf 00 14 13 30 00 90 be 1e c0 1d 31     ••••••••0••••••1
(0040) 37 36 39 2d 4c 32 33 45 2d 51 42 46 43 31 20 45     769-L23E-QBFC1 E
(0050) 74 68 65 72 6e 65 74 20 50 6f 72 74 03              thernet Port•
```

5.1.2 Driver Usage

Using CIPDriver

The *CIPDriver* is the base class for the other drivers, so everything on this page also applies to the other drivers as well.

Discovery and Identification

The *CIPDriver* provides to class methods for discovering and identifying devices. And because they are class methods, they can be used without creating an instance of a driver first. The *CIPDriver.discover()* method will broadcast a request for all devices on the network to identify themselves. This is similar to how the RSLinx Ethernet/IP driver works. It returns a list of dictionaries, where each `dict` is the Identity Object of the device.

```
>>> from pycomm3 import CIPDriver
>>> CIPDriver.discover()
```

For example, here is a response with 2 devices discovered:

```
[{'encap_protocol_version': 1, 'ip_address': '10.10.0.120', 'vendor': 'Rockwell␣
↪Automation/Allen-Bradley',
  'product_type': 'Communications Adapter', 'product_code': 185, 'revision': {'major': 2,
↪  'minor': 7},
```

(continues on next page)

(continued from previous page)

```
  'status': b'T\x00', 'serial': 'aabbcdd', 'product_name': '1763-L16BWA B/7.00', 'state
↪': 0},
 {'encap_protocol_version': 1, 'ip_address': '10.10.1.100', 'vendor': 'Rockwell␣
↪Automation/Allen-Bradley',
  'product_type': 'Communications Adapter', 'product_code': 191, 'revision': {'major':␣
↪20, 'minor': 19},
  'status': b'0\x00', 'serial': 'eeffgghh', 'product_name': '1769-L23E-QBFC1 Ethernet␣
↪Port', 'state': 3}]
```

The *CIPDriver.list_identity()* method is similar, but can be used to identify a specific device. Instead of broadcasting the request to every device, it requires a `path` to send the request to. This `path` argument is the same type of CIP path used in creating a driver and detailed in *Creating a Driver*.

```
>>> from pycomm3 import CIPDriver
>>> CIPDriver.list_identity('10.10.0.120')
{'encap_protocol_version': 1, 'ip_address': '10.10.0.120', 'vendor': 'Rockwell␣
↪Automation/Allen-Bradley',
'product_type': 'Communications Adapter', 'product_code': 185, 'revision': {'major': 2,
↪'minor': 7},
'status': b'T\x00', 'serial': 'aabbcdd', 'product_name': '1763-L16BWA B/7.00', 'state':␣
↪0}
>>> CIPDriver.list_identity('10.10.1.100')
{'encap_protocol_version': 1, 'ip_address': '10.10.1.100', 'vendor': 'Rockwell␣
↪Automation/Allen-Bradley',
'product_type': 'Communications Adapter', 'product_code': 191, 'revision': {'major': 20,
↪'minor': 19},
'status': b'0\x00', 'serial': 'eeffgghh', 'product_name': '1769-L23E-QBFC1 Ethernet Port
↪', 'state': 3}
```

Module Identification

For rack-based devices, the *CIPDriver.get_module_info()* method will return the identity for a `slot` in the rack. This method is *not* a class method, so it does require an instance of the driver to be created.

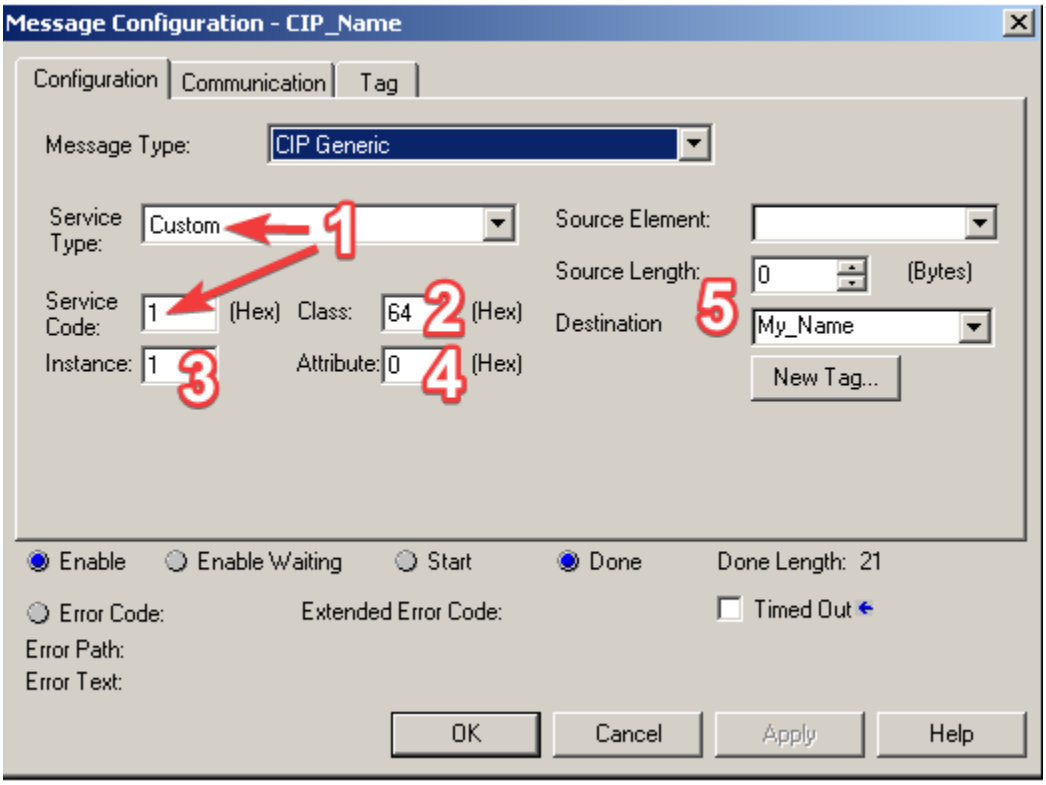
```
>>> from pycomm3 import CIPDriver
>>> driver = CIPDriver('10.10.1.100')
>>> driver.open()
>>> driver.get_module_info(0)  # Slot 0: PLC
{'vendor': 'Rockwell Automation/Allen-Bradley', 'product_type': 'Programmable Logic␣
↪Controller', 'product_code': 51,
'revision': {'major': 16, 'minor': 22}, 'status': b'`\x10', 'serial': '00000000',
'product_name': '1756-L55/A 1756-M13/A LOGIX5555'}
>>> driver.get_module_info(1) # Slot 1: EN2T
{'vendor': 'Rockwell Automation/Allen-Bradley', 'product_type': 'Communications Adapter',
↪ 'product_code': 166,
'revision': {'major': 5, 'minor': 8}, 'status': b'0\x00', 'serial': '00000000', 'product_
↪name': '1756-EN2T/B'}
>>> driver.close()
```

Generic Messaging

Generic messaging is a key feature of `pycomm3`, it allows the user to send custom CIP messages or implement features not included in one of the drivers. In fact, many features available in the drivers are implemented using the *`generic_message()`* method. This method operates in a similar way to *CIP Generic* messages in Logix with the `MSG` instruction. For more examples see the *Generic Messaging* section.

To demonstrate how a generic message can be used, below is the process that was used to implement the *`get_plc_name()`* feature for the *`LogixDriver`*.

First, the Obtaining the Controller's Program Name article from the Rockwell Knowledge Base shows how to configure a `MESSAGE` to read the program name from a PLC. It contains all the information we need: CIP service, class, instance, etc.



1. The service type is `0x01`, which is the `Get_Attributes_All` service define in the *Common Industrial Protocol Specification, Volume 1, Chapter 4: CIP Object Model*. See *CIP Services and Class Codes* for the predefined CIP services, classes, and other objects available in `pycomm3`. If the service is not already defined, you use either an `int` or a `bytes` string (`0x01`, `1`, `b'\x01`).
2. The class code, `0x64` is not named in the doc, but is defined as `ClassCode.program_name`.
3. The instance number of the class we want, `1`.
4. The attribute is `0`, so we can ignore it and not set the `attribute` parameter.
5. Since we're not in the PLC, we're not storing the response in a tag. If we set the `data_type` parameter to a `DataType`, that type will be used to decode the response. Else, the raw response `bytes` will be returned.

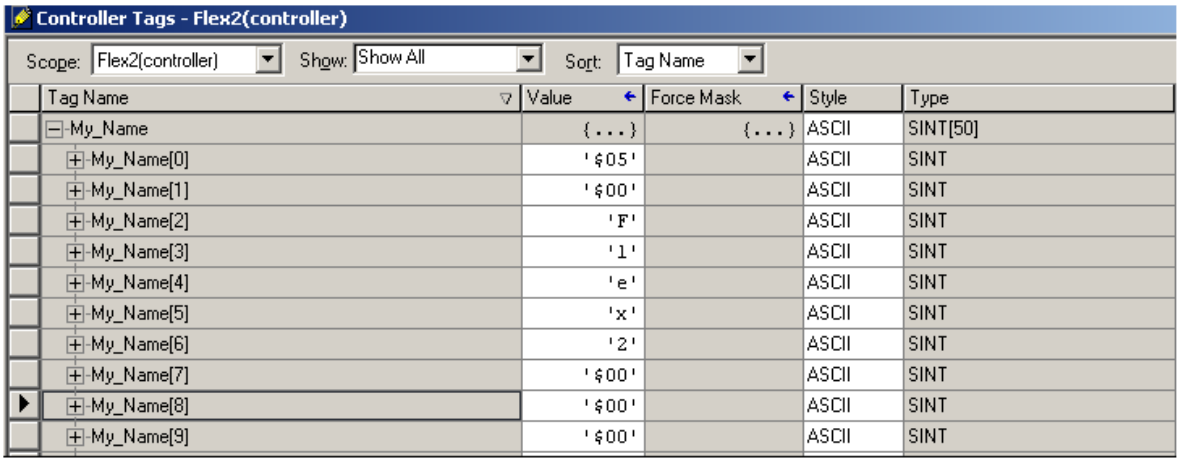
Next, the screenshot below contains enough information for us to determine the data type that can be used to decode the response.

The result will be deposited into a SINT array, which can very with length depending on the names stored.

The first value is the string length.

The Next is a NULL terminator marking the beginning of the string.

The following values starting at My_Name[2] are the controllers name characters.



Tag Name	Value	Force Mask	Style	Type
My_Name	{...}	{...}	ASCII	SINT[50]
My_Name[0]	'$05'		ASCII	SINT
My_Name[1]	'$00'		ASCII	SINT
My_Name[2]	'F'		ASCII	SINT
My_Name[3]	'l'		ASCII	SINT
My_Name[4]	'e'		ASCII	SINT
My_Name[5]	'x'		ASCII	SINT
My_Name[6]	'2'		ASCII	SINT
My_Name[7]	'$00'		ASCII	SINT
My_Name[8]	'$00'		ASCII	SINT
My_Name[9]	'$00'		ASCII	SINT

While the doc doesn't specifically say the response type, it's shows that it is stored in a `SINT[50]`. The first two bytes contains the length of the string, which corresponds to a integer(`INT` or `UINT`). Then the string data is stored in the remainder of the array, since PLCs are limited to fixed-size arrays the destination tag needs to be long enough to contain the maximum size possible. In Python we do not have that limitation, but this information tells us that the response is a *string, with 1 byte per character, and the length of the string is stored in the first 2 bytes*. That corresponds to the CIP `STRING` data type, which is a standard type that is already defined and we can just use.

Taking this information, we were able configure the *`generic_message()`* method to read the PLC program name:

```
    @with_forward_open
    def get_plc_name(self) -> str:
        """
        Requests the name of the program running in the PLC. Uses KB `23341`_ for␣
↪implementation.

        .. _23341: https://rockwellautomation.custhelp.com/app/answers/answer_view/a_id/
↪23341

        :return:  the controller program name
        """

        try:
            response = self.generic_message(
                service=Services.get_attributes_all,
                class_code=ClassCode.program_name,
                instance=1,
                data_type=STRING,
                name="get_plc_name",
            )
```

(continues on next page)

(continued from previous page)

```
            if not response:
                raise ResponseError(f"response did not return valid data - {response.
↪error}")

            self._info["name"] = response.value
            return self._info["name"]
        except Exception as err:
            raise ResponseError("failed to get the plc name") from err
```

Tip: Setting the `name` parameter is helpful because it will be used by the built in logging and can help differentiate between calls:

```
2021-03-09 18:09:50,802 [INFO] pycomm3.cip_driver.CIPDriver.generic_message(): Sending␣
↪generic message: get_plc_name
2021-03-09 18:09:50,802 [VERBOSE] pycomm3.cip_driver.CIPDriver._send(): >>> SEND >>>
(0000) 70 00 1c 00 00 0b 02 0b 00 00 00 00 5f 70 79 63     p•••••••••••_pyc
(0010) 6f 6d 6d 5f 00 00 00 00 00 00 00 00 0a 00 02 00     omm_••••••••••••
(0020) a1 00 04 00 c1 04 35 01 b1 00 08 00 53 00 01 02     ••••••5•••••S•••
(0030) 20 64 24 01                                          d$•
2021-03-09 18:09:50,803 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Sent:␣
↪GenericConnectedRequestPacket(message=[b'S\x00', b'\x01', b'\x02 d$\x01', b''])
2021-03-09 18:09:50,807 [VERBOSE] pycomm3.cip_driver.CIPDriver._receive(): <<< RECEIVE <<
↪<
(0000) 70 00 36 00 00 0b 02 0b 00 00 00 00 00 00 00 00     p•6•••••••••••••
(0010) 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00     ••••••••••••••••
(0020) a1 00 04 00 4a b7 cb 55 b1 00 22 00 53 00 81 00     ••••J••U••"•S•••
(0030) 00 00 0c 00 70 79 63 6f 6d 6d 33 5f 64 65 6d 6f     ••••pycomm3_demo
(0040) 00 00 00 00 02 00 01 00 64 00 02 00 09 00           ••••••••d•••••
2021-03-09 18:09:50,807 [DEBUG] pycomm3.cip_driver.CIPDriver.send(): Received:␣
↪GenericConnectedResponsePacket(service=b'\x01', command=b'p\x00', error=None)
2021-03-09 18:09:50,807 [INFO] pycomm3.cip_driver.CIPDriver.generic_message(): Generic␣
↪message 'get_plc_name' completed
```

Using LogixDriver

Tags and Data Types

When creating the driver it will automatically upload all of the controller scope tag and their data type definitions. These definitions are required for the *read()* and *write()* methods to function. Those methods abstract away a lot of the details required for actually implementing the Ethernet/IP protocol. Uploading the tags could take a few seconds depending on the size of program and the network. It was decided this small upfront overhead provided a greater benefit to the user since they would not have to worry about specific implementation details for different types of tags. The `init_tags` kwarg is `True` by default, meaning that all of the controller scoped tags will be uploaded. `init_program_tags` is a separate flag to control whether or not all the program-scoped tags are uploaded as well. By default, `init_program_tags` is `True`, set to `False` to disable and only upload controller-scoped tags.

Below shows how the init tag options are equivalent to calling the *get_tag_list()* method.

```
>>> plc1 = LogixDriver('10.20.30.100')
>>> plc2 = LogixDriver('10.20.30.100', init_tags=False)
>>> plc2.get_tag_list()
>>> plc1.tags == plc2.tags
True
>>> plc3 = LogixDriver('10.20.30.100', init_program_tags=True)
>>> plc4 = LogixDriver('10.20.30.100')
>>> plc4.get_tag_list(program='*')  # '*' means all programs
>>> plc3.tags == plc4.tags
True
```

Tag Structure

Each tag definition is a dict containing all the details retrieved from the PLC. *get_tag_list()* returns a list of dicts for the tag list while the *LogixDriver.tags* property stores them as a dict of `{tag name: definition}`.

Tag Definition Properties:

tag_name Symbolic name of the tag

instance_id Internal PLC identifier for the tag. Used for reads/writes on v21+ controllers. Saves space in packet by not requiring the full tag name to be encoded into the request.

tag_type

- `'atomic'` base data types like BOOL, DINT, REAL, etc.
- `'struct'` complex data types like STRING, TIMER, PID, etc as well as UDTs and AOIs.

data_type

- `'DINT'/'REAL'`/etc name of data type for atomic types
- `{data type definition}` for structures, detailed in *Structure Definitions*

data_type_name

- the string name of the data type: `'DINT'/'REAL'/'TIMER'/'MyCoolUDT'`

string **Optional** string size if the tag is a STRING type (or custom string)

external_access `'Read/Write'/'Read Only'/'None'` matches the External Access tag property in the PLC

dim number dimensions defined for the tag

- `0` - not an array
- `1-3` - a 1 to 3 dimension array tag, e.g. `DINT[5] -> 1, DINT[5,5] -> 2, DINT[5,5,5] -> 3`

dimensions length of each dimension defined, `0` if dimension does not exist. `[dim0, dim1, dim2]`

- `DINT[5] -> [5, 0, 0]`
- `DINT[5, 10] -> [5, 10, 0]`
- `DINT[5, 10, 15] -> [5, 10, 15]`

alias `True/False` if the tag is an alias to another.

> **Note:** This is not documented, but an educated guess found through trial and error.

type_class the *DataType* that was created for this tag

Structure Definitions

While uploading the tag list, any tags with complex data types will have the full definition of structure uploaded as well. Inside a tag definition, the *data_type* attribute will be a dict containing the structure definition. The *LogixDriver.data_types* property also provides access to these definitions as a dict of `{data type name: definition}`.

Data Type Properties:

name Name of the data type, UDT, AOI, or builtin structure data types

attributes List of names for each attribute in the structure. Does not include internal tags not shown in Logix, like the host DINT tag that BOOL attributes are mapped to.

template `dict` with template definition. Used internally within LogixDriver, allows reading/writing of full structs and allows the read/write methods to monitor the request/response size.

internal_tags A `dict` with each attribute (including internal, not shown in Logix attributes) of the structure containing the definition for the attribute, `{attribute: {definition}}`.

Definition:

tag_type Same as *Tag Structure*

data_type Same as *Tag Structure*

data_type_name Same as *Tag Structure*

string Same as *Tag Structure*

offset Location/Byte offset of this tag's data in the response data.

bit **Optional** BOOL tags are aliased to internal hidden integer tags, this indicates which bit it is aliased to.

array **Optional** Length of the array if this tag is an array, `0` if not an array,

---

Note: `attributes` and `internal_tags` do **NOT** include InOut parameters.

---

type_class The *DataType* type that was created to represent this structure

Reading/Writing Tags

All reading and writing is handled by the *read()* and *write()* methods. The original pycomm and other similar libraries will have different methods for handling different types like strings and arrays, this is not necessary in `pycomm3` due to uploading the tag list and creation of a *DataType* class for each type. Both methods accept any number of tags, they will automatically use the *Multiple Service Packet (0x0A)* service and track the request/return data size making sure to stay below the connection size. If there is a tag value that cannot fit within the request/reply packet, it will automatically handle that tag independently using the *Read Tag Fragmented (0x52)* or *Write Tag Fragmented (0x53)* requests. Users do not have to worry about the number of tags or their size in any single request, this is all handled automatically by the driver.

Program-Scoped Tags

Program-scoped tag names use the format *Program:<program>.<tag>*. For example, to access a tag named *SomeTag* in the program *MainProgram* you would use *Program:MainProgram.SomeTag* in the request. The tag list uploaded by the driver will also keep this format for the tag names.

Array Tags

To access an index of an array, include the index inside square brackets after the tag name. The format is the same as in Logix, where multiple dimensions are comma separated, e.g. `an_array[5]` for the 5th element of `an_array` or `array2[1,0]` to access the first element of the second dimension of `array2`. Not specifying an index is equivalent to index 0, i.e `array == array[0]`.

Whether reading or writing, the number of elements needs to be specified. To do so, specify the number of elements inside curly braces at the end of the tag name, e.g. `an_array{5}` for 5-elements of `an_array`. If omitted, the number of elements is assumed to be 1, i.e. `an_array == an_array[0] == an_array[0]{1}`. Only a single element count is used. For 2 and 3 dimensional arrays, the element count is the total number of elements across all dimensions. The tables below show a couple examples of how the element count works for multi-dimension arrays.

array (`DINT[3, 2]`)	array{4}	array[1,1]{3}
array[0, 0]	X	
array[0, 1]	X	
array[1, 0]	X	
array[1, 1]	X	X
array[2, 0]		X
array[2, 1]		X

array (`SINT[2, 2, 2]`)	array{4}	array[0,1,0]{5}
array[0, 0, 0]	X	
array[0, 0, 1]	X	
array[0, 1, 0]	X	X
array[0, 1, 1]	X	X
array[1, 0, 0]		X
array[1, 0, 1]		X
array[1, 1, 0]		X
array[1, 1, 1]		

BOOL Arrays

BOOL arrays work a little differently due them being implemented as DWORD arrays in the PLC. (That is the reason you can only make BOOL arrays in multiples of 32, DWORDs are 32 bits.) The element count in the request (`'{#}'`) represents the number of BOOL elements. To write multiple elements to a BOOL array, you must write the entire underlying DWORD element. This means the list of values must be in multiples of 32 and the starting index must also be multiples of 32, e.g. `'bools{32}'`, `'bools[32]{64}'`. There is no limitation on reading multiple elements or reading and writing a single element.

Reading Tags

LogixDriver.read() accepts any number of tags, all that is required is the tag names.Reading of entire structures is support as long as none of the attributes have an external access of *None*. To read a structure, just request the base name and the `value` for the `Tag` object will a a dict of `{attribute: value}`

Read an atomic tag

```
>>> plc.read('dint_tag')
Tag(tag='dint_tag', value=0, type='DINT', error=None)
```

Read multiple tags

```
>>> plc.read('tag_1', 'tag_2', 'tag_3')
[Tag(tag='tag_1', value=100, type='INT', error=None), Tag(tag='tag_2', value=True, type=
↪'BOOL', error=None), ...]
```

Read a structure

```
>>> plc.read('simple_udt')
Tag(tag='simple_udt', value={'attr1': 0, 'attr2': False, 'attr3': 1.234}, type='SimpleUDT
↪', error=None)
```

Read arrays

```
>>> plc.read('dint_array{5}')  # starts at index 0
Tag(tag='dint_array', value=[1, 2, 3, 4, 5], type='DINT[5]', error=None)
>>> plc.read('dint_array[20]{3}') # read 3 elements starting at index 20
Tag(tag='dint_array[20]', value=[20, 21, 22], type='DINT[3]', error=None)
```

Verify all reads were successful

```
>>> tag_list = ['tag1', 'tag2', ...]
>>> results = plc.read(*tag_list)
>>> if all(results):
...     print('All tags read successfully')
All tags read successfully
```

Writing Tags

LogixDriver.write() method accepts any number of tag-value pairs of the tag name and value to be written. For writing a single tag, you can do `write(<tag name>, <value>)`, but for multiple tags a sequence of tag-value tuples is required (`write((<tag1>, <value1>), (<tag2>, <value2>))`). For arrays, the value should be a list of the values to write. A `RequestError` will be raised if the value list is too short, else it will be truncated if too long. Writing a structure is supported as long as all attributes have Read/Write external access. The value for a struct should be a `dict` of `{<attribute name>: <value>}`, nesting as needed. It is not recommended to write full structures for builtin types, like `TIMER`, `PID`, etc.

Write a tag

```
>>> plc.write('dint_tag', 100)
Tag(tag='dint_tag', value=100, type='DINT', error=None)
```

Write many tags

```
>>> plc.write(('tag_1', 1), ('tag_2', True), ('tag_3', 1.234))
[Tag(tag='tag_1', value=1, type='INT', error=None), Tag(tag='tag_2', value=True, type=
↪'BOOL', error=None), ...]
```

Write arrays

```
>>> plc.write('dint_array{10}', list(range(10)))  # starts at index 0
Tag(tag='dint_array', value=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], type='DINT[10]', error=None)
>>> plc.write(('dint_array[10]{3}', [10, 11, 12]))  # write 3 elements starting at index␣
↪10
Tag(tag='dint_array[10]', value=[10, 11, 12], type='DINT[3]', error=None)
```

Write structures

```
>>> plc.write('my_udt', {'attr1': 100, 'attr2': [1, 2, 3, 4]})
Tag(tag='my_udt', value={'attr1': 100, 'attr2': [1, 2, 3, 4]}, type='MyUDT', error=None)
```

Check if all writes were successful

```
>>> tag_values = [('tag1', 10), ('tag2', True), ('tag3', 12.34)]
>>> results = plc.write(*tag_values)
>>> if all(results):
...     print('All tags written successfully')
All tags written successfully
```

String Tags

Strings are technically structures within the PLC, but are treated as atomic types in this library. There is no need to handle the `LEN` and `DATA` attributes, the structure is converted to/from Python `str` objects transparently. Any structures that contain only a DINT-`LEN` and a SINT[]-`DATA` attributes will be automatically treated as string tags. This allows the builtin STRING types plus custom strings to be handled automatically. Strings that are longer than the plc tag will be truncated when writing.

```
>>> plc.read('string_tag')
Tag(tag='string_tag', value='Hello World!', type='STRING', error=None)
>>> plc.write(('short_string_tag', 'Test Write'))
Tag(tag='short_string_tag', value='Test Write', type='STRING20', error=None)
```

Using SLCDriver

TODO

This document.

5.1.3 Examples

Basic Reading and Writing Tag Examples

Basic Reading

Reading a single tag returns a Tag object.

```
def read_single():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read('DINT1')
```

```
>>> read_single()
Tag(tag='DINT1', value=20, type='DINT', error=None)
```

Reading multiple tags returns a list of Tag objects.

```
def read_multiple():
    tags = ['DINT1', 'SINT1', 'REAL1']
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read(*tags)
```

```
>>> read_multiple()
[Tag(tag='DINT1', value=20, type='DINT', error=None), Tag(tag='SINT1', value=5,
↪ type='SINT', error=None), Tag(tag='REAL1', value=100.0009994506836, type=
↪'REAL', error=None)]
```

An array is represented in a single Tag object, but the `value` attribute is a list.

```
def read_array():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read('DINT_ARY1{5}')
```

```
def read_array_slice():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read('DINT_ARY1[50]{5}')
```

```
>>> read_array()
Tag(tag='DINT_ARY1', value=[0, 1000, 2000, 3000, 4000], type='DINT[5]',␣
↪error=None)
>>> read_array_slice()
Tag(tag='DINT_ARY1[50]', value=[50000, 51000, 52000, 53000, 54000], type=
↪'DINT[5]', error=None)
```

You can read strings just like a normal value, no need to handle the `LEN` and `DATA` attributes individually.

```
def read_strings():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read('STRING1', 'STRING_ARY1[2]{2}')
```

```
>>> read_strings()
[Tag(tag='STRING1', value='A Test String', type='STRING', error=None), Tag(tag=
↪'STRING_ARY1[2]', value=['THIRD', 'FoUrTh'], type='STRING[2]', error=None)]
```

Structures can be read as a whole, assuming that no attributes have External Access set to None. Structure tags will be a single Tag object, but the `value` attribute will be a `dict` of `{attribute: value}`.

```
def read_udt():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read('SimpleUDT1_1')
```

```
def read_timer():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.read('TIMER1')
```

```
>>> read_udt()
Tag(tag='SimpleUDT1_1', value={'bool': True, 'sint': 100, 'int': -32768, 'dint
↪': -1, 'real': 0.0}, type='SimpleUDT1', error=None)
>>> read_timer()
Tag(tag='TIMER1', value={'CTL': [False, False, False, False, False, False,␣
↪False, False, False, False, False,
                                 False, False, False, False, False, False,␣
↪False, False, False, False, False,
                                 False, False, False, False, False, False,␣
↪False, True, True, False],
                          'PRE': 30000, 'ACC': 30200, 'EN': False, 'TT': True,
↪'DN': True}, type='TIMER', error=None)
```

Note: Most builtin data types appear to have a BOOL array (or DWORD) attribute called CTL that is not shown in the Logix tag browser.

Basic Writing

Writing a single tag returns a single Tag object response.

```
def write_single():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.write(('DINT2', 100_000_000))
```

```
>>> write_single()
Tag(tag='DINT2', value=100000000, type='DINT', error=None)
```

Writing multiple tags will return a list of Tag objects.

```
def write_multiple():
    with LogixDriver('10.61.50.4/10') as plc:
        return plc.write(('REAL2', 25.2), ('STRING3', 'A test for writing to a␣
↪string.'))
```

```
>>> write_multiple()
[Tag(tag='REAL2', value=25.2, type='REAL', error=None), Tag(tag='STRING3',␣
↪value='A test for writing to a string.', type='STRING', error=None)]
```

Writing a whole structure is possible too. As with reading, all attributes are required to NOT have an External Access of None. Also, when writing a structure your value must match the structure exactly and provide data for all attributes.

The value should be a list of values or a dict of attribute name and value, nesting as needed for arrays or other structures with the target. This example shows a simple recipe UDT:

Attribute	Data Type
Enabled	BOOL
OpCodes	DINT[10]
Targets	REAL[10]
StepDescriptions	STRING[10]
TargetUnits	STRING8[10]
Name	STRING

```
def write_structure():
    with LogixDriver('10.61.50.4/10') as plc:
        recipe_data = {
            'Enabled': True,
            'OpCodes': [10, 11, 4, 20, 6, 20, 6, 30, 5, 0],
            'Targets': [100, 500, 85, 5, 15, 10.5, 20, 0, 0, 0],
            'StepDescriptions': ['Set Water Temperature',
                                 'Heated Water',
                                 'Start Agitator',
                                 'Hand Add - Flavor Part 1',
                                 'Timed Mix',
                                 'Hand Add - Flavor Part 2',
                                 'Timed Mix',
                                 'Transfer to Storage Tank',
                                 'Disable Agitator',
                                 ''],
            'TargetUnits': ['°F', 'lbs', '%', 'gal', 'min', 'lbs', 'min', '', '
↪', ''],
            'Name': 'Our Fictional Recipe',
        }

        plc.write(('Example_Recipe', recipe_data))
```

Examples of Working with the Tag List

Data Types

For UDT/AOI or built-in structure data-types, information and definitions are stored in the `data_types` property. This property allow you to query the PLC to determine what types of tags it may contain. For details on the contents of a data type definition view *Structure Definitions*.

Print out the public attributes for all structure types in the PLC:

```
def find_attributes():
    with LogixDriver('10.61.50.4/10') as plc:
        ...  # do nothing, we're just letting the plc initialize the tag list

    for typ in plc.data_types:
        print(f'{typ} attributes: ', plc.data_types[typ]['attributes'])
```

```
>>> find_attributes()
STRING attributes:  ['LEN', 'DATA']
TIMER attributes:  ['CTL', 'PRE', 'ACC', 'EN', 'TT', 'DN']
CONTROL attributes:  ['CTL', 'LEN', 'POS', 'EN', 'EU', 'DN', 'EM', 'ER', 'UL',
↪'IN', 'FD']
DateTime attributes:  ['Yr', 'Mo', 'Da', 'Hr', 'Min', 'Sec', 'uSec']
...
```

Tag List

Part of the requirement for reading/writing tags is knowing the tag definitions stored in the PLC so that user does not need to provide any information about the tag besides it's name. By default, the tag list is uploaded on creation of the LogixDriver, for details reference the *LogixDriver API*.

Example showing how the tag list is stored:

```
def tag_list_equal():
    with LogixDriver('10.61.50.4/10') as plc:
        tag_list = plc.get_tag_list()
        if {tag['tag_name']: tag for tag in tag_list} == plc.tags:
            print('They are the same!')

    with LogixDriver('10.61.50.4/10', init_tags=False) as plc2:
        plc2.get_tag_list()

    if plc.tags == plc2.tags:
        print('Calling get_tag_list() does the same thing.')
    else:
        print('Calling get_tag_list() does NOT do the same.')
```

```
>>> tag_list_equal()
They are the same!
Calling get_tag_list() does the same thing.
```

Filtering

There are multiple properties of tags that can be used to locate and filter down the tag list. For available properties, reference *Tag Structure*. Examples below show some methods for filtering the tag list.

Finding all PID tags:

```
def find_pids():
    with LogixDriver('10.61.50.4/10') as plc:

        # PIDs are structures, the data_type attribute will be a dict with␣
↪data type definition.
        # For tag types of 'atomic' the data type will a string, we need to skip␣
↪those first.
        # Then we can just look for tags whose data type name matches 'PID'
        pid_tags = [
            tag
```

(continues on next page)

(continued from previous page)

```
            for tag, _def in plc.tags.items()
            if _def['data_type_name'] == 'PID'
        ]

        print(pid_tags)
```

```
>>> find_pids()
['FIC100_PID', 'TIC100_PID']
```

Generic Messaging

The `LogixDriver.generic_message()` works in a similar way to the MSG instruction in Logix. It allows the user to perform messaging services not directly implemented in the library. It is also used internally to implement some of the CIP services used by the library (Forward Open, get/set PLC time, etc).

Accessing Drive Parameters

While a drive may not be a PLC, we can use generic messaging to read parameters from it. The target drive is a PowerFlex 525 and using this Rockwell KB Article we can get the appropriate parameters to read/write parameters from the drive.

```
def read_pf525_parameter():
    drive_path = '10.10.10.100/bp/1/enet/192.168.1.55'

    with CIPDriver(drive_path) as drive:
        param = drive.generic_message(
            service=Services.get_attribute_single,
            class_code=b'\x93',
            instance=41,  # Parameter 41 = Accel Time
            attribute=b'\x09',
            data_type=INT,
            connected=False,
            unconnected_send=True,
            route_path=True,
            name='pf525_param'
        )
        print(param)
```

```
>>> read_pf525_parameter()
pf525_param, 500, None, None
```

```
def write_pf525_parameter():
    drive_path = '10.10.10.100/bp/1/enet/192.168.1.55'

    with CIPDriver(drive_path) as drive:
        drive.generic_message(
            service=Services.set_attribute_single,
            class_code=b'\x93',
            instance=41,  # Parameter 41 = Accel Time
```

(continues on next page)

(continued from previous page)

```
            attribute=b'\x09',
            request_data=INT.encode(500),  # = 5 seconds * 100
            connected=False,
            unconnected_send=True,
            route_path=True,
            name='pf525_param'
        )
```

Reading Device Statuses

ENBT/EN2T OK LED Status

This message will get the current status of the OK LED from and ENBT or EN2T module.

```
def enbt_ok_led_status():
    message_path = '10.10.10.100/bp/2'

    with CIPDriver(message_path) as device:
        data = device.generic_message(
            service=Services.get_attribute_single,
            class_code=b'\x01',  # Values from RA Knowledgebase
            instance=1,  # Values from RA Knowledgebase
            attribute=5,  # Values from RA Knowledgebase
            data_type=INT,
            connected=False,
            unconnected_send=True,
            route_path=True,
            name='OK LED Status'
        )
        # The LED Status is returned as a binary representation on bits 4, 5,␣
↪6, and 7. The decimal equivalents are:
        # 0 = Solid Red, 64 = Flashing Red, and 96 = Solid Green. The ENBT/
↪EN2T do not display link lost through the OK LED.
        statuses = {
            0: 'solid red',
            64: 'flashing red',
            96: 'solid green'
        }
        print(statuses.get(data.value), 'unknown')
```

Link Status

This message will read the current link status for any ethernet module.

```
def link_status():
    message_path = '10.10.10.100/bp/2'

    with CIPDriver(message_path) as device:
        data = device.generic_message(
```

(continues on next page)

(continued from previous page)

```
            service=Services.get_attribute_single,
            class_code=b'\xf6',  # Values from RA Knowledgebase
            instance=1,  # For multiport devices, change to "2" for second␣
↪port, "3" for third port.
                         # For CompactLogix, front port is "1" and back port␣
↪is "2".
            attribute=2,  # Values from RA Knowledgebase
            data_type=INT,
            connected=False,
            unconnected_send=True,
            route_path=True,
            name='LinkStatus'
        )
        # Prints the binary representation of the link status. The definition␣
↪of the bits are:
        #   Bit 0 - Link Status - 0 means inactive link (Link Lost), 1 means␣
↪active link.
        #   Bit 1 - Half/Full Duplex - 0 means half duplex, 1 means full duplex
        #   Bit 2 to 4 - Binary representation of auto-negotiation and speed␣
↪detection status:
        #       0 = Auto-negotiation in progress
        #       1 = Auto-negotiation and speed detection failed
        #       2 = Auto-negotiation failed, speed detected
        #       3 = Auto-negotiation successful and speed detected
        #       4 = Manually forced speed and duplex
        #   Bit 5 - Setting Requires Reset - if 1, a manual setting requires␣
↪resetting of the module
        #   Bit 6 - Local Hardware Fault - 0 indicates no hardware faults, 1␣
↪indicates a fault detected.
        print(bin(data.value))
```

Stratix Switch Power Status

This message will read the current power status for both power inputs on a Stratix switch.

```
def stratix_power_status():
    message_path = '10.10.10.100/bp/2/enet/192.168.1.1'

    with CIPDriver(message_path) as device:
        data = device.generic_message(
            service=b'\x0e',
            class_code=863,  # use decimal representation of hex class code
            instance=1,
            attribute=8,
            connected=False,
            unconnected_send=True,
            route_path=True,
            data_type=INT,
            name='Power Status'
        )
        # Returns a binary representation of the power status. Bit 0 is PWR A,␣
↪Bit 1 is PWR B. If 1, power is applied. If 0, power is off.
```

(continues on next page)

(continued from previous page)

```
        pwr_a = 'on' if data.value & 0b_1 else 'off'
        pwr_b = 'on' if data.value & 0b_10 else 'off'
        print(f'PWR A: {pwr_a}, PWR B: {pwr_b}')
```

IP Configuration

Static/DHCP/BOOTP Status

This message will read the IP setting configuration type from an ethernet module.

```
def ip_config():
    message_path = '10.10.10.100/bp/2'

    with CIPDriver(message_path) as plc:  # L85
        data = plc.generic_message(
            service=b'\x0e',
            class_code=b'\xf5',
            instance=1,
            attribute=3,
            connected=False,
            unconnected_send=True,
            route_path=True,
            data_type=INT,
            name='IP_config'
        )

        statuses = {
            0b_0000: 'static',
            0b_0001: 'BOOTP',
            0b_0010: 'DHCP'
        }

        ip_status = data.value & 0b_1111  # only need the first 4 bits
        print(statuses.get(ip_status, 'unknown'))
```

Communication Module MAC Address

This message will read the MAC address of ethernet module where the current connection is opened.

```
def get_mac_address():
    with CIPDriver('10.10.10.100') as plc:
        response = plc.generic_message(
            service=Services.get_attribute_single,
            class_code=ClassCode.ethernet_link,
            instance=1,
            attribute=3,
            data_type=USINT[6],
            connected=False
        )
```

(continues on next page)

(continued from previous page)

```
        if response:
            return ':'.join(f'{x:0>2x}' for x in response.value)
        else:
            print(f'error getting MAC address - {response.error}')
```

Upload EDS File

This example shows how to use generic messaging to upload and save an EDS file from a device.

```
from pycomm3 import (CIPDriver, Services, ClassCode,  FileObjectServices,␣
↪FileObjectInstances,
                     FileObjectInstanceAttributes, Struct, UDINT, USINT, n_
↪bytes)
import itertools
import gzip
from pathlib import Path

SAVE_PATH = Path.home()


def upload_eds():
    """
    Uploads the EDS and ICO files from the device and saves the files.
    """
    with CIPDriver('192.168.1.236') as driver:
        if initiate_transfer(driver):
            file_data = upload_file(driver)
            encoding = get_file_encoding(driver)

            if encoding == 'zlib':
                # in this case the file has both the eds and ico files in it
                files = decompress_eds(file_data)

                for filename, file_data in files.items():
                    file_path = SAVE_PATH / filename
                    file_path.write_bytes(file_data)

            elif encoding == 'binary':
                file_name = get_file_name(driver)
                file_path = SAVE_PATH / file_name
                file_path.write_bytes(file_data)
            else:
                print('Unsupported Encoding')
        else:
            print('Failed to initiate transfer')


def initiate_transfer(driver):
    """
    Initiates the transfer with the device
```

(continues on next page)

(continued from previous page)

```
    """
    resp = driver.generic_message(
        service=FileObjectServices.initiate_upload,
        class_code=ClassCode.file_object,
        instance=FileObjectInstances.eds_file_and_icon,
        route_path=True,
        unconnected_send=True,
        connected=False,
        request_data=b'\xFF',  # max transfer size
        data_type=Struct(UDINT('FileSize'), USINT('TransferSize'))
    )
    return resp


def upload_file(driver):
    contents = b''

    for i in itertools.cycle(range(256)):
        resp = driver.generic_message(
            service=FileObjectServices.upload_transfer,
            class_code=ClassCode.file_object,
            instance=FileObjectInstances.eds_file_and_icon,
            route_path=True,
            unconnected_send=True,
            connected=False,
            request_data=USINT.encode(i),
            data_type=Struct(USINT('TransferNumber'), USINT('PacketType'), n_
↪bytes(-1, 'FileData'))

        )

        if resp:
            packet_type = resp.value['PacketType']
            data = resp.value['FileData']

            contents += data

            # CIP Vol 1 Section 5-42.4.5
            # 0 - first packet
            # 1 - middle packet
            # 2 - last packet
            # 3 - Abort transfer
            # 4 - first & last packet
            # 5-255 - Reserved
            if packet_type not in (0, 1):
                break
        else:
            print(f'failed response {resp}')
            break

    contents = contents[:-2]  # strip off checksum
    return contents
```

(continues on next page)

(continued from previous page)

```
def get_file_encoding(driver):
    """
    get the encoding format for the eds file object
    """
    attr = FileObjectInstanceAttributes.file_encoding_format

    resp = driver.generic_message(
        service=Services.get_attribute_single,
        class_code=ClassCode.file_object,
        attribute=attr.attr_id,
        instance=FileObjectInstances.eds_file_and_icon,
        route_path=True,
        unconnected_send=True,
        connected=False,
        data_type=attr.data_type,
    )
    _enc_code = resp.value if resp else None
    EDS_ENCODINGS = {
        0: 'binary',
        1: 'zlib'
    }
    file_encoding = EDS_ENCODINGS.get(_enc_code, 'UNSUPPORTED ENCODING')
    return file_encoding


def decompress_eds(contents):
    """
    extract the eds and ico files from the uploaded file

    returns a dict of {file name: file contents}
    """
    GZ_MAGIC_BYTES = b'\x1f\x8b'

    # there is actually 2 files, the eds file and the icon
    # we need to split the file contents since gzip
    # only supports single files

    end_file1 = contents.find(GZ_MAGIC_BYTES, 2)
    file1, file2 = contents[:end_file1], contents[end_file1:]
    eds = gzip.decompress(file1)
    ico = gzip.decompress(file2)
    eds_name = file1[10:file1.find(b'\x00', 10)].decode()
    ico_name = file2[10:file2.find(b'\x00', 10)].decode()

    return {eds_name: eds, ico_name: ico}


def get_file_name(driver):
    """
    Get the filename of the eds file object
```

(continues on next page)

(continued from previous page)

```
    """
    attr = FileObjectInstanceAttributes.file_name
    resp = driver.generic_message(
        service=Services.get_attribute_single,
        class_code=ClassCode.file_object,
        attribute=attr.attr_id,
        instance=FileObjectInstances.eds_file_and_icon,
        route_path=True,
        unconnected_send=True,
        connected=False,
        data_type=attr.data_type
    )

    file_name = resp.value['FileName'][0] if resp else None
    return file_name


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

5.1.4 API Reference

CIPDriver API

class pycomm3.**CIPDriver**(*path*, **args*, ***kwargs*)

A base CIP driver for the SLCDriver and LogixDriver classes. Implements common CIP services like (un)registering sessions, forward open/close, generic messaging, etc.

__init__(*path*, **args*, ***kwargs*)

Parameters **path** (str) – CIP path to intended target

The path may contain 3 forms:

- IP Address Only (`10.20.30.100`) - Use for a ControlLogix PLC is in slot 0 or if connecting to a CompactLogix or Micro800 PLC.
- IP Address/Slot (`10.20.30.100/1`) - (ControlLogix) if PLC is not in slot 0
- CIP Routing Path (`1.2.3.4/backplane/2/enet/6.7.8.9/backplane/0`) - Use for more complex routing.

Note: Both the IP Address and IP Address/Slot options are shortcuts, they will be replaced with the CIP path automatically. The `enet` / `backplane` (or `bp`) segments are symbols for the CIP routing port numbers and will be replaced with the correct value.

---

property connected: bool

Read-Only Property to check whether or not a connection is open.

Return type bool

Returns True if a connection is open, False otherwise

property connection_size

CIP connection size, `4000` if using Extended Forward Open else `500`

classmethod list_identity(*path*)

Uses the ListIdentity service to identify the target

Return type `Optional[str]`

Returns device identity if reply contains valid response else None

classmethod discover()

Discovers available devices on the current network(s). Returns a list of the discovered devices Identity Object (as `dict`).

Return type `List[Dict[str, Any]]`

get_module_info(*slot*)

Get the Identity object for a given slot in the rack of the current connection

Return type `dict`

open()

Creates a new Ethernet/IP socket connection to target device and registers a CIP session.

Returns True if successful, False otherwise

close()

Closes the current connection and un-registers the session.

generic_message(*service*, *class_code*, *instance*, *attribute=b''*, *request_data=b''*, *data_type=None*, *name='generic'*, *connected=True*, *unconnected_send=False*, *route_path=True*, ***kwargs*)

Perform a generic CIP message. Similar to how MSG instructions work in Logix.

Parameters

- **service** (`Union[int, bytes]`) – service code for the request (single byte)
- **class_code** (`Union[int, bytes]`) – request object class ID
- **instance** (`Union[int, bytes]`) – ID for an instance of the class If set with 0, request class attributes.
- **attribute** (`Union[int, bytes]`) – (optional) attribute ID for the service/class/instance
- **request_data** (`Any`) – (optional) any additional data required for the request.
- **data_type** (`Union[Type[DataType], DataType, None]`) – a `DataType` class that will be used to decode the response, None to return just bytes
- **name** (`str`) – return `Tag.tag` value, arbitrary but can be used for tracking returned Tags
- **connected** (`bool`) – `True` if service required a CIP connection (forward open), `False` to use UCMM
- **unconnected_send** (`bool`) – (Unconnected Only) wrap service in an UnconnectedSend service
- **route_path** (`Union[bool, Sequence[CIPSegment], bytes, str]`) – (Unconnected Only) `True` to use current connection route to destination, `False` to ignore, Or provide a path string, list of segments to be encoded as a PADDED_EPATH, or an already encoded path.

Return type *Tag*

Returns a Tag with the result of the request. (Tag.value for writes will be the request_data)

LogixDriver API

class pycomm3.**LogixDriver**(*path*, **args*, *init_tags=True*, *init_program_tags=True*, ***kwargs*)

An Ethernet/IP Client driver for reading and writing tags in ControlLogix and CompactLogix PLCs.

__init__(*path*, **args*, *init_tags=True*, *init_program_tags=True*, ***kwargs*)

Parameters

- **path** (str) – CIP path to intended target

  The path may contain 3 forms:

  - IP Address Only (`10.20.30.100`) - Use for a ControlLogix PLC is in slot 0 or if connecting to a CompactLogix or Micro800 PLC.
  - IP Address/Slot (`10.20.30.100/1`) - (ControlLogix) if PLC is not in slot 0
  - CIP Routing Path (`1.2.3.4/backplane/2/enet/6.7.8.9/backplane/0`) - Use for more complex routing.

  Note: Both the IP Address and IP Address/Slot options are shortcuts, they will be replaced with the CIP path automatically. The `enet` / `backplane` (or `bp`) segments are symbols for the CIP routing port numbers and will be replaced with the correct value.

- **init_tags** (bool) – if True (default), uploads all controller-scoped tag definitions on connect
- **init_program_tags** (bool) – if False, bypasses uploading program-scoped tags. set to False if there are a lot of program tags and you aren't using any of them to decrease tag upload times.

Tip: Initialization of tags is required for the *read()* and *write()* to work. This is because they require information about the data type and structure of the tags inside the controller. If opening multiple connections to the same controller, you may disable tag initialization in all but the first connection and set `plc2._tags = plc1.tags` to prevent needing to upload the tag definitions multiple times.

open()

Creates a new Ethernet/IP socket connection to target device and registers a CIP session.

Returns True if successful, False otherwise

property revision_major: int

Returns the major revision for the PLC or 0 if not available

Return type int

property tags: dict

Read-only property to access all the tag definitions uploaded from the controller.

Return type dict

property tags_json

Read-only property to access all the tag definitions uploaded from the controller. Filters out any non-JSON serializable objects.

property data_types: dict

Read-only property for access to all data type definitions uploaded from the controller.

Return type dict

property connected: bool

Read-Only Property to check whether or not a connection is open.

Return type bool

Returns True if a connection is open, False otherwise

property info: dict

Property containing a dict of all the information collected about the connected PLC.

Fields:

- *vendor* - name of hardware vendor, e.g. 'Rockwell Automation/Allen-Bradley'
- *product_type* - typically 'Programmable Logic Controller'
- *product_code* - code identifying the product type
- *revision* - dict of {'major': <major rev (int)>, 'minor': <minor rev (int)>}
- *serial* - hex string of PLC serial number, e.g. 'FFFFFFFF'
- *product_name* - string value for PLC device type, e.g. '1756-L83E/B'
- *keyswitch* - string value representing the current keyswitch position, e.g. 'REMOTE RUN'
- *name* - string value of the current PLC program name, e.g. 'PLCA'

The following fields are added from calling *get_tag_list()*

- *programs* - dict of all Programs in the PLC and their routines, {program: {'routines': [routine, ...}...}
- *tasks* - dict of all Tasks in the PLC, {task: {'instance_id': ...}...}
- *modules* - dict of I/O modules in the PLC, {module: {'slots': {1: {'types': ['0,' 'I', 'C']}, ...}, 'types':[...]}...}

Return type dict

property name: Optional[str]

Return type Optional[str]

Returns name of PLC program

get_plc_name()

Requests the name of the program running in the PLC. Uses KB 23341 for implementation.

Return type str

Returns the controller program name

`get_plc_info`()

Reads basic information from the controller, returns it and stores it in the `info` property.

Return type `dict`

`get_plc_time`(*fmt='%A, %B %d, %Y %I:%M:%S%p'*)

Gets the current time of the PLC system clock. The `value` attribute will be a dict containing the time in 3 different forms, *datetime* is a Python datetime.datetime object, *microseconds* is the integer value epoch time, and *string* is the *datetime* formatted using `strftime` and the `fmt` parameter.

Parameters **fmt** (`str`) – format string for converting the time to a string

Return type *Tag*

Returns a Tag object with the current time

`set_plc_time`(*microseconds=None*)

Set the time of the PLC system clock.

Parameters **microseconds** (`Optional[int]`) – None to use client PC clock, else timestamp in microseconds to set the PLC clock to

Return type *Tag*

Returns Tag with status of request

`get_tag_list`(*program=None*, *cache=True*)

Reads the tag list from the controller and the definition for each tag. Definitions include tag name, tag type (atomic vs struct), data type (including nested definitions for structs), external access, dimensions defined (0-3) for arrays and their length, etc.

Note: For program scoped tags the tag['tag_name'] will be `'Program:{program}.{tag_name}'`. This is so the tag list can be fed directly into the read function.

Parameters

- **program** (`Optional[str]`) – scope to retrieve tag list, None for controller-only tags, `'*'` for all tags, else name of program
- **cache** (`bool`) – store the retrieved list in the *tags* property. Disable if you wish to get tags retrieved to not overwrite the currently cached definition. For instance if you're checking tags in a single program but currently reading controller-scoped tags.

Return type `List[dict]`

Returns a list containing dicts for each tag definition collected

`read`(**tags*)

Read the value of tag(s). Automatically will split tags into multiple requests by tracking the request and response size. Will use the multi-service request to group many tags into a single packet and also will automatically use fragmented read requests if the response size will not fit in a single packet. Supports arrays (specify element count in using curly braces (array{10}). Also supports full structure reading (when possible), return value will be a dict of {attribute name: value}.

Parameters **tags** (`str`) – one or many tags to read

Return type `Union[Tag, List[Tag]]`

Returns a single or list of `Tag` objects

write(**tags_values*)

Write to tag(s). Automatically will split tags into multiple requests by tracking the request and response size. Will use the multi-service request to group many tags into a single packet and also will automatically use fragmented read requests if the response size will not fit in a single packet. Supports arrays (specify element count in using curly braces (array{10}). Also supports full structure writing (when possible), value must be a sequence of values or a dict of {attribute: value} matching the exact structure of the destination tag.

Parameters **tags_values** (Union[str, int, float, bool, List[Union[int, float, bool, str]], Dict[str, Union[int, float, bool, str, List[Union[int, float, bool, str]], Dict[str, ForwardRef]]], Tuple[str, Union[int, float, bool, str, List[Union[int, float, bool, str]], Dict[str, Union[int, float, bool, str, List[Union[int, float, bool, str]], Dict[str, ForwardRef]]]]]]) – (tag, value) tuple or sequence of tag and value tuples [(tag, value), ...]

Return type Union[*Tag*, List[*Tag*]]

Returns a single or list of Tag objects.

get_tag_info(*tag_name*)

Returns the tag information for a tag collected during the tag list upload. Can be a base tag or an attribute.

Parameters **tag_name** (str) – name of tag to get info for

Return type Optional[dict]

Returns a dict of the tag's definition

SLCDriver API

class pycomm3.**SLCDriver**(*path*, **args*, ***kwargs*)

An Ethernet/IP Client driver for reading and writing of data files in SLC or MicroLogix PLCs.

read(**addresses*)

Reads data file addresses. To read multiple words add the word count to the address using curly braces, e.g. `N120:10{10}`.

Does not track request/response size like the CLXDriver.

Parameters **addresses** (str) – one or many data file addresses to read

Return type Union[*Tag*, List[*Tag*]]

Returns a single or list of Tag objects

write(**address_values*)

Write values to data file addresses. To write to multiple words in a file use curly braces in the address to indicate the number of words, then set the value to a list of values to write e.g. `('N120:10{10}', [1, 2, ...])`.

Does not track request/response size like the CLXDriver.

Parameters **address_values** (Tuple[str, Union[int, float, bool, List[Union[int, float, bool, str]]]]) – one or many 2-element tuples of (address, value)

Return type Union[*Tag*, List[*Tag*]]

Returns a single or list of Tag objects

Data Types

class pycomm3.cip.data_types.**DataType**(*name=None*)

Base class to represent a CIP data type. Instances of a type are only used when defining the members of a structure.

Each type class provides `encode` / `decode` class methods. If overriding them, they must catch any unhandled exception and raise a `DataError` from it. For `decode`, `BufferEmptyError` should be reraised immediately without modification. The buffer empty error is needed for decoding arrays of unknown length. Typically for custom types, overriding the private `_encode`/`_decode` methods are sufficient. The private methods do not need to do any exception handling if using the base public methods. For `_decode` use the private `_stream_read` method instead of `stream.read`, so that `BufferEmptyError` exceptions are raised appropriately.

classmethod encode(*value*)

Serializes a Python object `value` to `bytes`.

Note: Any subclass overriding this method must catch any exception and re-raise a `DataError`

Return type `bytes`

classmethod decode(*buffer*)

Deserializes a Python object from the `buffer` of `bytes`

Note: Any subclass overriding this method must catch any exception and re-raise as a `DataError`. Except `BufferEmptyErrors` they must be re-raised as such, array decoding relies on this.

Return type `Any`

class pycomm3.cip.data_types.**ElementaryDataType**(*name=None*)

Type that represents a single primitive value in CIP.

code: int = 0

CIP data type identifier

size: int = 0

size of type in bytes

class pycomm3.cip.data_types.**BOOL**(*name=None*)

A boolean value, decodes `0x00` and `False` and `True` otherwise. `True` encoded as `0xFF` and `False` as `0x00`

code: int = 193

0xC1

class pycomm3.cip.data_types.**SINT**(*name=None*)

Signed 8-bit integer

code: int = 194

0xC2

class pycomm3.cip.data_types.**INT**(*name=None*)

Signed 16-bit integer

code: int = 195
0xC3

class pycomm3.cip.data_types.DINT(*name=None*)
Signed 32-bit integer

code: int = 196
0xC4

class pycomm3.cip.data_types.LINT(*name=None*)
Signed 64-bit integer

code: int = 197
0xC5

class pycomm3.cip.data_types.USINT(*name=None*)
Unsigned 8-bit integer

code: int = 198
0xC6

class pycomm3.cip.data_types.UINT(*name=None*)
Unsigned 16-bit integer

code: int = 199
0xC7

class pycomm3.cip.data_types.UDINT(*name=None*)
Unsigned 32-bit integer

code: int = 200
0xC8

class pycomm3.cip.data_types.ULINT(*name=None*)
Unsigned 64-bit integer

code: int = 201
0xC9

class pycomm3.cip.data_types.REAL(*name=None*)
32-bit floating point

code: int = 202
0xCA

class pycomm3.cip.data_types.LREAL(*name=None*)
64-bit floating point

code: int = 203
0xCB

class pycomm3.cip.data_types.STIME(*name=None*)
Synchronous time information

code: int = 204
0xCC

class pycomm3.cip.data_types.DATE(*name=None*)
Date information

code: int = 205

0xCD

class pycomm3.cip.data_types.**TIME_OF_DAY**(*name=None*)

Time of day

code: int = 206

0xCE

class pycomm3.cip.data_types.**DATE_AND_TIME**(*name=None*)

Date and time of day

code: int = 207

0xCF

classmethod encode(*time*, *date*, **args*, ***kwargs*)

Serializes a Python object `value` to `bytes`.

Note: Any subclass overriding this method must catch any exception and re-raise a `DataError`

Return type `bytes`

class pycomm3.cip.data_types.**StringDataType**(*name=None*)

Base class for any string type

len_type = None

data type of the string length

encoding = 'iso-8859-1'

encoding of string data

class pycomm3.cip.data_types.**LOGIX_STRING**(*name=None*)

Character string, 1-byte per character, 4-byte length

len_type

alias of *pycomm3.cip.data_types.UDINT*

class pycomm3.cip.data_types.**STRING**(*name=None*)

Character string, 1-byte per character, 2-byte length

code: int = 208

0xD0

len_type

alias of *pycomm3.cip.data_types.UINT*

class pycomm3.cip.data_types.**BytesDataType**(*name=None*)

Base type for placeholder bytes.

pycomm3.cip.data_types.**n_bytes**(*count*, *name=''*)

Create an instance of a byte string of `count` length. Setting `count` to `-1` will consume the entire remaining buffer.

class pycomm3.cip.data_types.**BitArrayType**(*name=None*)

Array of bits (Python bools) for `host_type` integer value

class pycomm3.cip.data_types.**BYTE**(*name=None*)
bit string - 8-bits

code: int = 209
0xD1

host_type
alias of *pycomm3.cip.data_types.USINT*

class pycomm3.cip.data_types.**WORD**(*name=None*)
bit string - 16-bits

code: int = 210
0xD2

host_type
alias of *pycomm3.cip.data_types.UINT*

class pycomm3.cip.data_types.**DWORD**(*name=None*)
bit string - 32-bits

code: int = 211
0xD3

host_type
alias of *pycomm3.cip.data_types.UDINT*

class pycomm3.cip.data_types.**LWORD**(*name=None*)
bit string - 64-bits

code: int = 212
0xD4

host_type
alias of *pycomm3.cip.data_types.ULINT*

class pycomm3.cip.data_types.**STRING2**(*name=None*)
character string, 2-bytes per character

code: int = 213
0xD5

len_type
alias of *pycomm3.cip.data_types.UINT*

class pycomm3.cip.data_types.**FTIME**(*name=None*)
duration - high resolution

code: int = 214
0xD6

class pycomm3.cip.data_types.**LTIME**(*name=None*)
duration - long

code: int = 215
0xD7

class pycomm3.cip.data_types.**ITIME**(*name=None*)
duration - short

code: int = 216

0xD8

class pycomm3.cip.data_types.**STRINGN**(*name=None*)

character string, n-bytes per character

code: int = 217

0xD9

classmethod encode(*value*, *char_size=1*)

Serializes a Python object `value` to `bytes`.

Note: Any subclass overriding this method must catch any exception and re-raise a `DataError`

Return type `bytes`

class pycomm3.cip.data_types.**SHORT_STRING**(*name=None*)

character string, 1-byte per character, 1-byte length

code: int = 218

0xDA

len_type

alias of *pycomm3.cip.data_types.USINT*

class pycomm3.cip.data_types.**TIME**(*name=None*)

duration - milliseconds

code: int = 219

0xDB

class pycomm3.cip.data_types.**EPATH**(*name=None*)

CIP path segments

code: int = 220

0xDC

classmethod encode(*segments*, *length=False*, *pad_length=False*)

Serializes a Python object `value` to `bytes`.

Note: Any subclass overriding this method must catch any exception and re-raise a `DataError`

Return type `bytes`

classmethod decode(*buffer*)

Deserializes a Python object from the `buffer` of `bytes`

Note: Any subclass overriding this method must catch any exception and re-raise as a `DataError`. Except `BufferEmptyErrors` they must be re-raised as such, array decoding relies on this.

Return type `Sequence[CIPSegment]`

class pycomm3.cip.data_types.**PACKED_EPATH**(*name=None*)

class pycomm3.cip.data_types.**PADDED_EPATH**(*name=None*)

class pycomm3.cip.data_types.**ENGUNIT**(*name=None*)

engineering units

code: int = 221

0xDD

class pycomm3.cip.data_types.**STRINGI**(*name=None*)

international character string

code: int = 222

0xDE

classmethod encode(**strings*)

Encodes `strings` to bytes

Return type `bytes`

classmethod decode(*buffer*)

Deserializes a Python object from the `buffer` of `bytes`

Note: Any subclass overriding this method must catch any exception and re-raise as a `DataError`. Except `BufferEmptyErrors` they must be re-raised as such, array decoding relies on this.

Return type `Tuple[Sequence[str], Sequence[str], Sequence[int]]`

class pycomm3.cip.data_types.**DerivedDataType**(*name=None*)

Base type for types composed of *ElementaryDataType*

class pycomm3.cip.data_types.**ArrayType**(*name=None*)

Base type for an array

class pycomm3.cip.data_types.**StructType**(*name=None*)

Base type for a structure

class pycomm3.cip.data_types.**CIPSegment**(*name=None*)

Base type for a CIP path segment

Segment Type			Segment Format				
7	6	5	4	3	2	1	0

classmethod encode(*segment*, *padded=False*)

Encodes an instance of a `CIPSegment` to bytes

Return type `bytes`

classmethod decode(*buffer*)

Attention: Not Implemented

Return type `Any`

class pycomm3.cip.data_types.**PortSegment**(*port*, *link_address*, *name=''*)

Port segment of a CIP path.

Segment Type			Extended Link Addr	Port Identifier			
7	6	5	4	3	2	1	0

port_segments = {'backplane': 1, 'bp': 1, 'cnet': 2, 'dh485-a': 2, 'dh485-b': 3, 'dhrio-a': 2, 'dhrio-b': 3, 'dnet': 2, 'enet': 2}

available port names for use in a CIP path

class pycomm3.cip.data_types.**LogicalSegment**(*logical_value*, *logical_type*, **args*, ***kwargs*)

Logical segment of a CIP path

Segment Type			Logical Type			Logical Format	
7	6	5	4	3	2	1	0

logical_types = {'attribute_id': 16, 'class_id': 0, 'connection_point': 12, 'instance_id': 4, 'member_id': 8, 'service_id': 24, 'special': 20}

available logical types

class pycomm3.cip.data_types.**NetworkSegment**(*name=None*)

class pycomm3.cip.data_types.**SymbolicSegment**(*name=None*)

class pycomm3.cip.data_types.**DataSegment**(*data*, *name=''*)

Segment Type			Segment Sub-Type				
7	6	5	4	3	2	1	0

class pycomm3.cip.data_types.**ConstructedDataTypeSegment**(*name=None*)

class pycomm3.cip.data_types.**ElementaryDataTypeSegment**(*name=None*)

class pycomm3.cip.data_types.**DataTypes**

Lookup table/map of elementary data types. Reverse lookup is by CIP code for data type.

bool

alias of *pycomm3.cip.data_types.BOOL*

sint

alias of *pycomm3.cip.data_types.SINT*

int

alias of *pycomm3.cip.data_types.INT*

dint

alias of *pycomm3.cip.data_types.DINT*

lint

alias of *pycomm3.cip.data_types.LINT*

usint

alias of *pycomm3.cip.data_types.USINT*

uint
alias of *pycomm3.cip.data_types.UINT*

udint
alias of *pycomm3.cip.data_types.UDINT*

ulint
alias of *pycomm3.cip.data_types.ULINT*

real
alias of *pycomm3.cip.data_types.REAL*

lreal
alias of *pycomm3.cip.data_types.LREAL*

stime
alias of *pycomm3.cip.data_types.STIME*

date
alias of *pycomm3.cip.data_types.DATE*

time_of_day
alias of *pycomm3.cip.data_types.TIME_OF_DAY*

date_and_time
alias of *pycomm3.cip.data_types.DATE_AND_TIME*

logix_string
alias of *pycomm3.cip.data_types.LOGIX_STRING*

string
alias of *pycomm3.cip.data_types.STRING*

byte
alias of *pycomm3.cip.data_types.BYTE*

word
alias of *pycomm3.cip.data_types.WORD*

dword
alias of *pycomm3.cip.data_types.DWORD*

lword
alias of *pycomm3.cip.data_types.LWORD*

string2
alias of *pycomm3.cip.data_types.STRING2*

ftime
alias of *pycomm3.cip.data_types.FTIME*

ltime
alias of *pycomm3.cip.data_types.LTIME*

itime
alias of *pycomm3.cip.data_types.ITIME*

stringn
alias of *pycomm3.cip.data_types.STRINGN*

short_string
alias of *pycomm3.cip.data_types.SHORT_STRING*

time
alias of *pycomm3.cip.data_types.TIME*

padded_epath
alias of *pycomm3.cip.data_types.PADDED_EPATH*

packed_epath
alias of *pycomm3.cip.data_types.PACKED_EPATH*

engunit
alias of *pycomm3.cip.data_types.ENGUNIT*

stringi
alias of *pycomm3.cip.data_types.STRINGI*

Custom Types

class pycomm3.custom_types.**IPAddress**(*name=None*)

class pycomm3.custom_types.**ModuleIdentityObject**(*name=None*)

class pycomm3.custom_types.**ListIdentityObject**(*name=None*)

pycomm3.custom_types.**StructTemplateAttributes**
alias of pycomm3.cip.data_types.Struct.<locals>.Struct

pycomm3.custom_types.**FixedSizeString**(*size_*, *len_type_=UDINT*)
Creates a custom string tag type

class pycomm3.custom_types.**Revision**(*name=None*)

5.1.5 CIP Reference

Documented CIP service and class codes are available in enum-like classes that can be imported for use, mostly useful for generic messaging. The following classes may be imported directly from the pycomm3 package.

Ethernet/IP Encapsulation Commands

```
class EncapsulationCommands(EnumMap):
    nop = b"\x00\x00"
    list_targets = b"\x01\x00"
    list_services = b"\x04\x00"
    list_identity = b"\x63\x00"
    list_interfaces = b"\x64\x00"
    register_session = b"\x65\x00"
    unregister_session = b"\x66\x00"
    send_rr_data = b"\x6F\x00"
    send_unit_data = b"\x70\x00"
```

CIP Services and Class Codes

```
class Services(EnumMap):

    # Common CIP Services
    get_attributes_all = b"\x01"
    set_attributes_all = b"\x02"
    get_attribute_list = b"\x03"
    set_attribute_list = b"\x04"
    reset = b"\x05"
    start = b"\x06"
    stop = b"\x07"
    create = b"\x08"
    delete = b"\x09"
    multiple_service_request = b"\x0A"
    apply_attributes = b"\x0D"
    get_attribute_single = b"\x0E"
    set_attribute_single = b"\x10"
    find_next_object_instance = b"\x11"
    error_response = b"\x14"
    restore = b"\x15"
    save = b"\x16"
    nop = b"\x17"
    get_member = b"\x18"
    set_member = b"\x19"
    insert_member = b"\x1A"
    remove_member = b"\x1B"
    group_sync = b"\x1C"

    # Rockwell Custom Services
    read_tag = b"\x4C"
    read_tag_fragmented = b"\x52"
    write_tag = b"\x4D"
    write_tag_fragmented = b"\x53"
    read_modify_write = b"\x4E"
    get_instance_attribute_list = b"\x55"

    @classmethod
    def from_reply(cls, reply_service):
        """
        Get service from reply service code
        """
        val = cls.get(USINT.encode(USINT.decode(reply_service) - 128))
        return val
```

```
class ClassCode(EnumMap):
    identity_object = b"\x01"
    message_router = b"\x02"
    device_net = b"\x03"
    assembly = b"\x04"
    connection = b"\x05"
    connection_manager = b"\x06"
    register = b"\x07"
```

(continues on next page)

(continued from previous page)

```
    discrete_input = b"\x08"
    discrete_output = b"\x09"
    analog_input = b"\x0A"
    analog_output = b"\x0B"
    presence_sensing = b"\x0E"
    parameter = b"\x0F"

    parameter_group = b"\x10"
    group = b"\x12"
    discrete_input_group = b"\x1D"
    discrete_output_group = b"\x1E"
    discrete_group = b"\x1F"

    analog_input_group = b"\x20"
    analog_output_group = b"\x21"
    analog_group = b"\x22"
    position_sensor = b"\x23"
    position_controller_supervisor = b"\x24"
    position_controller = b"\x25"
    block_sequencer = b"\x26"
    command_block = b"\x27"
    motor_data = b"\x28"
    control_supervisor = b"\x29"
    ac_dc_drive = b"\x2A"
    acknowledge_handler = b"\x2B"
    overload = b"\x2C"
    softstart = b"\x2D"
    selection = b"\x2E"

    s_device_supervisor = b"\x30"
    s_analog_sensor = b"\x31"
    s_analog_actuator = b"\x32"
    s_single_stage_controller = b"\x33"
    s_gas_calibration = b"\x34"
    trip_point = b"\x35"
    file_object = b"\x37"
    s_partial_pressure = b"\x38"
    safety_supervisor = b"\x39"
    safety_validator = b"\x3A"
    safety_discrete_output_point = b"\x3B"
    safety_discrete_output_group = b"\x3C"
    safety_discrete_input_point = b"\x3D"
    safety_discrete_input_group = b"\x3E"
    safety_dual_channel_output = b"\x3F"

    s_sensor_calibration = b"\x40"
    event_log = b"\x41"
    motion_axis = b"\x42"
    time_sync = b"\x43"
    modbus = b"\x44"
    modbus_serial_link = b"\x46"
```

(continues on next page)

(continued from previous page)

```
    symbol_object = b"\x6b"
    template_object = b"\x6c"
    program_name = b"\x64"  # Rockwell KB# 23341

    wall_clock_time = b"\x8b"  # Micro800 CIP client messaging quick start

    controlnet = b"\xF0"
    controlnet_keeper = b"\xF1"
    controlnet_scheduling = b"\xF2"
    connection_configuration = b"\xF3"
    port = b"\xF4"
    tcp_ip_interface = b"\xF5"
    ethernet_link = b"\xF6"
    componet_link = b"\xF7"
    componet_repeater = b"\xF8"
```

```
class CommonClassAttributes(EnumMap):
    revision = Attribute(1, UINT("revision"))
    max_instance = Attribute(2, UINT("max_instance"))
    number_of_instances = Attribute(3, UINT("number_of_instances"))
    optional_attribute_list = Attribute(4, UINT[UINT])
    optional_service_list = Attribute(5, UINT[UINT])
    max_id_number_class_attributes = Attribute(6, UINT("max_id_class_attrs"))
    max_id_number_instance_attributes = Attribute(7, UINT("max_id_instance_attrs"))
```

Identity Object

```
class IdentityObjectInstanceAttributes(EnumMap):
    vendor_id = Attribute(1, UINT("vendor_id"))
    device_type = Attribute(2, UINT("device_type"))
    product_code = Attribute(3, UINT("product_code"))
    revision = Attribute(4, Struct(USINT("major"), USINT("minor")))
    status = Attribute(5, WORD("status"))
    serial_number = Attribute(6, UDINT("serial_number"))
    product_name = Attribute(7, SHORT_STRING("product_name"))
```

Connection Manager Object

```
class ConnectionManagerServices(EnumMap):
    forward_close = b"\x4E"
    unconnected_send = b"\x52"
    forward_open = b"\x54"
    get_connection_data = b"\x56"
    search_connection_data = b"\x57"
    get_connection_owner = b"\x5A"
    large_forward_open = b"\x5B"
```

```
class ConnectionManagerInstances(EnumMap):
    open_request = b"\x01"
    open_format_rejected = b"\x02"
    open_resource_rejected = b"\x03"
    open_other_rejected = b"\x04"
    close_request = b"\x05"
    close_format_request = b"\x06"
    close_other_request = b"\x07"
    connection_timeout = b"\x08"
```

File Object

```
class FileObjectServices(EnumMap):
    initiate_upload = b"\x4B"
    initiate_download = b"\x4C"
    initiate_partial_read = b"\x4D"
    initiate_partial_write = b"\x4E"
    upload_transfer = b"\x4F"
    download_transfer = b"\x50"
    clear_file = b"\x51"
```

```
class FileObjectClassAttributes(EnumMap):
    directory = Attribute(
        32,
        Struct(UINT("instance_number"), STRINGI("instance_name"), STRINGI("file_name")),
    )  # array of struct, len in attr 3
```

```
class FileObjectInstanceAttributes(EnumMap):
    state = Attribute(1, USINT("state"))
    instance_name = Attribute(2, STRINGI("instance_name"))
    instance_format_version = Attribute(3, UINT("instance_format_version"))
    file_name = Attribute(4, STRINGI("file_name"))
    file_revision = Attribute(5, Struct(USINT("major"), USINT("minor")))
    file_size = Attribute(6, UDINT("file_size"))
    file_checksum = Attribute(7, INT("file_checksum"))
    invocation_method = Attribute(8, USINT("invocation_method"))
    file_save_params = Attribute(9, BYTE("file_save_params"))
    file_type = Attribute(10, USINT("file_type"))
    file_encoding_format = Attribute(11, USINT("file_encoding_format"))
```

```
class FileObjectInstances(EnumMap):
    eds_file_and_icon = 0xC8
    related_eds_files_and_icons = 0xC9
```

5.1.6 Contributing

Contributing to pycomm3

This document aims to provide a brief guide on how to contribute to `pycomm3`.

Who can contribute?

Anyone! Contributions from any user are welcome. Contributions aren't limited to changing code. Filing bug reports, asking questions, adding examples or documentation are all ways to contribute. New users may find it helpful to start with improving documentation, type hinting, or tests.

Asking a question

Questions can be submitted as either an issue or a discussion post. A general question not directly related to the code or one that may be beneficial to other users would be most appropriate in the discussions area. One that is about a specific feature or could turn into a feature request or bug report would be more appropriate as an issue. If submitting a question as an issue, please use the *question* template.

Submitting an Issue

No code is perfect, `pycomm3` is no different and user submitted issues aid in improving the quality of this library. Before submitting an issue, check to see if someone has already submitted one before so we can avoid duplicate issues.

Bug Reports

To submit a bug report, please create an issue using the *Bug Report* template. Please include as much information as possible relating to the bug. The more detailed the bug report, the easier and faster it will be to resolve. Some details to include:

- The version of `pycomm3` (easily found with the `pip show pycomm3` command)
- Model/Firmware/etc if the issue is related to a specific device or firmware version
- Logs (see the documentation to configure)
  - A helper method is provided to simplify logging configs, including logging to a file
  - Using the `LOG_VERBOSE` level is the most helpful
- Sample code that will reproduce the bug

Feature Requests

For feature requests or enhancements, please create an issue using the *Feature Request* template. New features could be things like:

- A missing feature from a similar library
  - e.g. Library *X* has a feature *Y*, would it be possible to add *Y* functionality to `pycomm3`?
- Change or modification to the API
  - If it's a breaking change be sure to include why the new functionality is better than the current

- Enhancing a current feature
- Removing an old/broken/unsupported feature

Submitting Changes

Submitting code or documentation changes is another way to contribute. All contributions should be made in the form of a pull request. You should fork this repository and clone it to your machine. All work is done in the `develop` branch first before merging to `master`. All pull requests should target the `develop` branch. This is because some of the tests are specific to a demo PLC. Once changes are completed in `develop` and all tests are passing, `develop` will be merged into `master` and a new release created and available on PyPI.

Some requirements for code changes to be accepted include:

- code should be *pythonic* and follow PEP8, PEP20, and other Python best-practices or common conventions
- public methods should have docstrings which will be included in the documentation
- comments and docstrings should explain *why* and *how* the code works, not merely *what* it is doing
- type hinting should be used as much as possible, all public methods need to have hints
- new functionality should have tests
- run the *user* tests and verify there are no issues
- avoid 3rd party dependencies, code should only require the Python standard library
- avoid breaking changes, unless adequately justified
- do not update the library version

Some suggested contributions include:

- type hinting
  - all public methods are type hinted, but many internal methods are missing them
- tests
  - new tests are always welcome, particularly offline tests or any methods missing tests
- examples
  - example scripts showing how to use this library or any of it's features
  - you may include just the example script if you're not comfortable with also updating the docs to include it

New Feature or an Example?

It can be tough to decide whether functionality should be added to the library or shown as an example. New features should apply to generally to almost all devices for a driver or implement new functionality that cannot be done externally. If submitting an example, please include name/username/email/etc in a comment/docstring if you wish to be credited.

Here are a couple examples of changes and why they were added either as a feature or example:

[Feature] Add support for writing structures with a dictionary for the value:

- Cannot be done without modifying internal methods
- New functionality not yet implemented
- Improves user experience
  - user can read a struct, change one value, and write it back without changing the data structure

[Example] Add support for reading/writing Powerflex drive parameters:

- Implemented using the `generic_message` method
- Does not apply to a wide arrange of device types
- Not a PLC, so doesn't fit in the Logix or SLC drivers
- Too specific for the CIPDriver, but not enough to create a new driver

Some questions to ask yourself when deciding between a feature or an example:

- Is this new functionality or a new use of current functionality? *Former may be a feature, latter could be an example*
- Can this be done using already available features? *Yes, then maybe an example*
- Does this apply to a wide arrange of devices? *Yes, then maybe a feature*
- Will this require internal changes to existing functionality? *Yes, then maybe a feature*
- Is this useful? *Either should be useful*

5.1.7 Release History

1.2.7

LogixDriver

- fixed issue with program-scoped tags in *get_tag_info* #216

1.2.6

LogixDriver

- fixed issue handling BOOLs in some predefined types #197

1.2.5

LogixDriver

- fixed issue parsing struct definitions for predefined types for v32+ #186

1.2.4

LogixDriver

- fixed issue for BOOL members inside structures that was introduced as part of 1.2.3 #182

1.2.3

LogixDriver

- fixed issue with bit-level access to integers inside nested structs #170

1.2.2

CIPDriver

- added support for string CIP paths in *generic_message* for *route_path*
- fixed bug where errors during discovery prevent any results from being returned
- fixed issue where `get_module_info` would always use first hop in path instead of the last

LogixDriver

- fixed issue with multi-request message size tracking being off by 2 bytes
- fixed issue with AOI structure handling with > 8 BOOL members being mapped to types larger than a USINT (SISAutomationIMA)

1.2.1

- added ability to configure custom logger via the *configure_default_logger* function

1.2.0

- fixed issue with logging configuration
- formatted project with black
- misc. documentation updates

LogixDriver

- fixed issue with writing a tag multiple times failing after the first write
- added *tags_json* property

SLCDriver

- fixed issue with parsing IO addresses
- improved address parsing speed by pre-compiling regex

1.1.1

LogixDriver

- fixed read/write errors by preventing program-scoped tags from using instance ids in the request

1.1.0

LogixDriver

- fixed bugs in handling of built-in types (TIMER, CONTROL, etc)
- fixed bugs in structure tag handling when padding exists between attributes
- **changed the meaning of the element count for BOOL arrays**
  - Previously, the `{#}` referred to the underlying `DWORD` elements of the `BOOL` array. A `BOOL[64]` array is actually a *DWORD[2]* array, so `array{1}` translated to BOOL elements 0-31 or the first `DWORD` element. Now, the `{#}` refers to the number of `BOOL` elements. So `array{1}` is only a single `BOOL` element and `array{32}` would be the 0-31 `BOOL` elements.
  - Refer to the documentation for limitations on writing.

1.0.1

- Fixed incorrect/no error in response Tag for some failed requests in a multi-request
- Minor refactor to status and extended status parsing

1.0.0

- **New type system to replace the `Pack` and `Unpack` helper classes**
  - New types represent any CIP type or object and allow encoding and decoding of values
  - Allows users to create their own custom types
  - **[Breaking]** `generic_message` replaced the `data_format` argument with `data_type`, see documentation for details.
- Added a new `discover()` method for finding Ethernet/IP devices on the local network
- **Added a `configure_default_logger` method for simple logging setup**
  - Packet contents are now logged using a custom `VERBOSE` level
- Internal package structure changed.
- Lots of refactoring, decoupling, etc
- Increased test coverage
- **New and improved documentation**
  - Still a work-in-progress

Logix Driver

- **Upload of program-scoped tags is now enabled by default**
  - Use `init_program_tags=False` in initializer for to upload controller-scoped only tags
- **Removed the `init_info` and `micro800` init args and the `use_instance_ids` property**
  - These have all been automatic for awhile now, but were left for backwards compatibility
  - If you need to customize this behavior, override the `_initialize_driver` method

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O