1 Introduction

This tutorial shows users how to receive Ciholas Data Protocol (CDP) data and decode it using Python 3.

First, begin by creating a UDP listen socket and receiving raw data. Then, parse out the received data using the CDP packet structure as a guide. Finally, the information inside the Position CDP data item is decoded.

Users already familiar with these steps can go directly to using cdp-py to learn how to use the Python 3 module to decode CDP data.

> Need another language other than python? Check out our github page for additional languages.

2 Getting Started

2.1 CDP Settings

The CDP provides a method of communication between devices and services. CDP data is transmitted over Ethernet as User Datagram Protocol (UDP) packets.

CDP packets are transmitted through CDP Streams. CDP Streams are identified by the IP address, port, and Ethernet interface through which the packets are sent. Streams are allowed to be both multicast and unicast.

For this tutorial, use the settings of *User* CDP Stream configured by the CUWB Manager. To find the CDP settings, follow these steps:

1. Open the CUWB manager.
2. Start the CUWBNet.
3. Click on the *config* button next to the running CUWBNet.
4. Click on the *configuration* tab at the top.
5. In the *Ethernet Settings* section, make note of the IP address, port, and interface values for the *User* CDP stream.
6. In this tutorial, the following CDP settings will be used:
    - *IP*: 239.255.76.67
    - *Port*: 7667
    - *Interface*: 10.99.51.0

3 Create a CDP Listen Socket

To get started, import the `socket` module, initialize the CDP settings, and create a UDP socket for listening to CDP packets.

```
import socket

# Remember to use your CDP settings
ip = '239.255.76.67'
port = 7667
interface = '10.99.51.0'

listen_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

Then, bind the socket to the IP address and port with the CDP settings.

```
listen_socket.bind((ip, port))
```

Finally, add the socket to the multicast group using the interface IP address from the CDP settings.

```
listen_socket.setsockopt(socket.SOL_IP, socket.IP_ADD_MEMBERSHIP, socket.inet_aton(ip)+socket.inet_aton(i
nterface))
```

The complete code should look like this:

```
import socket

# Remember to use your CDP settings
ip = '239.255.76.67'
port = 7667
interface = '10.99.51.0'

# Create UDP socket for listening to CDP packets
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind to the CUWB Network address
listen_socket.bind((ip, port))

# Add the socket to the multicast group on our specific interface
listen_socket.setsockopt(socket.SOL_IP, socket.IP_ADD_MEMBERSHIP, socket.inet_aton(ip)+socket.inet_aton(i
nterface))
```

4 Receiving Data

Now that the listen socket is setup, proceed to receive data from the socket using the `recvfrom` method. The return value of this method is a tuple `(bytes, address)`, where *bytes* is a sequence of bytes representing the received CDP data and *address* is the address of the socket sending the data. Once the data has been received, print it out to the console.

```
while True:
  data, address = listen_socket.recvfrom(65536)  # 2^16 is the maximum size of a CDP packet
  print(data)
```

Now, put it all together!

```
import socket

# Remember to use your CDP settings
ip = '239.255.76.67'
port = 7667
interface = '10.99.51.0'

# Create UDP socket for listening to CDP packets
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind to the CUWB Network address
listen_socket.bind((ip, port))

# Add the socket to the multicast group on our specific interface
listen_socket.setsockopt(socket.SOL_IP, socket.IP_ADD_MEMBERSHIP, socket.inet_aton(ip)+socket.inet_aton(i
nterface))

while True:
  # Receiving CDP data on the socket
  data, address = listen_socket.recvfrom(65536)  # 2^16 is the maximum size of a CDP packet
  print(data)
```

Example output:

```
b"LC02\x00\x00\x04sCDP0002\x00\x00\x00\x00\x00'\x01\x18\x00\x00\x02\x04\x01\xc7\x02\x04\x01\x00\x00IO\xac
\x15\xb3E\x00\x00\xd69\x00\x00\x00\x00"
b"LC02\x00\x00\x04tCDP0002\x00\x00\x00\x00\x00'\x01\x18\x00\xc7\x02\x04\x01\x00\x02\x04\x01\x00\x00\x7f\t
\xb0\x1eTE\x00\x00\xf29\x00\x00\x00\x00"
```

Run the script and make sure that raw data is being printed out to the console. If no data is printed, verify that the CUWBNet is running and that the correct CDP settings are being used.

5 Decoding the Header

A CDP packet is made up of a CDP Packet Header followed by one or more CDP data items. It is important to note that all CDP numerical fields are transmitted using little-endian format.

Start by decoding the header!

The CDP Packet Header has a total size of 20 bytes and can be broken down into:

- *Mark*: The magic word '0x3230434C' in little-endian (4 bytes).
- *Sequence*: The sequence number of the CDP packet (4 bytes).
- *String*: The ASCII string 'CDP0002\0' (8 bytes).
- *Serial Number*: Unique identifier of the reporting device (4 bytes).

To parse out the raw CDP data, use the `struct` module. This module allows users to convert binary data into Python types.

Begin by adding the import statement:

```
import socket
import struct
```

Use the `unpack` function, included in the `struct` module, to parse out the raw data. This function takes in a *format* string argument, which specifies the byte order, size in bytes, and type of data, and a *buffer* argument.

First, decode the *mark*. Since all CDP fields use little endian format, use `'<'` to indicate little endian byte order. As mentioned, the *mark* has a size of 4 bytes and is an integer (unsigned int in C), which corresponds to the format character `'I'`. For more details about format characters, see Format Characters.

```
while True:
  # Receiving CDP data on the socket
  data, address = listen_socket.recvfrom(65536)  # 2^16 is the maximum size of a CDP packet

  # Decode the CDP Packet Header

  # '<I' corresponds to a 4-byte integer (unsigned int in C) in little endian
  mark, = struct.unpack('<I', data[:4])
  data = data[4:]
  print(mark)
```

The result of `unpack` is always a tuple even if it contains exactly one item. Add a `,` after the *mark* variable to assign the value to the variable and not the tuple. Then, print out the *mark*, and strip out the data just unpacked from the raw CDP *data* variable.

Run the code. The result should be the magic word `0x3230434C` in hex (`842023756` in decimal) printed out to the console.

Now, decode the *sequence*, *string*, and *serial number* fields:

```
while True:
  # Receiving CDP data on the socket
  data, address = listen_socket.recvfrom(65536)  # 2^16 is the maximum size of a CDP packet

  # Decode the CDP Packet Header

  # '<I' corresponds to a 4-byte integer (unsigned int in C) in little endian
  mark, = struct.unpack('<I', data[:4])
  data = data[4:]

  sequence, = struct.unpack('<I', data[:4])
  data = data[4:]

  # '<8s' corresponds to an 8-byte bytes string (char[] in C) in little endian
  string, = struct.unpack('<8s', data[:8])
  data = data[8:]

  serial_number, = struct.unpack('<I', data[:4])
  data = data[4:]

  # Print out all the fields in the CDP packet header
  print('CDP Header', mark, sequence, string, serial_number)
```

Example output:

```
CDP Header 842023756 2160197632 b'CDP0002\x00' 16778507
CDP Header 842023756 2176974848 b'CDP0002\x00' 17105521
CDP Header 842023756 2193752064 b'CDP0002\x00' 17105521
```

Run the code again. The results should be the 4 fields included in the CDP Packet Header printed to the console. Congratulations!

6 Looping through CDP Data Items

Every CDP data item starts with a 4-byte CDP Data Header followed by 0 to 65535 bytes of data. The CDP Data Header specifies the *type* of the CDP data item (2 bytes) and the *size* of the data associated with it (2 bytes).

Since a CDP packet can contain multiple CDP data items, place the code inside a loop underneath the print statement:

```
# Print out all the fields in the CDP packet header
print('CDP Header', mark, sequence, string, serial_number)

# Decode the CDP data items

# Loop until we run out of data to decode
while data:

  # '<H' corresponds to a 2-byte integer (unsigned short in C) in little endian
  di_type, = struct.unpack('<H', data[:2])
  data = data[2:]

  di_size, = struct.unpack('<H', data[:2])
  data = data[2:]

  # Data associated with the CDP data item
  di_data = data[:di_size]
  data = data[di_size:]

  # Print out the type, size and data of the CDP data item
  print('CDP Data Item', di_type, di_size, di_data)
```

Example output:

```
CDP Data Item 301 25 b'\x00\x00\x00(\xee\x08\x00\x00\x83\xa8r\xff\xd0F)
\xe5d\xe7\xe7\xc5\xea\x8cT\xff\x01'
CDP Data Item 297 21 b'\x00\x00\x00;\xee\x08\x00\x00\x00\x00i\x00\x00\x00\xac\x01\x00\x00\x9e\xbe\x02'
CDP Data Item 298 22
b'\x00\x00\x00;\xee\x08\x00\x00\x00\x00\xd8\xff\x00\x00\xfe\xff\x00\x00\x0c\x00\xd0\x07'
```

7 Decoding the Position Data Item

So far the code receives CDP data and decodes the CDP Packet Header and the CDP Data Header of all the CDP data items. Now, take a closer look to what information a CDP data item can include.

Use the Position V3 data item as an example. This CDP data item reports the position in 3D of a reporting device.

The *type* of Position V3 is `0x0135` and it has a total size of 30 bytes. The CDP data item can be broken down into:

- *Serial Number*: The serial number of the reporting device (4 bytes).
- *Network Time*: The timestamp when the sensor recorded the data (8 bytes).
- *Coordinates*: The coordinates from the origin (12 bytes). This field can be further broken down into the *x-coordinate* (4 bytes), *y-coordinate* (4 bytes), and *z-coordinate* (4 bytes).
- *Quality*: The quality of the computed position (2 bytes).
- *Anchor Count*: The number of anchors involved in the calculation of this position (1 byte).
- *Flags*: Reserved for future use (1 byte).
- *Smoothing*: The effective smoothing factor (2 bytes).

Assuming PositionV3 data is being received, proceed to decode and print it:

```
# Check if we got Position V3 data
if di_type == 0x0135:

  serial_number, = struct.unpack('<I', di_data[:4])
  di_data = di_data[4:]

  # '<Q' corresponds to a 8-byte integer (unsigned long long in C) in little endian
  network_time, = struct.unpack('<Q', di_data[:8])
  di_data = di_data[8:]

  # '<iii' corresponds to 3 4-byte integers (int in C) in little endian
  x,y,z = struct.unpack('<iii', di_data[:12])
  di_data = di_data[12:]

  quality, = struct.unpack('<H', di_data[:2])
  di_data = di_data[2:]

  # '<B' corresponds to a 1-byte integer (unsigned char in C) in little endian
  anchor_count, = struct.unpack('<B', di_data[:1])
  di_data = di_data[1:]

  flags, = struct.unpack('<B', di_data[:1])
  di_data = di_data[1:]

  smoothing, = struct.unpack('<H', di_data[:2])
  di_data = di_data[2:]

  # Print out all the fields in the Position V3 data item
  print('Position V3', serial_number, network_time, x, y, z, quality, anchor_count, flags, smoothing)
```

Example output:

```
Position V3 17105038 6193615074405 -9001 -2138 738 9264 17 0 0
Position V3 17105038 6200004819039 -9016 -2109 744 9311 17 0 0
Position V3 17105038 6206394578946 -9014 -2112 744 9311 17 0 0
```

This is how the complete code looks after some clean up:

```
while True:
  # Receiving CDP data on the socket
  data, address = listen_socket.recvfrom(65536)  # 2^16 is the maximum size of a CDP packet

  # Decode the CDP Packet Header

  # '<I' corresponds to a 4-byte integer (unsigned int in C) in little endian
  mark, = struct.unpack('<I', data[:4])
  data = data[4:]

  sequence, = struct.unpack('<I', data[:4])
  data = data[4:]

  # '<8s' corresponds to an 8-byte bytes string (char[] in C) in little endian
  string, = struct.unpack('<8s', data[:8])
  data = data[8:]

  serial_number, = struct.unpack('<I', data[:4])
  data = data[4:]

  # Decode the CDP data items

  # Loop until we run out of data to parse out
  while data:

    # '<H' corresponds to a 2-byte integer (unsigned short in C) in little endian
    di_type, = struct.unpack('<H', data[:2])
    data = data[2:]

    di_size, = struct.unpack('<H', data[:2])
    data = data[2:]

    # Data associated with the CDP data item
    di_data = data[:di_size]
    data = data[di_size:]

    # Check if we got Position V3 data
    if di_type == 0x0135:

      serial_number, = struct.unpack('<I', di_data[:4])
      di_data = di_data[4:]

      # '<Q' corresponds to a 8-byte integer (unsigned long long in C) in little endian
      network_time, = struct.unpack('<Q', di_data[:8])
      di_data = di_data[8:]

      # '<iii' corresponds to 3 4-byte integers (int in C) in little endian
      x,y,z = struct.unpack('<iii', di_data[:12])
      di_data = di_data[12:]

      quality, = struct.unpack('<H', di_data[:2])
      di_data = di_data[2:]

      # '<B' corresponds to a 1-byte integer (unsigned char in C) in little endian
      anchor_count, = struct.unpack('<B', di_data[:1])
      di_data = di_data[1:]

      flags, = struct.unpack('<B', di_data[:1])
      di_data = di_data[1:]

      smoothing, = struct.unpack('<H', di_data[:2])
      di_data = di_data[2:]


      # Print out all the fields in the Position V3 data item
      print('Position V3', serial_number, network_time, x, y, z, quality, anchor_count, flags, smoothing)
```

8 Using CDP-Python

The `cdp-py` package contains a set of structural definitions of the CDP data items. It includes useful methods to decode and print all the existing CDP data items.

8.1 Installing

Python 3 is required to run `cdp-py`. The package can be installed using pip. To install the latest version use:

```
$ pip install cdp-py
```

For more details, see the cdp-py project at PyPI.

8.2 Importing

Import the `cdp` module by adding the following import statement:

```
import cdp
```

8.3 Decoding CDP Packets

Decoding a CDP packet is as simple as:

```
cdp_packet = cdp.CDP(data)
```

Assuming a UDP socket with the correct CDP settings has been created, the code should look like this:

```
import socket
import cdp

# Remember to use your CDP settings
ip = '239.255.76.67'
port = 7667
interface = '10.99.51.0'

# Create UDP socket for listening to CDP packets
listen_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Bind to the CUWB Network address
listen_socket.bind((ip, port))

# Add the socket to the multicast group on our specific interface
listen_socket.setsockopt(socket.SOL_IP, socket.IP_ADD_MEMBERSHIP, socket.inet_aton(ip)+socket.inet_aton(i
nterface))

while True:
  # Receiving CDP data on the socket
  data, address = listen_socket.recvfrom(65536)  # 2^16 is the maximum size of a CDP packet
  cdp_packet = cdp.CDP(data)
```

8.4 Decoding the Header

The *sequence* and *serial number* of the transmitting device included in the CDP Packet Header can be accessed as follows:

```
print(cdp_packet.sequence)
print(cdp_packet.serial_number)
```

Example output:

```
1141309440  # Sequence
01:00:050B  # Serial Number
```

The integer representation of the *serial number* can be printed using:

```
print(cdp_packet.serial_number.as_int)
```

Example output:

```
16778507  # Serial Number '01:00:050B' as an integer
```

8.5 Looping through CDP Data Items

Print out all of the CDP data items in a CDP packet by doing:

```
for item in cdp_packet.data_items:
  print(item)
```

The output should look something like this:

```
01:04:02AB, 0x010E, 01:04:01CC, 0, 0, -8837, -8772, 856
01:04:02AB, 0x0100, -2726, 318, 4276, 0, 0, 211, 48852817
```

The standard printable representation of a CDP data item includes the *serial number* of the transmitting device, followed by the *type* of the CDP data item and the values of the fields associated with that data item. All separated by commas.

8.6 Decoding the Position Data Item

To check if a particular *type* is included in the CDP packet, use the `cdp_packet.data_items_by_type` dictionary. The dictionary is indexed by the *types* of the CDP data items in the packet. The *values* are lists of data items because a CDP packet may contain multiple data items of the same type.

The Position V3 (0x0135) data item is decoded in this example:

```
# Check if we got Position V3 (0x0135) data and print all data items of this type
for pos_item in cdp_packet.data_items_by_type.get(0x0135, []):
  print(pos_item)
```

The output may look like:

```
FE:72:7BE8, 0x0135, 01:05:008E, 10168045792012, -8951, -2100, 812, 9212, 16, 0, 0
```

The first two values are the reporting device's *serial number* and Position V3 *type*. They are followed by the *serial number, _network time*, *xyz coordinates*, *quality*, *anchor count*, *flags*, and *smoothing* values that make up the Position V3 data item.

Additionally, users can access the attributes of the Position V3 data item by doing:

```
print('({}, {}, {})'.format(pos_item.x, pos_item.y, pos_item.z))
```

The above code prints the position coordinates from the origin as a point in the form `(x, y, z)`:

```
(-8951, -2100, 812)
```

9 Revision

Version	Date	Change Description
5.0.1	2025-10-31	Introduction update, Fixing Links
5.0.0	2025-09-15	Initial Preliminary Release

10 APPENDIX A : Software License Attributions

1 PyYAML

PyYAML
The MIT License (MIT)

- The MIT License (MIT)
  Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

  The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

2 setuptools

setuptools
The MIT License (MIT)

- The MIT License (MIT)
  Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

  The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

  THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3 twine

twine
Apache-2.0

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.