# Building Network Distributed Applications With The *XideKit* API

Ever since releasing ***Xideco***, I am finding more and more that my physical computing projects are gravitating towards using a network distributed application approach. Not only does this allow me to reuse existing modules in the context of a new application, but in many instances I am also able to add new or expanded functionality to a "live" application without effecting existing modules.

Both ZeroMQ and MessagePack are at the heart of these distributed network application designs. To help simplify creating a network distributed application, I created ***XideKit*** **,** a base API class, that encapsulates both ZeroMQ and MessagePack operations. This paper will discuss the details and use of the XideKit API. Code examples and videos are provided to aid in the discussion.

## Why ZeroMQ and MessagePack?

***Wide Array of Support***

Both ZeroMQ and MessagePack are available for a wide array of languages and operating systems, allowing for maximum flexibility when designing and deploying applications. Even though XideKit is Python3 centric, applications written in other languages may be added to a Xidekit network application, as long as they adhere to the application's messaging protocol. The messaging protocol is specified by the application designer and is *not* specified by XideKit.

***Provides A Higher Level View of Both System and Data***

By using a high level, transparent messaging protocol, seemingly disparate components can communicate and coexist with each other in a consistent manner. For example, to change the state of a GPIO output pin, a common protocol message is used, independent of board type. So if a new board type is added, the board support module will translate this message to its specific GPIO control library, without affecting the other boards or control driver module (e.g. a GUI).

Components written in other computer languages can be plugged into the network as long as they implement the control protocol. An example of this is Xideco's ability to use a node.js component to monitor protocol messages being generated by Python modules.

***Allows For Maximum Design Flexibility***

Applications may reside on a single computer or across multiple computers without any modifications to the module's source code.

Components can be added or removed to a "live" system without affecting other components within the system. Components can be "hot-plugged".

Each component runs in its own process. Surprisingly enough, there is little overhead in doing so, with the added benefit of memory safety.

# Some Background Information

***Publisher/Subscriber Pattern***

XideKit encapsulates the ZeroMQ Pub/Sub (publisher/subscriber) design pattern. When an application wishes to share information with other components on the network, it creates a MessagePack message payload. A message topic is appended to the message before transmission.

A we shall soon see, the message payload is actually a Python dictionary. The name value pairs define the messaging protocol. An example message to set a GPIO pin might be:

{'command': 'set_digital_pin', 'pin_number': '7', 'value': 1}

A message topic consists of a simple string. For the example above, if the topic is set to 'arduino', any module that has subscribed to receive messages with the topic of 'arduino' will receive the message. It is up to the subscriber to then interpret and process the received messages. All other messages are filtered out by ZeroMQ.

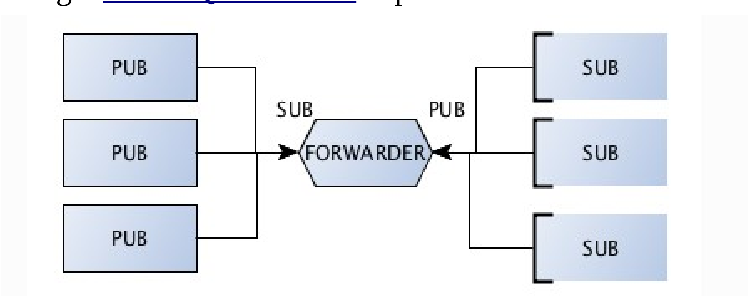
A subscriber has the option to subscribe to one, several or all message topics. In addition, ZeroMQ provides prefix filtering. For example if a subscriber subscribes to message topic “t12”, it will receive any message that starts with “t12”. The messages with topics “t12”, “t123”, and “t12HappyBirthday” will all be received, while a message topic of “t1” will be filtered out.

***Connecting Publishers and Subscribers To The Network***

XideKit applications connect to the network through an existing Xideco component, the Xideco Router (xirt). If you installed Xideco, then *xirt* is available through the command line. All subscribers connect to the router using the router's IP address and a well known IP port number. Similarly, all publishers connect to the router using the same IP address, but with a well known IP port number for publishers.

The router may be located on any computer in the network and all subscribers and publishers, regardless of where they exist on the network, use the same will known IP address/port values.

The router consists of a single ZeroMQ Forwarder depicted below.



The forwarder does not filter any of the messages from its connected publishers, but simply transmits all messages to all connected subscribers with original payloads and topics intact. The individual subscribers provide their own message filtering. In this way, the router never needs to be modified, and can simultaneously support multiple applications.

### *The XideKit Base Class*

XideKit is a base class that provides the minimal API for a XideKit application. It is *not* an *abstract* base class and therefore can be instantiated when appropriate.

### *The XideKit API*

Here is a summary of the XideKit methods:

| | | |
|---|---|---|
| Method | __init__ | The __init__ method sets up all the ZeroMQ "plumbing" |
| Method | set_subscriber_topic | This method sets the subscriber topic. |
| Method | publish_payload | This method will publish a payload with the specified topic. |
| Method | receive_loop | This is the receive loop for zmq messages. |
| Method | incoming_message_processing | Override this method with a message processor for the application |
| Method | clean_up | Clean up before exiting - override if additional cleanup is necessary |

The XideKit module is installed as part of the Xideco package. Source code may be viewed here.

Let's look at the code in detail.

## *The __init__ method:*

```
28 class XideKit:
29     """
30 
31     This is a base class to be inherited by a derived Xideco application.
32 
33     To import use:
34 
35         from xideco.xidekit.xidekit import XideKit
36 
37     Methods that may be overwritten:  the __init__ method, the receive_loop and incoming_message_processing
38     """
39 
40     def __init__(self, router_ip_address=None, subscriber_port='43125', publisher_port='43124'):
41         """
42         The __init__ method sets up all the ZeroMQ "plumbing"
43 
44         :param router_ip_address: Xideco Router IP Address - if not specified, it will be set to the local computer
45         :param subscriber_port: Xideco router subscriber port. This must match that of the Xideco router
46         :param publisher_port: Xideco router publisher port. This must match that of the Xideco router
47         :return:
48         """
49 
50         # If no router address was specified, determine the IP address of the local machine
51         if router_ip_address:
52             self.router_ip_address = router_ip_address
53         else:
54             s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
55             # use the google dns
56             s.connect(('8.8.8.8', 0))
57             self.router_ip_address = s.getsockname()[0]
58 
59         print('\n*************************************')
60         print('Using router IP address: ' + self.router_ip_address)
61         print('*************************************')
62 
63         self.subscriber_port = subscriber_port
64         self.publisher_port = publisher_port
65 
66         # establish the zeriomq sub and pub sockets
67         self.context = zmq.Context()
68         self.subscriber = self.context.socket(zmq.SUB)
69         connect_string = "tcp://" + self.router_ip_address + ':' + self.subscriber_port
70         self.subscriber.connect(connect_string)
71 
72         self.publisher = self.context.socket(zmq.PUB)
73         connect_string = "tcp://" + self.router_ip_address + ':' + self.publisher_port
74         self.publisher.connect(connect_string)
75 
```

When instantiating a XideKit derived class, the following parameters may be specified:

**router_ip_address**

If not specified, the router is assumed to be the local computer.

**subscriber_port**

The router subscriber port number. It not specified,  a default value of '43125' is used.

**publisher_port**

The router publisher port number. It not specified, it uses a default value of '43124'.

**Lines 51-61:**

If no router IP address is specified, the IP address of the local machine is determined. The router IP address used by this instance is printed out for identification.

**Lines 63-64:**

The subscriber and publisher port numbers are saved.

**Line 67:**

A single ZeroMQ “context” is shared within a process. The context is instantiated on this line.

**Lines 68-74:**

Using the context, a publisher and subscriber are instantiated using TCP as the base transport mechanism. Each connects to its respective router port.

## ***set_subscriber_topic***

```
76      def set_subscriber_topic(self, topic):
77          """
78          This method sets the subscriber topic.
79
80          You can subscribe to multiple topics by calling this method for
81          each topic.
82          :param topic: A topic string
83          :return:
84          """
85          if not type(topic) is str:
86              raise TypeError('Subscriber topic must be a string')
87
88          self.subscriber.setsockopt(zmq.SUBSCRIBE, topic.encode())
```

After the class is instantiated, this method may be called to set a subscription topic. It may be called more than once, but must be called at least once to subscribe to messages. The topic may be an empty string, in which case all messages will be received without any filtering.

## *publish_payload*

```python
def publish_payload(self, payload, topic=''):
    """
    This method will publish a payload with the specified topic.

    :param payload: A dictionary of items
    :param topic: A string value
    :return:
    """
    if not type(topic) is str:
        raise TypeError('Publish topic must be a string', 'topic')

    if not type(payload) is dict:
        raise TypeError('Publish payload must be a dictionary', payload)

    # create a message pack payload
    message = umsgpack.packb(payload)

    pub_envelope = topic.encode()
    self.publisher.send_multipart([pub_envelope, message])
```

A payload dictionary and topic string are passed to this method. It builds the message and publishes it to the network.

## receive_loop

```python
110 def receive_loop(self):
111     """
112     This is the receive loop for zmq messages.
113
114     It is assumed that this method will be overwritten to meet the needs of the application and to handle
115     received messages.
116     :return:
117     """
118     while True:
119         try:
120             data = self.subscriber.recv_multipart(zmq.NOBLOCK)
121             self.incoming_message_processing(data[0].decode(), umsgpack.unpackb(data[1]))
122             time.sleep(.001)
123         except zmq.error.Again:
124             time.sleep(.001)
125         except KeyboardInterrupt:
126             self.clean_up()
```

This method listens for receipt of subscribed messages. It is non-blocking. If a message is available, lines 120 through 122 are executed. The message is processed by the call to incoming_message_processing. If no message is available, ZeroMQ throws a zmq.error.Again exception, and that exception is caught on line 123. Lastly, the method will call the clean_up method if Control-C is entered.

## incoming_message_processing

```python
    def incoming_message_processing(self, topic, payload):
        """
        Override this method with a message processor for the application

        :param topic: Message Topic string
        :param payload: Message Data
        :return:
        """
        print('this method should be overwritten in the child class', topic, payload)
```

This method is called when a message is received. It is overwritten by the user to meet the needs of the application. The topic string and payload dictionary are passed in.

## clean_up

```python

    def clean_up(self):
        """
        Clean up before exiting - override if additional cleanup is necessary

        :return:
        """
        self.publisher.close()
        self.subscriber.close()
        self.context.term()
        sys.exit(0)
```

This method closes the publisher and subscriber sockets,  terminates the context, and exits.

# Examples

Note: at the end of the discussion for each example, you will find a link for a video of the example in action.

## Example 1 – A Simple Publisher, Subscriber and Monitor

### The Publisher

```python
import time
import sys

from xideco.xidekit.xidekit import XideKit

# create 2 publisher instances
my_pub1 = XideKit()
my_pub2 = XideKit()

# initialize the message
message = 0

# have both publishers send message with contents of the current message value
# send a message every quarter of a second
while True:
    try:
        my_pub1.publish_payload({'info': message}, 'p1')
        print("Message sent from my_pub1 = {0}   Message sent from my_pub2 = {1}\n".format(message, message+1))
        my_pub2.publish_payload({'info': message + 1}, 'p2')
        message += 2
        time.sleep(.25)
    except KeyboardInterrupt:
        sys.exit(0)
```

This is an extremely simple example. We create 2 instances of the the XideKit class, my_pub1 and my_pub2. A forever loop is executed that publishes a payload to topic 'p1' though the first instance of XideKit and the topic 'p2' through the second instance. The payload is a simple counter that has the key of 'info'. The counter is continuously incremented.

## The Subscriber

```python
23 import sys
24
25 from xideco.xidekit.xidekit import XideKit
26
27
28 class MySub(XideKit):
29     def __init__(self, router_ip_address=None, subscriber_port='43125', publisher_port='43124'):
30         """
31         For this example we simply call the __init__ of the super class.
32         """
33         super().__init__(router_ip_address, subscriber_port, publisher_port)
34
35     def incoming_message_processing(self, topic, payload):
36         """
37         This method is overwritten in the inherited class to process the data
38         :param topic: Message topic string
39         :param payload: Message content
40         :return:
41         """
42         print("Message From {0} : {1} \n".format(topic, payload['info']))
43
44
45 if __name__ == '__main__':
46     try:
47         my_sub = MySub()
48         my_sub.set_subscriber_topic('p')
49         my_sub.receive_loop()
50     except KeyboardInterrupt:
51         sys.exit(0)
52
```

This example creates the MySub class that inherits from the XideKit class (line 28). For its __init__ method, it simply calls the __init__method of the XideKit super class.

It overrides the incoming_message_processing method to print out the topic and payload.

Line 47 instantiates the class, and line 48 subscribes to topic 'p'. If you recall, the publishers above are publishing topics 'p1' and 'p2'. Because ZeroMQ uses a prefix filter, both topics are subscribed to and their payloads will be printed. Finally, the receive_loop method is called, and the process will wait for any incoming messages that meet the subscribe topic criteria.

Let's run this. First we will execute the router by calling xirt. It was installed when Xideco was installed. It will announce the IP address it is running on. Next we will start the subscriber. It too will announce the IP address of the router and will wait until we start the publisher. Finally we start the publisher and we will the print statements being called for both processes.

## The Message Monitor

```
23 import sys
24
25 from xideco.xidekit.xidekit import XideKit
26
27
28 class MyMonitor(XideKit):
29     def __init__(self, router_ip_address=None, subscriber_port='43125', publisher_port='43124'):
30         """
31         This method monitors all messages going through a Xideco router.
32         :param router_ip_address: Xideco Router IP Address - if not specified, it will be set to the local computer
33         :param subscriber_port: Xideco router subscriber port. This must match that of the Xideco router
34         :param publisher_port: Xideco router publisher port. This must match that of the Xideco router
35         :return:
36         """
37         super().__init__(router_ip_address, subscriber_port, publisher_port)
38
39     def incoming_message_processing(self, topic, payload):
40         """
41         This method is overwritten in the inherited class to process the data
42         :param topic: Message topic string
43         :param payload: Message content
44         :return:
45         """
46         print(topic, payload)
47
48
49 if __name__ == '__main__':
50     try:
51         my_mon = MyMonitor()
52         my_mon.set_subscriber_topic('')
53         my_mon.receive_loop()
54     except KeyboardInterrupt:
55         sys.exit(0)
```

This is class on the subscriber class above, but the subscribed topic has been changed to subscribe to all messages. Line 52 subscribes to ' ', which means it subscribes to all messages.

The monitor may be connected while any XideKit application is running. It is a great tool to see what messages are being generated.

You may view a video of the demo using the three components here.

## Example 2 – Simple Arduino Control With A *tkinter* GUI


In this example, two classes are created and both inherit from the XideKit base class. One class creates a simple tkinter GUI that will control a digital output pin on an Arduino, while also displaying the value of an analog input pin. This class both publishes control messages and subscribes to data update messages. The second class communicates directly with the Arduino using the asyncio library, PyMata3. Its subscriber directly controls the state of a digital output pin, and its publisher sends out any changes detected on an analog input pin. Note the GUI is not an asyncio process, but inter-operates with the asyncio Arduino code. This is another example of application flexibility.

## The Gui Class

### __init__ Method Overwritten

```
import time
from tkinter import *
from tkinter import ttk

import umsgpack
import zmq

from xideco.xidekit.xidekit import XideKit


# noinspection PyMethodMayBeStatic,PyUnresolvedReferences
class Gui(XideKit):
    def __init__(self, router_ip_address=None, subscriber_port='43125', publisher_port='43124'):
        """
        :param router_ip_address: ip address of router
        :param subscriber_port: router subscriber port
        :param publisher_port: router publisher port
        :return:
        """
        super().__init__(router_ip_address, subscriber_port, publisher_port)

        # get instance of Tk and set as root
        self.root = Tk()

        # set the window title
        self.root.title('Simple XideKit Arduino/Gui Demo')

        # create the main frame, add a grid and configure the frame
        self.mainframe = ttk.Frame(self.root, padding="2 2 12 12")
        self.mainframe.grid(column=0, row=0, sticky=(N, W, E, S))
        self.mainframe.columnconfigure(0, weight=1)
        self.mainframe.rowconfigure(0, weight=1)

        # add a button that will ultimately turn an LED on and OFF the remote Arduino
        self.led = Button(self.mainframe, text="Blue LED On", command=self.on, background='red', width="30")
        self.led.grid(column=2, row=1, sticky=W)

        # add a label that will ultimately display the current value of a potentiometer connected to the remote
        # arduino.
        self.pot = Label(self.mainframe, text="No Data Received Yet", width="30")
        self.pot.grid(column=2, row=2, sticky=W)

        # adjust the layout with some padding
        for child in self.mainframe.winfo_children():
            child.grid_configure(padx=5, pady=5)
        self.mainframe.focus()
```

The GUI class overrides the base class __init__ method. It simply calls the base class __init__ and then creates the GUI using standard tkinter calls.

### Application Specific Methods

```
    def on(self):
        """
        When the button is pressed and going to an ON state, this method is called
        :return:
        """
        self.led.configure(bg='#00CC00', text="LED Off", command=self.off)
        # print('LED On')
        self.publish_payload({'command': 'On'}, "B")

    def off(self):
        """
        When the button is pressed and going to an OFF state, this method is called
        :return:
        """
        self.led.configure(bg='#FF0101', text="LED On", command=self.on)
        # print('LED Off')
        self.publish_payload({'command': 'Off'}, "B")
```

These methods are called when the GUI button is clicked. A ZeroMQ message is formed with the command of either 'on' or 'off', and then the message is published to the network. This method publishes with the topic of “B”.

### receive_loop Method Overwritten

```
87      def receive_loop(self):
88          """
89          This is the receive loop for zmq messages
90          It is assumed that this method may be overwritten to meet the needs of the application
91          It returns payload via user provided callback method
92          :return:
93          """
94          while True:
95              try:
96                  data = self.subscriber.recv_multipart(zmq.NOBLOCK)
97                  self.incoming_message_processing(data[0].decode(), umsgpack.unpackb(data[1]))
98                  time.sleep(.001)
99              except zmq.error.Again:
100                 try:
101                     time.sleep(.001)
102                     self.root.update()
103                 except KeyboardInterrupt:
104                     self.root.destroy()
105                     self.publisher.close()
106                     self.subscriber.close()
107                     self.context.term()
108                     sys.exit(0)
109             except KeyboardInterrupt:
110                 self.root.destroy()
111                 self.publisher.close()
112                 self.subscriber.close()
113                 self.context.term()
114                 sys.exit(0)
115
```

To allow the tkinter event loop to coexist within the context of receive_loop, line 102 calls root.update to refresh the gui.

### incoming_message_processing Method Overwritten

```python

    def incoming_message_processing(self, topic, payload):
        """
        This method processes the incoming message
        :param topic: topic string
        :param payload: message
        :return:
        """
        # extract the command from the message dictionary
        command = payload['command']

        # this should be an updated potentiometer value, so update the gui with the new value
        gui.pot.configure(text=command)


```

This method is called by receive_loop. It extracts the command from the payload (in this case the latest value from the analog input) and updates the GUI with that value.

### Invoking the GUI Process

```python
if __name__ == '__main__':
    gui = Gui()
    gui.set_subscriber_topic('A')
    # noinspection PyBroadException
    try:
        gui.receive_loop()
    except Exception:
        sys.exit(0)

```

An instance of the Gui class is instantiated, topic string 'A' is subscribed to and the receive loop is started.

## The Arduino Class

### __init__ Method Overwritten

```python
import asyncio

import umsgpack
import zmq
from pymata_aio.constants import Constants
from pymata_aio.pymata3 import PyMata3

from xideco.xidekit.xidekit import XideKit


class Arduino(XideKit):
    """
    The Arduino class encapsulates a PyMata3 instance to control a digital output pin
    and to receive data updates from an analog input pin.
    """
    # Constants
    BLUE_LED = 9  # digital pin number of blue LED
    POTENTIOMETER = 2  # analog pin number for the potentiometer
    DATA = 1  # position in callback data for current data value

    def __init__(self, router_ip_address=None, subscriber_port='43125', publisher_port='43124'):
        """
        This method instantiates a PyMata3 instance. It sets pin 9 as a digital output and analog pin 2 as sn input.
        A callback method is associated with the analog input pin to report the current value.

        :return:
        """
        super().__init__(router_ip_address, subscriber_port, publisher_port)

        self.board = PyMata3(3)
        self.board.set_pin_mode(self.BLUE_LED, Constants.OUTPUT)
        self.board.set_pin_mode(self.POTENTIOMETER, Constants.ANALOG, self.analog_callback)
```

Here, an instance of the PyMata3 class is instantiated to allow control and monitoring of the Arduino. The pin modes are set and for the potentiometer, and a callback method is specified to handle data update notifications.

### analog_callback Method

```python
    def analog_callback(self, data):
        """
        The method that PyMata3 calls when an analog value is being reported
        :param data: A list with the 2nd element containing the current value of the potentiometer
        :return:
        """
        value = str(data[self.DATA])
        self.publish_payload({'command': value}, "A")
        self.board.sleep(.01)
```

When a change in value is detected for the potentiometer, PyMata3  calls this method. It extracts the data value reported by PyMata3 and creates a payload message with a topic string of “A” and publishes it to the network for the GUI to consume.

### receive_loop Method Overwritten

```python
def receive_loop(self):
    """
    This is the receive loop for zmq messages
    It is assumed that this method will be overwritten to meet the needs of the application and to handle
    received messages.
    :return:
    """
    while True:
        try:
            data = self.subscriber.recv_multipart(zmq.NOBLOCK)
            self.incoming_message_processing(data[0].decode(), umsgpack.unpackb(data[1]))
            self.board.sleep(.01)
        except zmq.error.Again:
            try:
                self.board.sleep(.01)
            except:
                self.clean_up()
        except KeyboardInterrupt:
            self.clean_up()
```

This method is overwritten to call the PyMata3 sleep method. This sleep method is used instead of time.sleep in order to satisfy a PyMata3 asyncio requirement.

### incoming_message Method Overwritten

```python
def incoming_message_processing(self, topic, payload):
    """
    This method is overwritten in the inherited class to process the data
    :param topic: topic string
    :param payload: message data
    :return:
    """
    command = payload['command']
    if command == 'On':
        asyncio.ensure_future(self.board.core.digital_write(self.BLUE_LED, 1))
    elif command == 'Off':
        asyncio.ensure_future(self.board.core.digital_write(self.BLUE_LED, 0))
    self.board.sleep(.1)
```

This method processes the On/Off messages from the GUI. It is using the asyncio.ensure_future methods required by the PyMata3 library. Note that all of the asyncio processing is confined to this process. The GUI is not an asyncio process and is totally unaware of asyncio.

### Invoking the Arduino Process

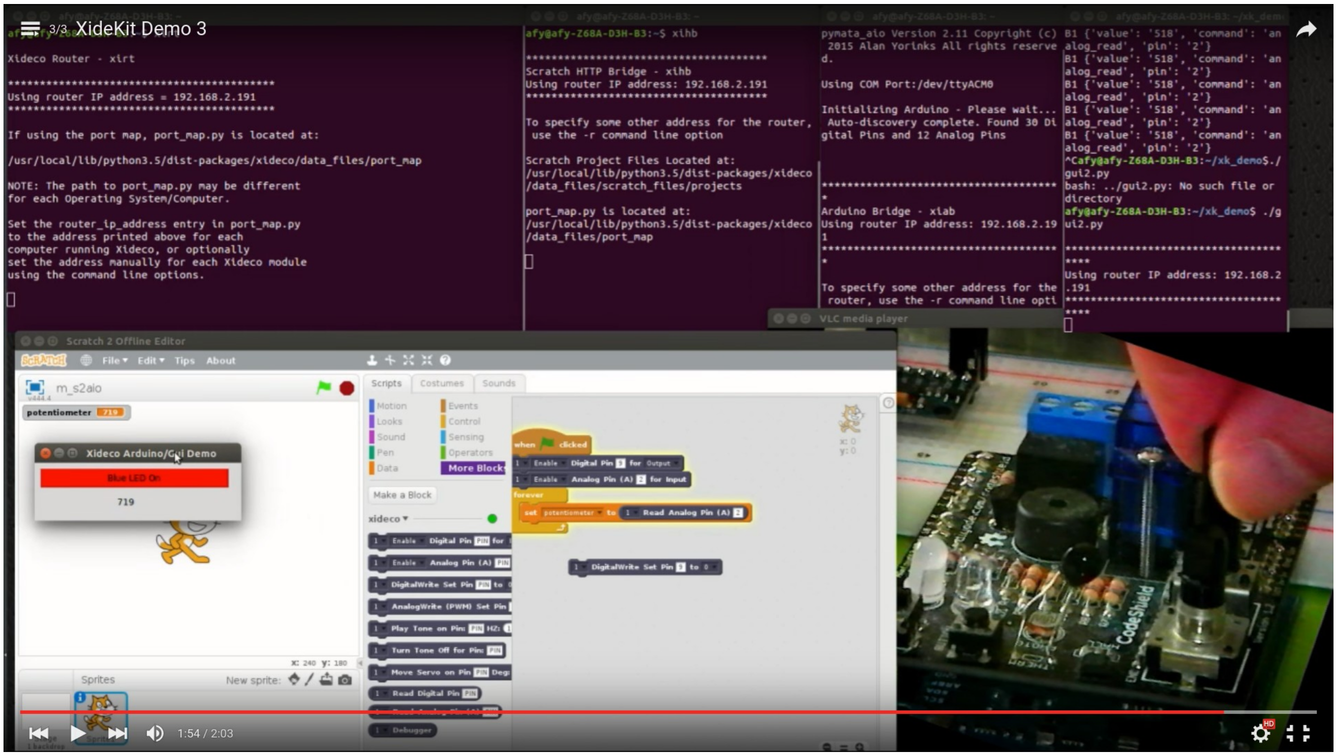
```python
if __name__ == "__main__":
    arduino = Arduino()
    arduino.set_subscriber_topic('B')
    arduino.receive_loop()
```

An instance of the derived Arduino class is instantiated, it then subscribes to 'B' as its topic and finally starts the receive_loop.

### Running The Example

To run the example, open a command window and execute xirt to start the router. Open another command window and execute the GUI, and finally open a third command window and execute Arduino. [Here is a video](#) demonstrating this.

## Example 3




### Modifying The GUI Class To Run with Xideco ([gui2.py](#))

This example modifies the gui class from example 2 to communicate directly with a currently running Xideco session. It demonstrates that both Scratch and the tkinter GUI can be attached to the network simultaneously and either one can be used to control the LED on the Arduino. Both Scratch and the tkinter GUI subscribe for analog input updates and updates are displayed in unison on both interfaces.

### Modifying __init__ To Initialize The Arduino Pins

```
        # set up pin modes on arduino
        # digital output for pin 9
        cmd = {u"command": "digital_pin_mode", u"enable": "Enable", u"pin": "9", u"mode": "Output"}
        self.publish_payload(cmd, "A1")

        # analog input for pin A2
        cmd = {u"command": "analog_pin_mode", u"enable": "Enable", u"pin": "2"}
        self.publish_payload(cmd, "A1")
```

The code above is added to the end of the __init__ method. It conforms to the Xideco protocol to set the pin modes for the Arduino used in the example. It publishes the payloads to topic “A1”

## Modifying the *on* and *off* Methods

```python
    def on(self):
        """
        When the button is pressed and going to an ON state, this method is called
        :return:
        """
        command = "digital_write"
        pin = 9
        value = 1

        cmd = {u"command": command, u"pin": pin, u"value": value}
        self.led.configure(bg='#00CC00', text="LED Off", command=self.off)
        # print('LED On')
        self.publish_payload(cmd, "A1")

    def off(self):
        """
        When the button is pressed and going to an OFF state, this method is called
        :return:
        """
        command = "digital_write"
        pin = 9
        value = 0
        self.led.configure(bg='#FF0101', text="LED On", command=self.on)
        # print('LED Off')
        cmd = {u"command": command, u"pin": pin, u"value": value}

        self.publish_payload(cmd, "A1")
```

The payloads and topics were modified from example 2 to conform to the Xideco messaging protocol.

## Modifying the incoming_message_processing Method

```python

    def incoming_message_processing(self, topic, payload):
        """
        This is the message processor
        :param topic: topic string
        :param payload: message data
        :return:
        """
        # extract the command from the message dictionary
        if payload['command'] == 'analog_read':
            data = str(payload['value'])

            # this should be an updated potentiometer value, so update the gui with the new value
            gui.pot.configure(text=data)


if __name__ == '__main__':
    gui = Gui()
    gui.set_subscriber_topic('B1')
    # noinspection PyBroadException
    try:
        gui.receive_loop()
    except Exception:
        sys.exit(0)
```

The incoming_message_processing method is also modified to conform to the Xideco protocol to receive the analog input updates.

The subscription topic for the class is changed to 'B1', again to conform to the Xideco protocol.

## Running Example 3

This example starts with a Xideco/Scratch/Arduino application. A video of the demo may be found here.

The demo begins with a Scratch project that will configure pin 9 as a digital output and pin A2 as an analog input.

Next, xirt is invoked, followed by starting xihb, the Xideco HTTP bridge. This bridge receives the HTTP message from Scratch, translates them to the Xideco protocol and publishes the messages. It also subscribes to data update messages. Xihb translates these messages to Scratch HTTP reporter messages.

Note that after xihb is started the little red indicator on the Scratch editor turns green, indicating that Scratch is successfully connected to and communicating with xihb.

After that, xiab, the Xideco Arduino bridge is started. We add a Scratch digital_write block on the Scratch editor for pin 9 and when we execute it, nothing happens. To check to make sure that the messages are being sent from xihb, we start up the monitor. Clicking on the digital_write_block shows that the correct message is being sent, but the LED still does not light. The reason is that we need to click on the green flag in Scratch to start the script from the beginning where the pins will be configured. We then see a flurry of activity on the monitor. This is the analog input value report messages being sent from xiab. Now when the digital_write block is clicked, the LED can be controlled and if we change the value of the potentiometer, we can see that the variable in the upper left hand corner of Scratch is changing as well.

Finally we invoke gui2.py while all of Xideco is still running, and using the GUI, we can control the LED from either Scratch or the GUI and we can see that potentiometer updates are simultaneously being received by Scratch and the GUI.