Prio API

Prio API

Immersive Virtual World
Interaction System

User's Manual

YEI Technology
630 Second Street
Portsmouth, Ohio 45662

www.YeiTechnology.com
www.PrioVR.com

©2007-2014 YEI Corporation
Printed in USA

This page intentionally left blank

This page intentionally left blank

Prio API

Immersive Virtual World
Interaction System

User's Manual

YEI Technology
630 Second Street
Portsmouth, Ohio 45662

Toll-Free: 888-395-9029
Phone: 740-355-9029

www.YeiTechnology.com
www.PrioVR.com

©2007-2014 YEI Corporation
Printed in USA

Table of Contents

1. Usage/Safety Considerations....1
  1.1. Usage Conditions....1
  1.2. Technical Support and Repairs....1
2. Overview of the YEI Prio API....2
  2.1. Introduction....2
  2.2. System Requirements....2
  2.3. License....2
3. Getting Started with the Prio API....3
  3.1. Prio & API Terminology....3
  3.2. Prio Logical Ids....4
  3.3. Setting Up the Prio API....4
4. Using the Prio API....5
  4.1. Creating and Destroying Instances with the Prio API....5
  4.2. Streaming Data with the Prio API....6
    4.2.1. Setting the Streaming Slots....6
    4.2.2. Getting the Size of the Stream Data....7
    4.2.3. Getting the Stream Data: Non-blocking....8
    4.2.4. Getting the Stream Data: Blocking....8
    4.2.5. Getting the Stream Data: Recorded....9
    4.2.6. Stopping Streaming and Clean Up....11
5. Plug-ins and Wrappers....12
  5.1. Using the Prio API with Unity & C#....12
    5.1.1. Setting up the Plug-in....12
    5.1.2. Using the Plug-in....13
  5.2. Using the Prio API with Python....13
6. Reference Guide....14
  6.1. Prio API Specific Methods....14
  6.2. Prio API Command Methods....16

This page intentionally left blank

This page intentionally left blank

1. Usage/Safety Considerations

1.1. Usage Conditions

- Do not use Prio devices in any system on which people's lives depend (life support, weapons, etc.)
- Because of its reliance on a magnetometer, Prio devices will not work properly near the earth's north or south pole.
- Because of its reliance on a magnetometer and accelerometer, Prio devices will not work properly in outer space or on planets with no magnetic field.
- Care should be taken when using Prio devices in a car or other moving vehicle, as the disturbances caused by the vehicle's acceleration may cause the sensor to give inaccurate readings.
- Because of its reliance on a magnetometer, care should be taken when using Prio devices near ferrous metal structures, magnetic fields, current carrying conductors, and should be kept about 6 inches away from any computer screens or towers.
- Since Prio devices use RF communication technology, communication failure modes should be carefully considered when designing a system that uses Prio devices.
- The Prio Hub is powered by a rechargeable lithium-polymer battery. Lithium-polymer batteries have high energy densities and can be dangerous if not used properly.
- Refer to the Usage/Safety Considerations section of the *YEI Prio User's Manual* for further information pertaining to safety of Prio devices.

1.2. Technical Support and Repairs

YEI provides technical and user support via our toll-free number (888-395-9029), via email (support@yeitech.com), and via community forum (forum.yeitechnology.com). Support is provided for the lifetime of the product. Requests for repairs should be made through the Support department.

2. Overview of the YEI Prio API

2.1. Introduction

The YEI Prio API is an open source application program interface for the YEI Prio devices. The API is a collection of convenience functions that wrap all normal functionality of the Prio devices for use in a program written in C/C++ or any language that can import a compiled library (.dll, .so, etc). YEI Technology offers a 32-bit and 64-bit version of the API.

2.2. System Requirements

Operating System:

- Windows 7 or higher (32-Bit/64-Bit)
- Linux (Coming Soon)
- Mac OS (Coming Soon)

Hardware:

- USB Port (2.0 or higher recommended)

2.3. License

The YEI Prio API is released under the YEI 3-Space Open Source License, which allows for both non-commercial use and commercial use with certain restrictions.

- For Non-Commercial Use, your use of Covered Works is governed by the GNU GPL v.3, subject to the YEI 3-Space Open Source Licensing Overview and Definitions.
- For Commercial Use, a YEI Commercial/Redistribution License is required, pursuant to the YEI 3-Space Open Source

Licensing Overview and Definitions. Commercial Use, for the purposes of this License, means the use, reproduction and/or Distribution, either directly or indirectly, of the Covered Works or any portion thereof, or a Compilation, Improvement, or Modification, for Pecuniary Gain. A YEI Commercial/Redistribution License may or may not require payment, depending upon the intended use.

Full details of the YEI 3-Space Open Source License can be found online at http://www.yeitechnology.com/yei-3-space-open-source-license

3. Getting Started with the Prio API

Prio device data can be accessed using the command protocol discussed in the *YEI Prio User's Manual*, but there are a few things to know first that will help you to understand how the system works and what the data you are getting means.

3.1. Prio & API Terminology

There are a few terms you should be familiar with when using Prio devices and the API.

Tare – Taring is the process of setting a zero orientation for a sensor. When the sensor refers to a "tared orientation", it means an orientation reading that is relative to the zero orientation that has been set for that sensor.

Corrected Data – Corrected data is data from the component sensors of a Prio sensor(a gyroscope, accelerometer, or magnetometer) that has been scaled into real world units and has had a bias applied.

Raw Data – Raw data is data from the component sensors that is in its most basic form, directly as it came from the component sensors. This data needs to be scaled and biased before being used.

Calib Params – Calibration parameters determine how a sensor is scaled and biased. They consist of 9 values representing a row major scale/rotation matrix, and then 3 values representing a bias. They are used on the sensor by adding the bias on the component sensor values, then by multiplying the value by the scale matrix.

Beta Calc Params – The beta calculation parameters influence how the orientation filter settles.

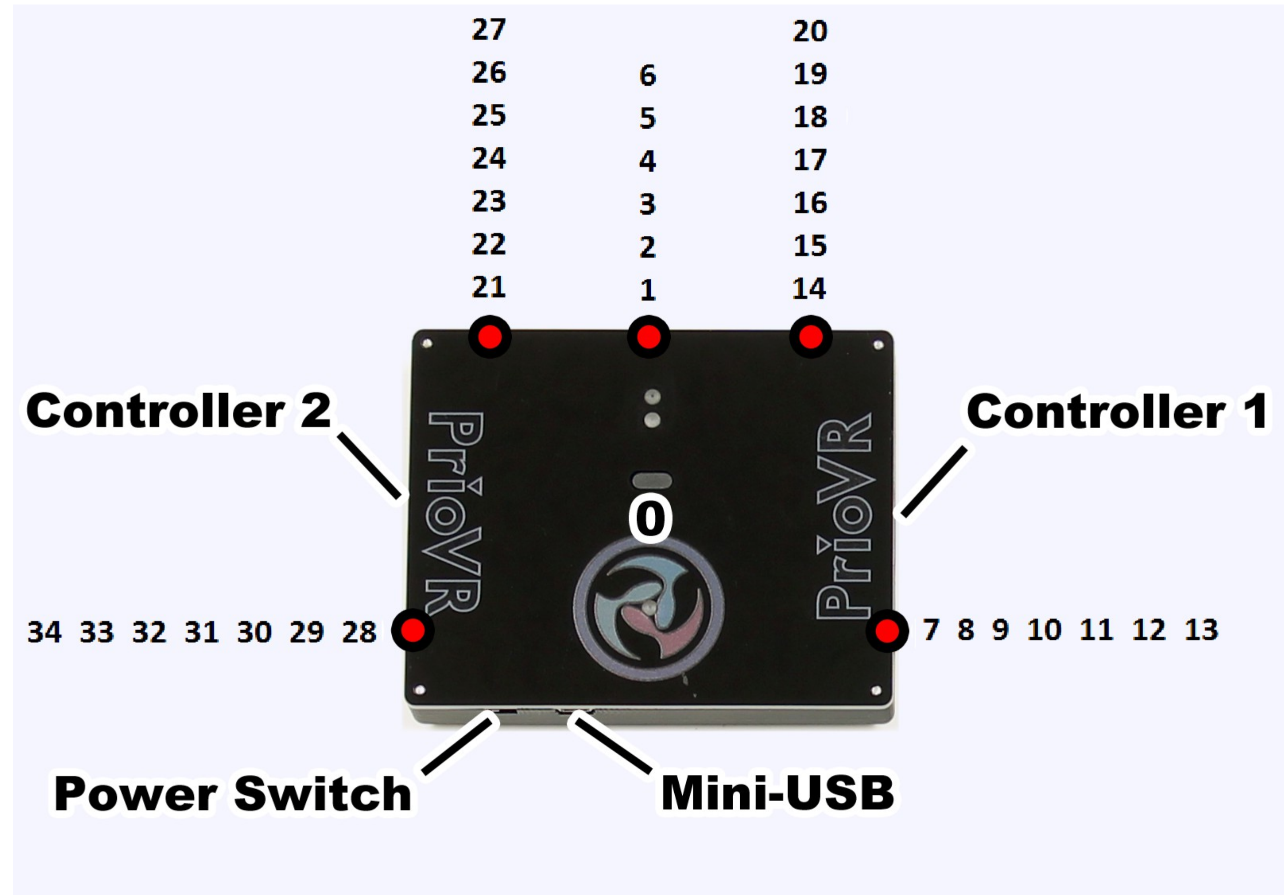
Streaming – When the sensor outputs data automatically, without data needing to be requested.

Logical ID – The logical ID is an ID that is used for sending commands to specific Prio devices. The next section goes over this in more detail.

Device ID – The device ID is an ID that the API uses to communicate with Prio devices. This ID is needed for many functions of the API.

3.2. Prio Logical Ids

When calling certain functions in the API a logical ID must be given to specify which device to communicate with (the Prio Base Station, Hub, or a sensor node). When calling a function to the Base Station, the defined macro *PRIO_BASE_STATION_LOGICAL_ID* should be used. When calling a function to the Hub, the defined macro *PRIO_HUB_LOGICAL_ID* should be used. When sending a command to all sensor nodes, the defined macro *PRIO_BROADCAST_LOGICAL_ID* should be used. Finally, when sending a command to a single sensor node, use the following chart to determine the logical ID of that particular sensor node. The numbers closer to the hub indicate sensor nodes closer to the hub in their chain. The sensor node on the hub itself has the logical ID 0.



Rev1 Prio Hub Chart

3.3. Setting Up the Prio API

Before using the Prio API it is best to be sure to have the latest API and firmware on the Prio devices.

For instructions for setting up to use the Prio API in Visual Studio please refer to the documentation *YEI Prio API Quick Start Guide: Setup Environment for Visual Studio*.

4. Using the Prio API

The Prio API makes it easy for users to get, set, stream, and broadcast data to and from the Prio devices and to find Prio devices on the system.

For an overview of the different ways of finding Prio devices and creating API instances, please refer to the *YEI Prio API Quick Start Guide* or the example *creating_instances.c* file.

4.1. Creating and Destroying Instances with the Prio API

The first step in using the API is connecting to a Prio device. To do this we must know which serial port the Prio device is associated with. The API makes this simple with the *prio_getComPorts* function. The following snippet shows how to find a Prio device and create an instance for the API to use.

```
prio_ComPort com_port;
prio_device_id prio_device;

prio_getComPorts(&com_port, 1, offset, PRIO_FIND_ALL_KNOWN)
prio_device = prio_createDeviceStr(com_port.com_port, PRIO_TIMESTAMP_SYSTEM);
if (prio_device == PRIO_NO_DEVICE_ID)
{
    printf("Could not create a device\n");
    return 1;
}
```

The *prio_getComPorts* function finds all of the serial ports on the system and filters them based on whether a Prio device is associated with it or not. It writes the found serial ports to an array of *prio_ComPort* structures that is passed in as a parameter.

Once the serial port is known, use the *prio_createDeviceStr* function to generate a device ID to use with the other API functions. The device ID is used to directly interact with the Prio device on that serial port. Note that once a device ID is successfully created, the serial port used is closed to other programs. If the serial port is needed by another program the function *prio_disconnectDevice* will free the serial port without destroying the instance created by the API. It is also good practice when using the *prio_createDeviceStr* function to check if the device ID is a valid one. The API provides an enum of device ID masks that can be used to check this.

Once a Prio device instance is created it sends a call to the Prio Hub to return a bit-field of enumerated sensor nodes. This lets the API know what sensor nodes the Prio Hub thinks are present and active. To be sure what sensor nodes are present and active, it is best to call the *prio_enumerateHub* function.

```
PRIO_ERROR error;

error = prio_enumerateHub(prio_device, NULL);
```

The function takes a *prio_device_id* type and an unsigned integer pointer for a timestamp as parameters, and returns a *PRIO_ERROR* enum type that denotes whether or not an error occured. Almost every API function has these parameters and returns this type. It is always best to check if a command was successful as it is helpful in debugging issues, especially with wireless communication.

After this initial setup the Prio device is ready to start communicating with your application. Please note that when you are finished with the Prio device calling *prio_destroyDevice* will free the serial port and destroy the instance. Also once you are done with the API, calling *prio_resetPrioApi* will clean up anything remaining in the library and properly stop any streaming and destroy loose instances.

```
prio_destroyDevice(prio_device);

error = prio_resetPrioApi();
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

4.2. Streaming Data with the Prio API

Other than using write/read commands to get data from a Prio device, the Prio API is able to make use of the Prio device's streaming option. It does so by using C++11's multithreading library. Streaming data is the best option for real-time applications.

4.2.1. Setting the Streaming Slots

The Prio devices allows users to choose up to 8 different types of data they want to stream by setting the streaming slots of the device. The API makes this easy by providing a *PRIO_STREAM_COMMAND* enum type that denotes what commands can be streamed and a *prio_stream_commands* type that the function *prio_setStreamingSlots* takes as a parameter. By default the streaming slots are set to only stream the tared orientation as a quaternion from the sensor nodes creation.

```
uint32_t timestamp;
prio_stream_commands prio_slots[8] =
{
    PRIO_GET_TARED_ORIENTATION_AS_QUATERNION,
    PRIO_GET_CORRECTED_ACCELEROMETER_VECTOR,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL,
    PRIO_NULL
};

error = prio_setStreamingSlots(prio_device, prio_slots, &timestamp);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

4.2.2. Getting the Size of the Stream Data

Once the streaming slots are set to return the desired data, we need to know the full size of the stream data. To start, we can calculate or get the size of the data denoted by the streaming slots. In the previous example, the streaming slots are set up to stream the tared orientation as a quaternion and corrected accelerometer data from the sensor nodes. Note that each stream command returns a different number of floats. From this information we can calculate the total size; 4 floats for the quaternion and 3 floats for the accelerometer data is a total of 7 floats. Alternatively, the API provides the function *prio_getLengthOfStreamData* that requires less knowledge of the streaming slots.

```
uint32_t stream_data_len = 0;
error = prio_getLengthOfStreamData(prio_device, &stream_data_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Next, we need to get the number of active sensor nodes from the Prio device. This can be done using the *prio_getActiveSensors* function.

```
uint8_t* active_nodes;
uint8_t active_nodes_len = 0;
error = prio_getActiveSensors(prio_device, NULL, 0, &active_nodes_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
active_nodes = (uint8_t*)malloc(active_nodes_len * sizeof(uint8_t));
error = prio_getActiveSensors(prio_device, active_nodes, active_nodes_len, NULL);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Finally, we multiply the numbers together to get the full length of the stream data and we can now be sure we have allocated enough memory for getting the stream data. The API also provides another way of getting the full length of the stream data with the function *prio_getFullLengthOfStreamData*.

```
uint32_t full_stream_data_len = 0;
error = prio_getFullLengthOfStreamData(prio_device, &full_stream_data_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Now that we know how much memory needs to be allocated, we can start streaming with the function *prio_startStreaming* and begin getting the stream data. The API provides a few methods to retrieve the stream data: non-blocking, blocking, and recorded. Each method is designed for an array of different applications in mind. Along with getting the stream data additional is the stream header data that is conveniently removed into the *prio_StreamHeaderData* structure that each method takes as a parameter.

```
prio_StreamHeaderData header_data;
float* stream_data;
uint32_t stream_timestamp;
stream_data = (float*)malloc(full_stream_data_len * sizeof(float));

error = prio_startStreaming(prio_device, 0, NULL);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

4.2.3. Getting the Stream Data: Non-blocking

The non-blocking method, *prio_getLastStreamData*, is ideal for real-time applications as it returns information the fastest.

```
// In some kind of loop
error = prio_getLastStreamData(prio_device, &header_data, stream_data,
                               full_stream_data_len, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

4.2.4. Getting the Stream Data: Blocking

The blocking method, *prio_getLatestStreamData*, is ideal for applications where unique data is required, such as when data logging. The function accomplishes this by taking a timeout parameter in milliseconds which tells it how long to wait for new data to arrive.

```
// In some kind of loop
error = prio_getLatestStreamData(prio_device, &header_data, stream_data,
                                 full_stream_data_len, 16, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

4.2.5. Getting the Stream Data: Recorded

When receiving recorded stream data, we need to set a sample size for the recording. By default it is set to the max size of a std::deque container. It is best to set a max sample size of the recording using the *prio_setMaxRecordedSamples* function. Note that this container is a First-In-First-Out (FIFO) data structure, and once the max size is reached the oldest data in the container will be removed.

```
uint32_t max_samples_len;
...
error = prio_setMaxRecordedSamples(prio_device, max_samples_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Once the max sample size is set, we need to tell the API to start recording. This can be done in 2 ways. One way is to set the start_recording parameter for the *prio_startStreaming* function to some value greater than 0. The other is to call the *prio_startRecordingData* function after streaming has already started.

Now we must decide what method we wish to use to get the recorded samples. The API provides 3 methods to gett recorded samples.

One method is to use the *prio_popFrontRecordedSample* function. This function removes and returns the recorded sample at the front of the recording container.

```
// In some kind of loop
error = prio_popFrontRecordedSample(prio_device, &header_data, stream_data,
                                    full_stream_data_len, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Another method is to use the *prio_getRecordedSampleAtIndex* function. This function returns the recorded sample at the given index from the recording container.

```
uint32_t idx;
...
// In some kind of loop
error = prio_getRecordedSampleAtIndex(prio_device, &header_data, stream_data,
                                      full_stream_data_len, idx, &stream_timestamp);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

The last method available is to use the *prio_getRecordedSamples* function. This function returns all of the recorded samples in the recording container. With this method some setup is necessary. Specifically, you will need to find the size of the recorded container using *prio_getLengthOfRecordedSamples* and *prio_getFullLengthOfRecordedSamples*, and stopping the recording using *prio_stopRecordingData* is necessary.

```
error = prio_stopRecordingData(prio_device);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}

uint32_t recorded_samples_len = 0;
error = prio_getLengthOfRecordedSamples(prio_device, &recorded_samples_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}

uint32_t full_recorded_samples_len = 0;
error = prio_getFullLengthOfRecordedSamples(prio_device,
                                            &full_recorded_samples_len);
if (error != PRIO_NO_ERROR)
{
    printf("ERROR: %s\n", prio_error_string[error]);
}

prio_StreamHeaderData* recorded_header_data;
float* recorded_data;
uint32_t* recorded_timestamps;
recorded_header_data = (prio_StreamHeaderData*)malloc(
    recorded_samples_len * sizeof(prio_StreamHeaderData));
recorded_data = (float*)malloc(full_recorded_samples_len * sizeof(float));
recorded_timestamps = (uint32_t*)malloc(recorded_samples_len * sizeof(uint32_t));

error = prio_getRecordedSamples(prio_device,
                                &recorded_header_data, recorded_samples_len,
                                recorded_data, full_recorded_samples_len,
                                &recorded_timestamps, recorded_samples_len);
if (error == PRIO_NO_ERROR)
{
    // Do something...
}
else
{
    printf("ERROR: %s\n", prio_error_string[error]);
}
```

Note that allocating memory for the header data and timestamps is only necessary if you want that data to be returned.

4.2.6. Stopping Streaming and Clean Up

To stop streaming call the *prio_stopStreaming* function. This will also stop recording if it is still recording. Once streaming has stopped, it is best to clean up any allocated memory we no longer need. To clean up allocated memory the API used for recording samples, call the *prio_clearRecordedSamples* function.

5. Plug-ins and Wrappers

The Prio API can be integrated into many project types. This section will provide overviews on how to incorporate the API into game engines and other languages. The below examples assume some level of experience with each system.

5.1. Using the Prio API with Unity & C#

This section is to help those that would like to create their own Unity plug-in for the Prio API. For users that want to use the Prio API Unity Plug-in provided by YEI Technology, please refer to the *YEI Prio API Unity Plug-in Quick Start Guide*.

5.1.1. Setting up the Plug-in

To use the API in any version of Unity, the library file should be placed in the **Unity\Editor** folder and in the same directory as the build project executable. To use the API with the Unity Pro version, follow the instructions on Unity's website.

To make calls to the API functions a C# script can be used to wrap the methods needed. The following is a snippet example from the *prio_api.cs* file used for the Prio API Unity Plug-in. Please refer to this file for more examples.

```
namespace prio_api
{
    public enum PRIO_ERROR
    {
        PRIO_NO_ERROR,
        PRIO_ERROR_COMMAND_FAIL,
        PRIO_INVALID_COMMAND,
        PRIO_INVALID_LOGICAL_ID,
        PRIO_INVALID_DEVICE_ID,
        PRIO_ERROR_PARAMETER,
        PRIO_ERROR_TIMEOUT,
        PRIO_ERROR_WRITE,
        PRIO_ERROR_READ,
        PRIO_ERROR_STREAM_SLOTS_FULL,
        PRIO_ERROR_STREAM_CONFIG,
        PRIO_ERROR_MEMORY,
        PRIO_ERROR_FIRMWARE_INCOMPATIBLE,
        PRIO_ERROR_RESET_API
    }

    public static class prio
    {
        [DllImport("Prio_API_32",
                   CallingConvention = CallingConvention.Cdecl,
                   EntryPoint= "prio_startStreaming")]

        public static extern PRIO_ERROR startStreaming(uint device,
                                                       uint start_recording,
                                                       ref uint timestamp);
    }
}
```

5.1.2. Using the Plug-in

Now importing the file into any Unity script will grant access to all the newly wrapped API methods and attributes.

```
void Start()
{
    uint timestamp = 0;
    PRIO_ERROR error = prio.startStreaming(device, 0, timestamp);
    if (error)
    {
        Debug.Log("Could not start streaming {0}", error);
    }
    _is_streaming = true;
    Debug.Log("=============Start Streaming=============");
}
```

For more information on Unity plug-ins please refer to the Unity documentation.

5.2. Using the Prio API with Python

This section is to help those that would like to create their own Python wrapper for the Prio API. For users that want to use the Prio API Python wrapper provided by YEI Technology, please refer to the *YEI Prio Python Wrapper Quick Start Guide*.

The following is a snippet example from the *prio_api.py* file used for the Prio API Python wrapper. Please refer to this file for more examples.

```
from ctypes import cdll

PRIO_API = cdll.LoadLibrary('./Prio_API_32.dll')

def startStreaming(device, start_recording, timestamp):
      return PRIO_API.prio_startStreaming(device, start_recording, timestamp)
```

For more information on developing library wrappers please refer to the Python documentation.

6. Reference Guide

6.1. Prio API Specific Methods

The Prio API specific methods are convenience functions that allow for easy control of Prio devices and the streaming of data.

Name	Description
prio_initPrioApi	Initializes the Prio API.
prio_delPrioApi	Deletes the data of the Prio API.
prio_resetPrioApi	Reset the Prio API.
prio_setSoftwareWirelessRetries	Sets the amount of retires on the wireless commands via software.
prio_getSoftwareWirelessRetries	Gets the amount of retries on the wireless commands via software.
prio_getComPorts	Gets the serial ports of YEI Prio devices and other available serial ports.
prio_createDeviceU16Str	Creates a device ID with wide char support.
prio_createDeviceStr	Creates a device ID.
prio_getDeviceInfo	Gets information from the Prio device associated with the device ID.
prio_destroyDevice	Destroys the device and frees the serial port.
prio_reconnectDevice	Reconnects the device to the serial port.
prio_disconnectDevice	Disconnects the device and frees the serial port.
prio_setTimestampMode	Sets the timestamp mode on an existing Prio device instance.
prio_isSensorPresent	Checks if the sensor node at the logical ID is present on the Prio device.
prio_getActiveSensors	Gets a list of active sensor nodes at the logical IDs on the Prio device.
prio_writeReadCommand	Writes and reads a raw command to a Prio device.
prio_isConnected	Checks if the Prio device is connected.
prio_getDeviceType	Gets the device type of a Prio device.
prio_getDeviceState	Gets the device state of a Prio device.
prio_getStoredSerialNumber	Gets the stored serial number of a Prio device.
prio_getStoredSerialNumberAsHex	Gets the stored serial number of a Prio device as a hex string.
prio_getLastStreamHeaderData	Gets the last stream header data from Prio device.
prio_getLengthOfStreamData	Gets the length of the stream data from Prio device.
prio_getFullLengthOfStreamData	Gets the full length of the stream data from Prio device.
prio_getLastStreamData	Gets the last stream data from Prio device.
prio_getLatestStreamData	Gets the latest stream data from Prio device.
prio_setMaxRecordedSamples	Sets the max length of samples that the recording container can store.
prio_getMaxRecordedSamples	Gets the max length of samples that the recording container can store.
prio_startRecordingData	Starts recording the stream data from Prio device.
prio_stopRecordingData	Stops recording the stream data from Prio device.
prio_getLengthOfRecordedSamples	Gets the current length of samples that the recording container has stored.
prio_getFullLengthOfRecordedSamples	Gets the full length of the data of all the samples that the recording container has stored.

Name	Description
prio_getRecordedSamples	Gets all the samples from the recording container for the Prio device.
prio_getRecordedSampleAtIndex	Gets a sample at the index from the recording container for the Prio device.
prio_popFrontRecordedSample	Gets and removes the sample at the front of the recording container for the Prio device.
prio_clearRecordedSamples	Clears the samples from the recording container for the Prio device.

6.2. Prio API Command Methods

The API command methods are wrapper functions that allow for a more intuitive use of the sensor commands from the *Prio User's Manual* section 4.6.

Name	Description	Command
prio_getTaredOrientationAsQuaternion	Gets tared orientation from sensor node at the logical ID.	0(0x00)
prio_getUntaredOrientationAsQuaternion	Gets untared orientation from sensor node at the logical ID.	6(0x06)
prio_setOffsetWithQuaternion	Sets an offset quaternion to sensor node at the logical ID.	21(0x15)
prio_getAllCorrectedSensorData	Gets all corrected sensor data from sensor node at the logical ID.	37(0x25)
prio_getCorrectedGyroscope	Gets corrected gyroscope data from sensor node at the logical ID.	38(0x26)
prio_getCorrectedAccelerometer	Gets corrected accelerometer data from sensor node at the logical ID.	39(0x27)
prio_getCorrectedMagnetometer	Gets corrected magnetometer data from sensor node at the logical ID.	40(0x28)
prio_getAllRawSensorData	Gets all raw sensor data from sensor node at the logical ID.	64(0x40)
prio_getRawGyroscope	Gets raw gyroscope data from sensor node at the logical ID.	65(0x41)
prio_getRawAccelerometer	Gets raw accelerometer data from sensor node at the logical ID.	66(0x42)
prio_getRawMagnetometer	Gets raw magnetometer data from sensor node at the logical ID.	67(0x43)
prio_setStreamingSlots	Sets streaming slots for the device.	80(0x50)
prio_getStreamingSlots	Gets streaming slots from the device.	81(0x51)
prio_startStreaming	Starts/Resumes streaming on the device.	85(0x55)
prio_stopStreaming	Stops/Pauses streaming on the device.	86(0x56)
prio_setTimestamp	Sets the device's timestamp on the device. (Base Station only)	95(0x5f)
prio_setTareWithCurrentOrientation	Sets the tare to the current orientation of sensor node at the logical ID.	96(0x60)
prio_setMagnetometerCalibParams	Sets magnetometer calibration parameters on sensor node at the logical ID.	160(0xa0)
prio_getMagnetometerCalibParams	Gets magnetometer calibration parameters from sensor node at the logical ID.	162(0xa2)
prio_setBetaCalcParams	Sets beta calculation parameters on sensor node at the logical ID.	167(0xa7)
prio_getBetaCalcParams	Gets beta calculation parameters from sensor node at the logical ID.	168(0xa8)
prio_setAutoCalibEnabled	Sets auto calibration enabled to sensor node at the logical ID.	169(0xa9)
prio_getAutoCalibEnabled	Gets auto calibration enabled from sensor node at the logical ID.	170(0xaa)

Name	Description	Command
prio_enumerateHub	Enumerates paired Hub.	176(0xb0)
prio_getHubEnumerationValue	Gets enumeration value from paired Hub.	177(0xb1)
prio_getCountOfEnumeratedSensors	Gets total count of enumerated sensor nodes.	178(0xb2)
prio_resetAllSensors	Resets all sensor nodes.	179(0xb3)
prio_powerDownAllSensors	Powers down all sensor nodes.	180(0xb4)
prio_powerUpAllSensors	Powers up all sensor nodes.	181(0xb5)
prio_broadcastSynchronizationPulse	Synchronizes listening devices timestamps. (Base Station only)	182(0xb6)
prio_autoPairBaseStationWithHub	Pairs device with a Hub that has a button pressed. (Base Station only)	186(0xba)
prio_getWirelessPanID	Gets wireless pan ID from device.	192(0xc0)
prio_setWirelessPanID	Sets wireless pan ID to device.	193(0xc1)
prio_getWirelessChannel	Gets wireless channel from device.	194(0xc2)
prio_setWirelessChannel	Sets wireless channel to device.	195(0xc3)
prio_setLedMode	Sets LED mode to device. (Base Station only)	196(0xc4)
prio_commitWirelessSettings	Commits wireless settings to device. (Base Station only)	197(0xc5)
prio_getWirelessAddress	Gets wireless address from device. (Unavailable)	198(0xc6)
prio_setWirelessAddress	Sets wireless address to device. (Unavailable)	199(0xc7)
prio_getLedMode	Gets LED mode from device. (Base Station only)	200(0xc8)
prio_getBatteryPercentRemaining	Gets battery value from paired Hub.	202(0xca)
prio_getSerialNumberAtLogicalID	Gets serial number from device at the logical ID. (Base Station only)	208(0xd0)
prio_setSerialNumberAtLogicalID	Sets serial number of device at the logical ID. (Base Station only)	209(0xd1)
prio_getWirelessChannelStrengths	Gets strength of the wireless channels from device. (Base Station only)	210(0xd2)
prio_setWirelessRetries	Sets wireless retries to device. (Base Station only)	211(0xd3)
prio_getWirelessRetries	Gets wireless retries from device. (Base Station only)	212(0xd4)
prio_getWirelessSlotsOpen	Gets open wireless slots from device. (Base Station only)	213(0xd5)
prio_getSignalStrength	Gets wireless signal strength from device. (Base Station only)	214(0xd6)
prio_setTimestampEnabled	Sets timestamps enabled for device. (Unavailable)	219(0xdb)
prio_getFirmwareVersionString	Gets firmware version string from device at the logical ID.	223(0xdf)
prio_restoreFactorySettings	Restores factory settings to device at the logical ID.	224(0xe0)
prio_commitSettings	Commits changed settings to device at the logical ID.	225(0xe1)
prio_resetProcessor	Resets processor of device at the logical ID.	226(0xe2)
prio_enterBootloaderMode	Puts device in bootloader mode.	229(0xe5)
prio_getHardwareVersionString	Gets hardware version string from device at the logical ID.	230(0xe6)

Name	Description	Command
prio_selfTest	Runs self test on the device. (Base Station only)	235(0xeb)
prio_getSerialNumber	Gets serial number from device at the logical ID.	237(0xed)
prio_setLedColor	Sets LED color to device. (Base Station only)	238(0xee)
prio_setLedColorSuit	Sets LED color to device at the logical ID. (Unavailable for Base Station)	238(0xef)
prio_getLedColor	Gets LED color from device. (Base Station only)	239(0xef)
prio_getButtonState	Gets button state from paired Hub.	250(0xfa)

Serial Number: ____

Notes:

YEI Technology
630 Second Street
Portsmouth, Ohio 45662

Toll-Free: 888-395-9029
Phone: 740-355-9029

www.YeiTechnology.com
www.PrioVR.com

©2007-2014 YEI Corporation
Printed in USA