YEI 3-Space Python API

API Documentation

May 14, 2012

Contents

1 Module threespace_api

Authors: "Sam Bushman" <sbushman@yostengineering.com>, "Chris George" <cgeorge@yostengineering.com>, "Dan Morrison" <dmorrison@yostengineering.com>

1.1 Classes

- **AsynchronousBroadcaster**: The AsynchronousBroadcaster class allows the programmer to easily enable asynchronous commands for a cluster of sensors all at once.
(Section 2, p. 5)
- **BroadcastSensor**: The BroadcastSensor class is a virtual sensor of sorts that polls all currently connected wired sensors for data.
(Section 4, p. 32)
- **BroadcastConnection**: The BroadcastConnection class is a thread that executes the BroadcastSensor's assigned command to an assigned sensor and returnes the data from the sensor to the BroadcastSensor.
(Section 3, p. 30)
- **TSBootloader**: The TSBootloader class is an interface layer for 3-Space Sensor units that communicate through the USB port and are in bootloader mode.
(Section 6, p. 51)
- **TSSensor**: The TSSensor class is an interface layer for 3-Space Sensor USB units that communicate through the USB port.
(Section 10, p. 84)
- **TSDongle**: The TSDongle class is an interface layer for 3-Space Sensor Dongle units that communicate through the USB port and acts as an autonomous object that handles communication between the dongle and wireless 3-Space Sensor units.
(Section 8, p. 62)
- **TSWLSensor**: The TSWLSensor class is an interface layer for 3-Space Sensor Wireless units that communicate through the USB port or a 3-Space Sensor Dongle.
(Section 11, p. 133)
- **TSDLSensor**: The TSDLSensor class is an interface layer for 3-Space Sensor Data-logging units that communicate through the USB port.
(Section 7, p. 57)
- **TSBTSensor**: The TSBTSensor class is an interface layer for 3-Space Sensor Bluetooth units that communicate through the USB port.
(Section 5, p. 46)
- **TSEMSensor**: The TSEMSensor class is an interface layer for 3-Space Sensor Embedded units that communicate through the USB port or RS-232 port.
(Section 9, p. 80)

1.2 Functions

getLastError()

Gets the last known error that the API encountered.

Return Value

A tuple describing the error type, 3-Space Sensor device that the error occured on, 3-Space Sensor device's serial number, the data that caused the error, the data's length, and a message of the error if available.

1.3 Variables

Name	Description
global_wired_broadcaster	A global object meant for communicating with all USB/RS232 connected sensors at once. See `BroadcastSensor` for more information on usage. **Value:** `BroadcastSensor()`
global_asynch_broadcaster	A global object meant for communicating with groups of known/connected sensors at once. See `AsynchronousBroadcaster` for more information on usage. **Value:** `AsynchronousBroadcaster()`
EIGHTBITS	**Value:** `8`
FIVEBITS	**Value:** `5`
PARITY_EVEN	**Value:** `'E'`
PARITY_MARK	**Value:** `'M'`
PARITY_NONE	**Value:** `'N'`
PARITY_ODD	**Value:** `'O'`
PARITY_SPACE	**Value:** `'S'`
SEVENBITS	**Value:** `7`
SIXBITS	**Value:** `6`
STOPBITS_ONE	**Value:** `1`
STOPBITS_ONE_POINT_FIVE	**Value:** `1.5`
STOPBITS_TWO	**Value:** `2`
__package__	**Value:** `None`

2 Class threespace_api.AsynchronousBroadcaster

object
└ **threespace_api.AsynchronousBroadcaster**

The AsynchronousBroadcaster class allows the programmer to easily enable asynchronous commands for a cluster of sensors all at once. The AsynchronousBroadcaster can start an asynchronous communication session, poll for updated data, and stop communication with a couple function calls. Data is read from wired and wireless sensors with this class. While wireless sensors are simply put into an "asynchronous communication mode", wired sensors are polled in real time when polling of the broadcaster takes place in order to emulate the asynchronous behavior of the wireless sensors. All data returned is timestamped based on a common start time to ensure easy interpolation. A single global instance of this class called global_asynch_broadcaster is interacted with in order to perform broadcasting actions.

Attributes:

- **dongle_list**: A list of references to TSDongle objects. This list is used for polling for data and enabling, and disabling asynchronous communication.
- **poll_func**: A function reference used for determining what wired broadcast function to call when data polling takes place. This variable is set when an asynchronous session is started.
- **start_time**: A float that represents the time to base all timestamps on. This variable is -1 when no broadcast session is taking place.
- **wired_do_remove**: A boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy.
- **is_broadcasting**: A boolean that indicates the current status of the broadcaster. If True, the broadcaster is currently in an active asynchronous session. If False, the broadcaster is not actively communicating with any 3-Space Sensor devices.
- **sensor_filter**: A list of 3-Space Sensor device types that the global_asynch_broadcaster is to be only communicating with. If None all 3-Space Sensor device types are communicated with.

2.1 Methods

__init__(*self*)

Initializes an AsynchronousBroadcaster object.

Overrides: object.__init__

addToList(*self*, *dngl*)

Adds a passed in TSDongle reference to the internal list of dongles.

Parameters

`dngl`: A TSDongle reference. This reference is added to the internal list.

Return Value

None

delFromList(*self*, *dngl*)

Removes a TSDongle reference from the internal list of dongles based on the passed in reference.

Parameters

`dngl`: A TSDongle reference. This reference is used to find (and remove) a TSDongle reference from the internal list. If the reference is not in the list, the function returns performing no action.

Return Value

None

poll(*self*)

Returns a map of a the most recient data collected from each sensor.

Return Value

A map of data. The keys in the map are the serial numbers for each sensor transmitting data. The values of the map is a tuple. The first value of the tuple is the transmitted data, while the second is a timestamp of when the data was recieved. *data_map[sensor_serial] = (transmitted data, timestamp)* None is returned if no asynchronous broadcasting is taking place.

stop(*self*, *num_retries*=`3`, *callback_func*=`None`, *try_again*=`False`)

Stops all asynchronous transmissions.

Parameters

`num_retries`: An optional integer indicating the number of times to retry stopping the asynchronous session for a given device if the previous attempt was unsuccessful. The default value is 3.

`callback_func`: An optional function reference that is used for checking if a 3-Space Sensor stopped an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again`: An optional boolean that is used for indicating a retry of stopping the asynchronous session for the 3-Space Sensors still in an asynchronous session. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in stopping their asynchronous session.

beginFiltTaredOrientQuat(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Tared Orientation of each sensor as a Quaternion (x, y, z, w).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated quaternion to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltTaredOrientEuler(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Tared Orientation of each sensor as a set of Euler Angles (in radians) (ordered as pitch, yaw, roll).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated set of Euler Angles to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltTaredOrientMat(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Tared Orientation of each sensor as a Matrix (row major).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated matrix to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltTaredOrientAxisAngle(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Tared Orientation of each sensor as an Axis Angle construct (x, y, z, angle_in_radians).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated Axis Angle to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltTaredOrientFwdDwn(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Tared Orientation of each sensor as a pair of vectors (one pointing 'forward' and the other pointing 'down').

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated Axis Angle to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltOrientQuat(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Un-Tared Orientation of each sensor as a Quaternion (x, y, z, w).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated quaternion to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltOrientEuler(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Un-Tared Orientation of each sensor as a set of Euler Angles (in radians) (ordered as pitch, yaw, roll).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated set of Euler Angles to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltOrientMat(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Un-Tared Orientation of each sensor as a Matrix (row major).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated matrix to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltOrientAxisAngle(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Un-Tared Orientation of each sensor as an Axis Angle construct (x, y, z, angle_in_radians).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated Axis Angle to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltOrientFwdDwn(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the Filtered Un-Tared Orientation of each sensor as a pair of vectors (one Vector3 pointing 'forward' and the other Vector3 pointing 'down').

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated Forward and Down vectors to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltTaredFwdDwnVecSensFrame(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the forward and down vectors as determined by the Kalman filter, relative to the tare orientation. This version returns the data as the forward and down vectors of the coordinate system defined by the sensor reference frame (Forward Vector3, Down Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated Forward and Down vectors to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginFiltNorthEarthVecSensFrame(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the North and Earth vectors as determeined by the Kalman filter, relative to the global sensor references. Returned vectors are in the coordinate system defined by the sensor reference frame (North Vector3, Earth Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated North and Earth vectors to the system.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginAllSensorsNormalized(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the normalized data collected from each sensor on the 3-Space Sensor (gyro Vector 3, accelerometer Vector3, compass Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated array of normalized sensor data.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginGyroNormalized(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the normalized data collected from the gyroscope on the 3-Space Sensor (gyro Vector 3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send updated gyroscope data.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginAccelerometerNormalized(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the normalized data collected from the accelerometer on the 3-Space Sensor (accelerometer Vector3).

Parameters

`interval:`	A length of time in milliseconds that each wireless sensor will wait before attempting to send updated accelerometer data.
`filter:`	A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.
`num_retries:`	An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.
`do_remove:`	An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.
`callback_func:`	An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.
`try_again:`	An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginCompassNormalized(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the normalized data collected from the compass on the 3-Space Sensor (compass Vector3).

Parameters

`interval:`	A length of time in milliseconds that each wireless sensor will wait before attempting to send updated compass data.
`filter:`	A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.
`num_retries:`	An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.
`do_remove:`	An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.
`callback_func:`	An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.
`try_again:`	An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginAccelerometerUnfiltered(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the unfiltered (but calibrated) data collected from the accelerometer on the 3-Space Sensor (accelerometer Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send updated accelerometer data.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginCompassUnfiltered(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the unfiltered (but calibrated) data collected from the compass on the 3-Space Sensor (compass Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated array of compass data.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginAllSensorsRaw(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the raw data collected from each sensor on the 3-Space Sensor (gyro Vector 3, accelerometer Vector3, compass Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send an updated array of raw sensor data.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginGyroRaw(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the raw data collected from the 3-Space Sensor's gyroscope. (gyro Vector 3).

Parameters

`interval:`	A length of time in milliseconds that each wireless sensor will wait before attempting to send updated gyroscope data.
`filter:`	A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.
`num_retries:`	An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.
`do_remove:`	An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.
`callback_func:`	An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.
`try_again:`	An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginAccelerometerRaw(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the raw data collected from the 3-Space Sensor's accelerometer. (accelerometer Vector3).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send updated accelerometer data.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginCompassRaw(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the raw data collected from the 3-Space Sensor's compass (compass Vector3).

Parameters

`interval:`	A length of time in milliseconds that each wireless sensor will wait before attempting to send updated compass data.
`filter:`	A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.
`num_retries:`	An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.
`do_remove:`	An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.
`callback_func:`	An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.
`try_again:`	An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

beginButtonState(*self*, *interval*, *filter*=`None`, *num_retries*=`3`, *do_remove*=`False`, *callback_func*=`None`, *try_again*=`False`)

Begins an asynchronous session. This session will be tranmitting the current state of the sensor's physical buttons. Designation of 'left' and 'right' assumes the sensor to be oriented such that the LED side of the sensor is facing up and the side of the sensor that contains the USB port is facing towards the user (is_left_pressed Boolean, is_right_pressed Boolean).

Parameters

`interval:` A length of time in milliseconds that each wireless sensor will wait before attempting to send updated button states.

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors connected to the host computer are used. The default value is None.

`num_retries:` An optional integer indicating the number of times to retry starting the asynchronous start command to a given device if the previous attempt was unsuccessful. The default value is 3.

`do_remove:` An optional boolean that is passed as the 'do_remove' argument for all calls made to the wired broadcaster. When True, wired sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

`callback_func:` An optional function reference that is used for checking if a 3-Space Sensor started an asynchronous session. The function must have at least two parameters (first parameter for a TSSensor object, second parameter for a boolean). The default value is None.

`try_again:` An optional boolean that is used for indicating a retry of starting asynchronous for the 3-Space Sensors in 'filter'. The default value is False.

Return Value

A list of 3-Space Sensors (listed by TSSensor objects) who were unsuccessful in starting their asynchronous session.

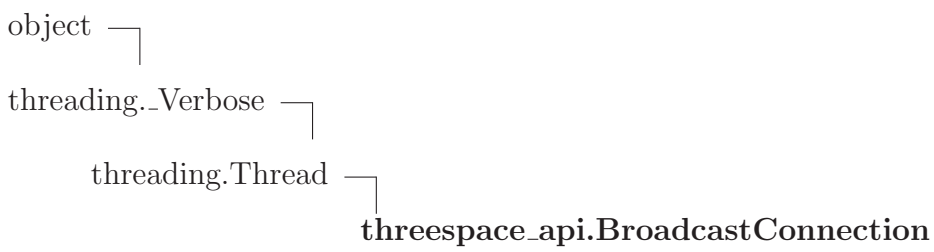
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

2.2 Properties

Name	Description
Inherited from object	
__class__	

3 Class threespace_api.BroadcastConnection



The BroadcastConnection class is a thread that executes the BroadcastSensor's assigned command to an assigned sensor and returnes the data from the sensor to the BroadcastSensor.

Attributes:

- **output**: A storage bin for the return data from the assigned command. The BroadcastSensor reads data from this variable once the command has been completed.
- **func**: A function reference which identifies the function to execute once the connection's thread is started.
- **param**: A storage bin for any single parameter that must be passed to the function once execution begins.

3.1 Methods

__init__(*self*, *func*, *param*=`None`)

Initializes a BroadcastConnection object.

Parameters

`func:` A reference to the function to be executed once the connection's thread is started.

`param:` An optional argument that holds any single parameter that must be passed to func when it is called. This argument has a default value of None.

Overrides: object.__init__

run(*self*)

The execution thread that starts when BroadcastConnection.start() is called.

Overrides: threading.Thread.run

Inherited from threading.Thread

__repr__(), getName(), isAlive(), isDaemon(), is_alive(), join(), setDaemon(), set-

Name(), start()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

3.2 Properties

Name	Description
Inherited from threading.Thread daemon, ident, name	
Inherited from object __class__	

4 Class threespace_api.BroadcastSensor

object
└ **threespace_api.BroadcastSensor**

The BroadcastSensor class is a virtual sensor of sorts that polls all currently connected wired sensors for data. Returned data is assumed to be current as of the time the call was made. The BroadcastSensor is accessed via a global variable instance called global_wired_broadcaster.

Attribute: sensor_list: A map of references to 3-Space Sensors connected via USB (listed by serial number). All sensors in this map may be polled for data when the BroadcastSensor's functions are called. *sensor_list[serial_number] = sensor*

4.1 Methods

__init__(*self*)

Initializes a BroadcastSensor object.

Overrides: object.__init__

addToList(*self*, *sensor*)

Adds a passed in TSSensor reference to the internal list of sensors.

Parameters

`sensor`: A TSSensor reference. This reference is added to the internal list.

delFromList(*self*, *sensor_serial_number*)

Removes a TSSensor reference from the internal list of sensors based on the passed in reference.

Parameters

`sensor_serial_number`: A 3-Space sensor serial number. This serial number is used to find (and remove) a TSSensor reference from the internal list. If the serial number does not match a sensor in the list, the function returns performing no action.

getFiltTaredOrientQuat(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered tared orientation as a quaternion (x, y, z, w).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltTaredOrientEuler(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered tared orientation as a set of euler angles (in radians) ordered as such: (pitch, yaw, roll).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltTaredOrientMat(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered tared orientation as a matrix (row major).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltTaredOrientAxisAngle(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered tared orientation as an Axis Angle construct (x, y, z, angle_in_radians).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltTaredOrientFwdDwn(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current Filtered Tared Orientation of each sensor as a pair of vectors (one pointing 'forward' and the other pointing 'down').

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltOrientQuat(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered un-tared orientation as a quaternion (x, y, z, w).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltOrientEuler(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered un-tared orientation as a set of euler angles (in radians) ordered as such: (pitch, yaw, roll).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltOrientMat(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered un-tared orientation as a matrix (row major).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltOrientAxisAngle(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current filtered un-tared orientation as an Axis Angle construct (x, y, z, angle_in_radians).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltOrientFwdDwn(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their current Filtered Un-Tared Orientation of each sensor as a pair of vectors (one pointing 'forward' and the other pointing 'down').

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltTaredFwdDwnVecSensFrame(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their forward and down vectors as determined by the Kalman filter, relative to the tare orientation. This version returns the data as the forward and down vectors of the coordinate system defined by the sensor reference frame (Forward Vector3, Down Vector3).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getFiltNorthEarthVecSensFrame(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for their North and Earth vectors as determeined by the Kalman filter, relative to the global sensor references. Returned vectors are in the coordinate system defined by the sensor reference frame (North Vector3, Earth Vector3).

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getAllSensorsNormalized(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the normalized data of their on-board sensors. The data is stored in a tuple with nine entries. The first three elements are the normalized data from the sensor's gyro, the second three elements are the normalized data from the sensor's accelerometer, and the last three elements are the normalized data from the sensor's compass. (gyro Vector 3, accelerometer Vector3, compass Vector3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getGyroNormalized(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the normalized data of their on-board sensors. The data is stored in a tuple with three entries representing the normalized data from the sensor's gyro. (gyro Vector 3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getAccelerometerNormalized(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the normalized data of their on-board sensors. The data is stored in a tuple with three entries representing the normalized data from the sensor's accelerometer. (accelerometer Vector3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getCompassNormalized(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the normalized data of their on-board sensors. The data is stored in a tuple with three entries representing the normalized data from the sensor's compass. (compass Vector3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getAccelerometerUnfiltered(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the unfiltered (but compensated) data of their on-board accelerometer. The data is stored in a tuple with 3 entries representing the unfiltered data from the sensor's accelerometer. (accelerometer Vector3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getCompassUnfiltered(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the unfiltered (but compensated) data of their on-board compass. The data is stored in a tuple with 3 entries representing the unfiltered data from the sensor's compass. (compass Vector3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getAllSensorsRaw(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the raw data of their on-board sensors. The data is stored in a tuple with nine entries. The first three elements are the raw data from the sensor's gyro, the second three elements are the raw data from the sensor's accelerometer, and the last three elements are the raw data from the sensor's compass. (gyro Vector 3, accelerometer Vector3, compass Vector3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getGyroRaw(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the raw data of their on-board gyroscopes. The data is stored in a tuple with three entries representing the raw data from the sensor's gyroscope. (gyro Vector 3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getAccelerometerRaw(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the raw data of their on-board accelerometers. The data is stored in a tuple with three entries representing the raw data from the sensor's accelerometer. (accelerometer Vector 3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

getCompassRaw(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the raw data of their on-board compasses. The data is stored in a tuple with three entries representing the raw data from the sensor's compass. (compass Vector 3)

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

setTareCurrentOrient(*self*, *filter*=`None`, *do_remove*=`False`)

Tares all wired sensors in 'filter' with their current orientation.

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

setLEDColor(*self*, *color*, *filter*=`None`, *do_remove*=`False`)

Sets all wired sensors' LED color in 'filter' to the passed in color value.

Parameters

`color:` A tuple of 3 floats. The floats range in value from 0.0 to 1.0. The first value of the tuple is the red channel, the second is the green channel, and the third is the blue channel. *color_tuple = (red, green, blue)*

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

getButtonState(*self*, *filter*=`None`, *do_remove*=`False`)

Polls all known wired 3-Space Sensors for the current state of their physical buttons. The button state is returned as a tuple of bools. The first tuple value references the left button and the second references the right button. A value of True indicates the button is pressed, and a value of False indicates the button is not pressed.

Parameters

`filter:` A list of 3-Space Sensors that will be used to start this asynchronous function (listed by TSSensor objects). If None all 3-Space Sensors in the map 'sensor_list' are used. The default value is None.

`do_remove:` An optional boolean that when True, sensors that are disconnected are automatically removed from the broadcaster's list of sensors. If removed, these sensors must be added back to the broadcaster manualy. The default value is False.

Return Value

A map of values are returned. The keys to the map are the serial numbers of all sensors that are known. The values of the map are the requested data. *data_map[serial_number] = data*

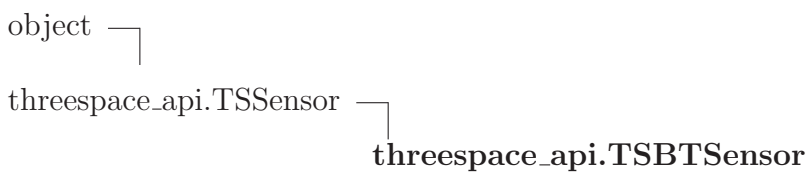
Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __new__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

4.2 Properties

Name	Description
Inherited from object	
__class__	

5 Class threespace_api.TSBTSensor



The TSBTSensor class is an interface layer for 3-Space Sensor Bluetooth units that communicate through the USB port. Functions available through this class are either pythonic versions of the sensor's available commands or convenience functions for handling the establishment and use of the class's serial port.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **device_type**: A string expressing the type of 3-Space Sensor associated with the class instance.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the sensor takes place on.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before the next command is sent.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.

5.1 Methods

setLEDMode(*self*, *mode*)

Sets the mode of the sensor's LED (if wireless). The LED has two possible mode, 'static' and standard.

- If the LED is in 'static' mode, this means that it will only display the color set by the command setLEDColor.
- If the LED is in 'standard' mode, it will display the standard LED colors as described below:
  - Upon receipt of a packet, the wireless unit will flash green temporarily. This will occur regardless of whether the wireless unit is plugged in or not.
  - When the wireless unit is plugged in and charging, the sensor will flash orange every second.
  - When the wireless unit is plugged in and fully charged, the sensor will flash green every second.
  - When the wireless unit falls below a certain battery life level, it will flash red in increasingly quicker intervals. *(Note that this does not happen if the sensor is plugged in.)*

Parameters

`mode`: An integer of two valid values. If 0, the sensor will be set to 'standard' LED mode. If 1, the sensor will be set to 'static' LED mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getLEDMode(*self*)

Reads the mode of the sensor's LED (if wireless). The LED has two possible mode, 'static' and standard.

- If the LED is in 'static' mode, this means that it will only display the color set by the command setLEDColor.
- If the LED is in 'standard' mode, it will display the standard LED colors as described below:
  - Upon receipt of a packet, the wireless unit will flash green temporarily. This will occur regardless of whether the wireless unit is plugged in or not.
  - When the wireless unit is plugged in and charging, the sensor will flash orange every second.
  - When the wireless unit is plugged in and fully charged, the sensor will flash green every second.
  - When the wireless unit falls below a certain battery life level, it will flash red in increasingly quicker intervals. *(Note that this does not happen if the sensor is plugged in.)*

Return Value

An integer of two valid values. If 0, the sensor will be set to 'standard' LED mode. If 1, the sensor will be set to 'static' LED mode.

getBatteryVoltage(*self*)

Return the voltage of the battery as measured by the sensor.

Return Value

A float whose value is the current voltage of the battery.

getBatteryLife(*self*)

Return the percentage of battery life left in the sensor's battery.

Return Value

An integer whose value is the current percentage of the battery's life.

getBatteryStatus(*self*)

Return a status number indicating the state of the sensor's battery.

Return Value

An integer whose value is the current status of the battery. 1 represents fully charged, 2 represents charging.

setUARTRate(*self*, *rate*)

A placer function since the parent class may perform this operation, but this class cannot. Will just return False.

Parameters

`rate`: An integer whose value is the desired baud rate of the UART.

Return Value

False

Overrides: threespace_api.TSSensor.setUARTRate

getUARTRate(*self*)

A placer function since the parent class may perform this operation, but this class cannot. Will just return None.

Return Value

None

Overrides: threespace_api.TSSensor.getUARTRate

Inherited from threespace_api.TSSensor(Section 10)

__init__(), __new__(), calibrateGyro(), close(), commitSettings(), disableAxis(), disableButton(), disableWatchdogTimer(), enableWatchdogTimer(), enterFirmwareUpdateMode(), getAccelerometerCalibrationParam(), getAccelerometerEnabled(), getAccelerometerNormalized(), getAccelerometerRange(), getAccelerometerRaw(), getAccelerometerUnfiltered(), getActualUpdateRate(), getAllSensorsNormalized(), getAllSensorsRaw(), getAvgPercent(), getAxisDirections(), getButtonGyroDisableLength(), getButtonState(), getClockSpeed(), getCompassCalibrationParam(), getCompassEnabled(), getCompassNormalized(), getCompassRange(), getCompassRaw(), getCompassUnfiltered(), getConfidence(), getControlData(), getControlMode(), getDesiredUpdateRate(), getFiltGyroRate(), getFiltNorthEarthVecSensFrame(), getFiltOrientAxisAngle(), getFiltOrientEuler(), getFiltOrientFwdDwn(), getFiltOrientMat(), getFiltOrientQuat(), getFiltTaredFwdDwnVecSensFrame(), getFiltTaredOrientAxisAngle(), getFiltTaredOrientEuler(), getFiltTaredOrientFwdDwn(), getFiltTaredOrientMat(), getFiltTaredOrientQuat(), getFilterMode(), getGyroCalibrationParam(), getGyroEnabled(), getGyroNormalized(), getGyroRaw(), getGyroscopeRange(), getJoystickEnabled(), getJoystickMousePresent(), getKalmanMat(), getLEDColor(), getLastError(), getLookupTblVertVal(), getMouseEnabled(), getMouseRelative(), getMultiRefChkVecAccelerometer(), getMultiRefChkVecCompass(), getMultiRefResolution(), getMultiRefVecAccelerometer(), getMultiRefVecCompass(), getMultiRefWeightPwr(), getNumMultiRefCells(), getOversampleRate(), getRefVecAccelerometer(), getRefVecCompass(), getRefVecMode(), getRhoDataAccelerometer(), getRhoDataCompass(), getRunningAverageMode(), getSerialNumber(), getTareOrientMat(), getTareOrientQuat(), getTemperatureCelsius(), getTemperatureFahrenheit(), getUSBMode(), isConnected(), read(), reconnect(), resetKalmanFilter(),

restoreFactorySettings(), setAccelerometerCalibrationParam(), setAccelerometerEnabled(), setAccelerometerRange(), setAvgPercent(), setAxisDirections(), setButtonGyroDisableLength(), setClockSpeed(), setCompassCalibrationParam(), setCompassEnabled(), setCompassRange(), setConfidenceRhoModeAccel(), setConfidenceRhoModeCompass(), setControlData(), setControlMode(), setFilterMode(), setGlobalAxis(), setGyroCalibrationParam(), setGyroEnabled(), setGyroscopeRange(), setJoystickEnabled(), setJoystickMousePresent(), setLEDColor(), setLookupTblVertVal(), setMouseEnabled(), setMouseRelative(), setMultiRefChkVecAccelerometer(), setMultiRefChkVecCompass(), setMultiRefResolution(), setMultiRefVecAccelerometer(), setMultiRefVecCompass(), setMultiRefVecZero(), setMultiRefWeightPwr(), setOrientationButton(), setOversampleRate(), setPhysicalButton(), setRefVecAccelerometer(), setRefVecCompass(), setRefVecCurrentOrient(), setRefVecMode(), setRunningAverageMode(), setScreenPointAxis(), setShakeButton(), setStaticRhoModeAccel(), setStaticRhoModeCompass(), setTareCurrentOrient(), setTareMatrix(), setTareQuaternion(), setUSBMode(), setUpdateRate(), setupSimpleJoystick(), setupSimpleLightgun(), setupSimpleMouse(), softwareReset(), write()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

5.2 Properties

Name	Description
Inherited from object	
__class__	

6 Class threespace_api.TSBootloader

object
└ **threespace_api.TSBootloader**

The TSBootloader class is an interface layer for 3-Space Sensor units that communicate through the USB port and are in bootloader mode. Functions available through this class are either pythonic versions of the sensor's available commands or convenience functions for handling the establishment and use of the class's serial port.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **device_type**: A string expressing the type of 3-Space Sensor associated with the class instance. "WL" indicates a wireless sensor and "USB" indicates a wired sensor.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the sensor takes place on.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before the next command is sent.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.
- **page_size**: An integer that denotes the byte size to be written to the 3-Space Sensor.
- **error**: A TSSError instance that keeps track of the last known error the class encountered.

6.1 Methods

__new__(*cls*, *com_port*, *read_timeout*=2, *write_timeout*=2, *baudrate*=115200)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__ extit(inherited documentation)

__init__(*self*, *com_port*, *read_timeout*=2, *write_timeout*=2, *baudrate*=115200)

Initializes the TSBootloader class.

Parameters

`com_port:` A string storing the name of the serial port intended for communication. The name of the port is dependent of the operating system.

`read_timeout:` An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up. The default value is 2.0.

`write_timeout:` An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up. The default value is 2.0.

`baudrate:` An optional integer argument expressing the data transmission rate of the serial port. The default value is 115200 bits per second.

Overrides: object.__init__

close(*self*)

Closes the device's serial port, preventing any future communication.

read(*self*, *size*)

Reads a specificed number of bytes from the device's serial port.

Parameters

`size:` An integer expressing the number of bytes to attempt to read from the device's serial port.

Return Value

A binary string representing the bytes read from the serial port. If an error occurs, an empty string is returned.

write(*self*, *data*)

Writes a binary string of data to the sensor's serial port.

Parameters

`data`: A binary string of data.

Return Value

True is returned if the write was successful. False is returned if some sort of error occured. When a write error occurs, the sensor's serial port is also closed and set to None.

isConnected(*self*, *try_reconnect*=`False`)

Tests if the device's serial port if properly connected and functioning.

Parameters

`try_reconnect`: An optional boolean that when True, an attempt at reconnection is made if the device is not connected. The default value is False.

Return Value

True is returned if the serial port is connected and working properly. False is returned if the port is not connected or has otherwise failed.

reconnect(*self*)

Attempts to establish a functioning connection to the device on the serial port.

Return Value

True is returned if the reconnection was successful. False is returned if the serial port could not be reconnnected.

getLastError(*self*)

Gets the last known error that the 3-Space Sensor unit encountered.

Return Value

A tuple describing the error type, 3-Space Sensor device that the error occured on, 3-Space Sensor device's serial number, the data that caused the error, the data's length, an exception and a message of the error if available.

isInBootloaderMode(*self*)

Checks if the 3-Space Sensor is in fact in bootloader mode.

Return Value

True is returned if the sensor is in bootloader. False is returned if the sensor is not in bootloader.

writeUnencryptedData(*self*, *data*)

Writes unencrypted data to the 3-Space Sensor.

Parameters

`data`: A byte string that is unencrypted.

Return Value

True if the data was successfuly written to the device. False if the data was not written.

setToFirmwareMode(*self*)

Resets the device to use firmware loaded to the TSBootloader. Will remain in bootloader if firmware was corrupted.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

writeEncryptedData(*self*, *data*)

Writes encrypted data to the 3-Space Sensor.

Parameters

`data`: A byte string that is encrypted.

Return Value

True if the data was successfuly written to the device. False if the data was not written.

finishWrite(*self*)

Tells the TSBootloader to stop writting.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getInformation(*self*)

Gets information about the TSBootloader device.

Return Value

A map of properties of the TSBootloader or None if failed to retrieve the data. The keys to the map are:

- 'J-Tag_ID'
- 'Program_Offset'
- 'Flash_Size'
- 'BTL_Page_Size'
- 'User_Page_Offset'
- 'User_Page_Size'
- 'App_Firmware_Version'
- 'BTL_Firmware_Version'

setSerialNumber(*self*, *serial_number*)

Sets the serial number for the 3-Space Sensor device.

Parameters

`serial_number:` An integer that denotes the unique serial number for the sensor.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

isFirmwareValid(*self*)

Checks the firmware loaded into the 3-Space Sensor is valid or not.

Return Value

True is returned if the firmware is valid. False is returned if the firmware is invalid or corrupted.

setWriteStart(*self*, *start_addr*)

Sets the memory address for writting firmware to the TSBootloader.

Parameters

`start_addr:` An integer that denotes the start position for writting the firmware.

getSerialNumber(*self*)

Reads the serial number of the sensor.

Return Value

An integer whose value is the serial number of the sensor or None if failed to receive the data.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

6.2 Properties

Name	Description
Inherited from object	
__class__	

7 Class threespace_api.TSDLSensor

The TSDLSensor class is an interface layer for 3-Space Sensor Data-logging units that communicate through the USB port. Functions available through this class are either pythonic versions of the sensor's available commands or convenience functions for handling the establishment and use of the class's serial port.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **device_type**: A string expressing the type of 3-Space Sensor associated with the class instance.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the sensor takes place on.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before the next command is sent.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.
- **error**: A TSSError instance that keeps track of the last known error the class encountered.

7.1 Methods

enableMassStorageMode(*self*)

When enabled, the 3-Space Sensor acts as a mass storage device on the host system. This is useful for reading/writing log files to the sensor.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

disableMassStorageMode(*self*)

When disabled, the 3-Space Sensor's mass storage features are hidden from the host machine.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

formatInitSDCard(*self*)

The SD card contained within the 3-Space Sensor is formatted to the FAT-32 file system and initialized for writing/reading.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

startLogSession(*self*)

Begins a data logging session based on the contents of the capture.cfg file.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

stopLogSession(*self*)

Stops any active capture sessions.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

setClock(*self*, *month*, *day*, *year*, *hour*, *minute*, *second*)

Sets the current data and time of the 3-Space Sensor's internal clock to the inputted date and time.

Parameters

`month:` An integer whose value can range from 0 - 19. This value represents the month of the date to be set.

`day:` An integer whose value can range from 0 - 39. This value represents the day of the date to be set.

`year:` An integer whose value can range from 0 - 159. This value represents the year of the date to be set.

`hour:` An integer whose value can range from 0 - 39. This value represents the hour of the time to be set.

`minute:` An integer whose value can range from 0 - 79. This value represents the minute of the time to be set.

`second:` An integer whose value can range from 0 - 79. This value represents the second of the time to be set.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

getClock(*self*)

Gets the current data and time of the 3-Space Sensor's internal clock.

Return Value

A list of integers expressing the current date and time of the clock on the sensor. The list is formatted as such: *[month, day, year, hour, minute, second]*

getBatteryVoltage(*self*)

Return the voltage of the battery as measured by the sensor.

Return Value

A float whose value is the current voltage of the battery.

getBatteryLife(*self*)

Return the percentage of battery life left in the sensor's battery.

Return Value

An integer whose value is the current percentage of the battery's life.

getBatteryStatus(*self*)

Return a status number indicating the state of the sensor's battery.

Return Value

An integer whose value is the current status of the battery. 1 represents fully charged, 2 represents charging.

setUARTRate(*self*, *rate*)

A placer function since the parent class may perform this operation, but this class cannot. Will just return False.

Parameters

`rate`: An integer whose value is the desired baud rate of the UART.

Return Value

False

Overrides: threespace_api.TSSensor.setUARTRate

getUARTRate(*self*)

A placer function since the parent class may perform this operation, but this class cannot. Will just return None.

Return Value

None

Overrides: threespace_api.TSSensor.getUARTRate

Inherited from threespace_api.TSSensor(Section 10)

__init__(), __new__(), calibrateGyro(), close(), commitSettings(), disableAxis(), disableButton(), disableWatchdogTimer(), enableWatchdogTimer(), enterFirmwareUpdateMode(), getAccelerometerCalibrationParam(), getAccelerometerEnabled(), getAccelerometerNormalized(), getAccelerometerRange(), getAccelerometerRaw(), getAccelerometerUnfiltered(), getActualUpdateRate(), getAllSensorsNormalized(), getAllSensorsRaw(), getAvgPercent(), getAxisDirections(), getButtonGyroDisableLength(), getButtonState(), getClockSpeed(), getCompassCalibrationParam(), getCompassEnabled(), getCompassNormalized(), getCompassRange(), getCompassRaw(), getCompassUnfiltered(), getConfidence(), getControlData(), getControlMode(), getDesiredUpdateRate(), getFiltGyroRate(), getFiltNorthEarthVecSensFrame(), getFiltOrientAxisAngle(), getFiltOrientEuler(), getFiltOrientFwdDwn(), getFiltOrientMat(), getFiltOrientQuat(), getFiltTaredFwdDwnVecSensFrame(), getFiltTaredOrientAxisAngle(), getFiltTaredOrientEuler(), getFiltTaredOrientFwdDwn(), getFiltTaredOrientMat(), getFiltTaredOrientQuat(), getFilterMode(), getGyroCalibrationParam(), getGyroEnabled(), getGyroNormalized(), getGyroRaw(), getGyroscopeRange(), getJoystickEnabled(), getJoystickMousePresent(), getKalmanMat(), getLEDColor(), getLastError(), getLookupTblVertVal(), getMouseEnabled(), get-

MouseRelative(), getMultiRefChkVecAccelerometer(), getMultiRefChkVecCompass(), getMultiRefResolution(), getMultiRefVecAccelerometer(), getMultiRefVecCompass(), getMultiRefWeightPwr(), getNumMultiRefCells(), getOversampleRate(), getRefVecAccelerometer(), getRefVecCompass(), getRefVecMode(), getRhoDataAccelerometer(), getRhoDataCompass(), getRunningAverageMode(), getSerialNumber(), getTareOrientMat(), getTareOrientQuat(), getTemperatureCelsius(), getTemperatureFahrenheit(), getUSBMode(), isConnected(), read(), reconnect(), resetKalmanFilter(), restoreFactorySettings(), setAccelerometerCalibrationParam(), setAccelerometerEnabled(), setAccelerometerRange(), setAvgPercent(), setAxisDirections(), setButtonGyroDisableLength(), setClockSpeed(), setCompassCalibrationParam(), setCompassEnabled(), setCompassRange(), setConfidenceRhoModeAccel(), setConfidenceRhoModeCompass(), setControlData(), setControlMode(), setFilterMode(), setGlobalAxis(), setGyroCalibrationParam(), setGyroEnabled(), setGyroscopeRange(), setJoystickEnabled(), setJoystickMousePresent(), setLEDColor(), setLookupTblVertVal(), setMouseEnabled(), setMouseRelative(), setMultiRefChkVecAccelerometer(), setMultiRefChkVecCompass(), setMultiRefResolution(), setMultiRefVecAccelerometer(), setMultiRefVecCompass(), setMultiRefVecZero(), setMultiRefWeightPwr(), setOrientationButton(), setOversampleRate(), setPhysicalButton(), setRefVecAccelerometer(), setRefVecCompass(), setRefVecCurrentOrient(), setRefVecMode(), setRunningAverageMode(), setScreenPointAxis(), setShakeButton(), setStaticRhoModeAccel(), setStaticRhoModeCompass(), setTareCurrentOrient(), setTareMatrix(), setTareQuaternion(), setUSBMode(), setUpdateRate(), setupSimpleJoystick(), setupSimpleLightgun(), setupSimpleMouse(), softwareReset(), write()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

7.2 Properties

Name	Description
Inherited from object	
__class__	

8 Class threespace_api.TSDongle

object
└ **threespace_api.TSDongle**

The TSDongle class is an interface layer for 3-Space Sensor Dongle units that communicate through the USB port and acts as an autonomous object that handles communication between the dongle and wireless 3-Space Sensor units. Functions available through this class are either pythonic versions of the dongle's available commands or convenience functions for handling the establishment and use of the class's serial port.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the dongle takes place on.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor Dongle.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **device_type**: A string expressing the type of 3-Space Sensor device associated with the class instance. Since the connected device is always a dongle, this variable will always ahve a value of 'DNG'.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor Dongle commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before the next command is sent.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.
- **sensor_list**: A list that stores references to all wireless sensors created with this dongle.
- **pan_id**: An integer that stores the current panID for the dongle. This variable is used for restoring the panID of the dongle in the event of reconnection.
- **wireless_channel**: An integer that stores the current wireless channel for the dongle. This variable is used for restoring the wireless channel of the dongle in the event of reconnection.

- **wireless_table**: An array of 15 values. Each value represents the corresponding index into the dongle's wireless hardware ID table. This local copy is used for convenience's sake and may be read if the programmer requires the state of the entire table at once.
- **last_asynch**: A map that stores the last received asynchronous data from the wireless sensors in this dongle's sensor_list.
- **error**: A TSSError instance that keeps track of the last known error the class encountered.

8.1 Methods

__new__(*cls*, *com_port*, *read_timeout*=2, *write_timeout*=2, *baudrate*=115200)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__ extit(inherited documentation)

__init__(*self*, *com_port*, *read_timeout*=2.0, *write_timeout*=2.0, *baudrate*=115200)

Intializes the 3-Space Dongle

Parameters

`com_port:`	A string storing the name of the serial port intended for communication. The name of the port is dependent of the operating system.
`read_timeout:`	An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up. The default value is 2.0.
`write_timeout:`	An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up. The default value is 2.0.
`baudrate:`	An optional integer argument expressing the data transmission rate of the serial port. The default value is 115200 bits per second.

Overrides: object.__init__

__getitem__(*self*, *idx*)

Allows the TSDongle class be index like a list in order to retrieve wireless sensors that have already been connected.

Return Value

A TSWLSensor object associated with the TSDongle, or None if there is no sensor at the idx location.

close(*self*)

When called, the dongle's serial port is closed and the dongle's send/recieve thread is stopped.

read(*self*, *amnt_to_read*)

Returns the response to the last issued command as binary data.

Parameters

`amnt_to_read:` An integer indicating the number of bytes to attempt to read from the dongle's serial port.

Return Value

A binary string representing the bytes read from the serial port. If an error occurs, an empty string is returned.

write(*self*, *command_string*)

The passed in binary data is written to the dongle's serial port.

Parameters

`command_string:` A binary string of data to send on the serial port.

Return Value

A boolean. If the command was successfuly executed, True is returned. If the command was not successfuly executed, or the transmission of the command failed, False is returned.

isConnected(*self*, *try_reconnect*=`False`)

Tests if the dongle's serial port if properly connected and functioning.

Parameters

`try_reconnect:` An optional boolean that when True, an attempt at reconnection is made if the dongle is not connected. The default value is False.

Return Value

True is returned if the serial port is connected and working properly. False is returned if the port is not connected or has otherwise failed.

reconnect(*self*)

Attempts to establish a functioning connection to the dongle on the serial port.

Return Value

True is returned if the reconnection was successful. False is returned if the serial port could not be reconnnected.

addSensor(*self*, *idx*, *hw_id*, *axis_dir*=`None`)

A convenience function that creates and returns a new TSWLSensor object (or returns an existing one depending on input data) and updates the object's place in the dongle's sensor list. The dongle's wireless hardware list is also automatically updated based on the function's input.

Parameters

`idx:` An integer whose value is the logical ID (index into the dongle's wireless hardware table) to register the sensor identified by hw_id to.

`hw_id:` An integer whose value is the serial number of the wireless sensor desired to create a TSWLSensor object for.

`axis_dir:` An optional tuple argument that expresses a default axis directions configuration for the sensor. The first value of the tuple is a string expressing the ordering of the axis. The second value is a boolean expressing the negation of the X-Axis, the third value is a boolean expressing the negation of the Y-Axis, and the fourth value is a boolean expressing the negation of the Z-Axis. A value of True for any of the booleans indicates that the corresponding axis is to be negated. A value of False for any of the booleans indicates the axis is not to be negated. Valid string values for the first tuple value are:

- 'XZY'
- 'YXZ'
- 'YZX'
- 'ZXY'
- 'ZYX'

The default value is None, indicating that no default axis direction should be set. *(axis_order_string, neg_x_bool, neg_y_bool, neg_z_bool)*

Return Value

A TSWLSensor object representing the wireless sensor expressed by hw_id (or assigned to the Logical ID idx).

asynchBulkRead(*self*)

Get a table of asynchronous data recieved from wireless sensors associated with the dongle.

Return Value

A dictionary of data consisting of the data (arbitrary length of bytes) recieved asynchronously from wireless sensors along with timestamps (int) indicating their arrival time relative to the previously recieved packet. The dictionary is formatted as such: *data, timestamp = return_dict[sensor_serial_number]* None is returned if no data was recieved from the dongle.

getLastError(*self*)

Gets the last known error that the 3-Space Sensor unit encountered.

Return Value

A tuple describing the error type, 3-Space Sensor device that the error occured on, 3-Space Sensor device's serial number, the data that caused the error, the data's length, an exception and a message of the error if available.

getPanID(*self*)

Return the dongle's PanID.

Return Value

An integer whose value is the panID for the dongle. None is returned if there was some sort of connection error.

setPanID(*self*, *id*)

Set the dongle's PanID. Only devices with the same PanID will be able to communicate with it.

Parameters

`id:` An integer whose value is the desired panID for the dongle.

getChannel(*self*)

Return the dongle's wireless channel.

Return Value

An integer whose range is from 11-26 (decimal), inclusive. This value is the wireless channel. None is returned if there was some sort of connection error.

setChannel(*self*, *channel*)

Sets this dongle's wireless channel to the passed in value. Only devices on the same channel will be able to communicate.

Parameters

`channel`: An integer whose value is in the range 11-26 (decimal), inclusive. This value is the channel to assign to the sensor. An attempt to set the channel to a value outside this range will have no effect.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setLEDMode(*self*, *mode*)

Sets the mode of the dongle's LED (if wireless). The LED has two possible mode, 'static' and standard.

- If the LED is in 'static' mode, this means that it will only display the color set by the command setLEDColor.
- If the LED is in 'standard' mode, it will display the standard LED colors as described below:
  - Upon receipt of a packet, the dongle will flash green temporarily.
  - If the dongle transmits a packet that does not reach its destination, the dongle will flash red temporarily.

Parameters

`mode`: An integer of two valid values. If 0, the dongle will be set to 'standard' LED mode. If 1, the dongle will be set to 'static' LED mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

commitWirelessSettings(*self*)

Commits wireless configuration settings to the dongle's non-volitile memory. The settings committed with this command are:

- PanID
- Wireless Channel
- Wireless Address

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getWirelessAddress(*self*)

Reads the wireless address of the dongle.

Return Value

An integer whose value is the dongle's wireless address is returned. None is returned if there was some sort of connection error.

setWirelessAddress(*self*, *addr*)

Sets the wireless address of the dongle.

Parameters

`addr`: An integer whose value is the desired wireless address for the dongle.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getLEDMode(*self*)

Reads the mode of the dongle's LED (if wireless). The LED has two possible mode, 'static' and standard.

- If the LED is in 'static' mode, this means that it will only display the color set by the command setLEDColor.
- If the LED is in 'standard' mode, it will display the standard LED colors as described below:
  - Upon receipt of a packet, the wireless unit will flash green temporarily. This will occur regardless of whether the wireless unit is plugged in or not.

Return Value

An integer of two valid values. If 0, the dongle will be set to 'standard' LED mode. If 1, the dongle will be set to 'static' LED mode.

getWirelessHWID(*self*, *idx*)

Return the hardware ID (sensor serial number) that is associated with the logical ID given by an index.

Parameters

`idx:` An integer whose value is the index of the dongle's wireless table to read the hardware ID from.

Return Value

An integer whose value is the serial number of the wireless sensor currently listed in the dongle's wireless table under idx. None is returned if there was some sort of connection error.

setWirelessHWID(*self*, *idx*, *hw_id*)

Set the hardware ID (sensor serial number) that is associated with the logical ID given by an index. If the hw_id passed in is already in the wireless table, the previous idx the hw_id resided at will be set to 0 and the hw_id be set to the new idx. The dongle's internal sensor list is also updated based on the input to this function. Passing in 0 for hw_id effectivly clears the slot, preventing communication with that idx. The reference to a TSWLSensor at that idx is also removed, so if 0 is passed in the associated reference to that TSWLSensor should be deleted.

Parameters

`idx:` An integer whose value is the index of the dongle's wireless table to read the hardware ID from.

`hw_id:` An integer whose value is the serial number of the wireless sensor currently listed in the dongle's wireless table under idx.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getChannelNoise(*self*)

Returns an integer for each channel representing the level of noise on each respective channel.

Return Value

A list of 16 integers. Each of these values range from 0...255 with 0 meaning the channel is completely clear, and 255 meaning that it is unusably noisy. The first value returned corresponds to channel 11, and the last value returned corresponds to channel 26. None is returned if there was some sort of connection error.

setWirelessRetries(*self*, *retries*)

Set the number of times the dongle will retry a failed transmission to a wireless sensor.

Parameters

`retries:` An integer whose value is the desired number of times to retry a transmission to a wireless sensor before giving up. The maximum value, as well as the default value, is 3.

getWirelessRetries(*self*)

Returns the number of times the dongle will retry a failed transmission to a wireless sensor.

Return Value

An integer whose value is the desired number of times to retry a transmission to a wireless sensor before giving up. The maximum value, as well as the default value, is 3. None is returned if there was some sort of connection error.

getWirelessSlotsOpen(*self*)

All commands sent to the dongle to be transmitted wirelessly are buffered until previous pending transmissions complete. There are fifteen such slots, and while the dongle handles the management of them internally, no additional transmissions can be queued up if the maximum number of slots are occupied already. If sending a large batch of commands to a single sensor, it is useful to check this to make sure the transmission can be initiated.

Return Value

An integer whose value is the number of available wireless slots. The maximum number of slots possible is 15. None is returned if there was some sort of connection error.

getReceptionStrength(*self*)

Returns a value ranging from 0.. 255 indicating the relative strength of the most recent packet received by the dongle. The value scales linearly with distance from the dongle. Low values do not necessarily mean that transmissions are less accurate, but rather the strength of the signal that transmitted the last packet was weak.

Return Value

A float whose value is the signal strength of the last wireless packet recieved from a wireless sensor. None is returned if there was some sort of connection error.

setHIDUpdateRate(*self*, *rate*)

Sets the update rate of the mouse and joystick HID devices.

Parameters

`rate`: An integer whose value is the intended update rate of HID device polling in milliseconds. The minimum valid value is 5 milliseconds, while the maximum valid value is 50 milliseconds. The dongle's default rate is 25.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getHIDUpdateRate(*self*)

Gets the update rate of the mouse and joystick HID devices.

Return Value

An integer whose value is the intended update rate of HID device polling in milliseconds. The minimum value is 5 milliseconds, while the maximum value is 50 milliseconds. The dongle's default rate is 25.

restoreFactorySettings(*self*)

Restores all settings to factory settings. The settings are not committed to non-volatile memory by this command, so the commit settings command will have to be used if this is desired.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

commitSettings(*self*)

Commits settings to non-volatile memory. This will cause them to persist even if the dongle is reset.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

softwareReset(*self*)

Resets the dongle without powering it down. Any settings saved to non-volatile memory will be restored and the dongle. During the reset, the dongle will be inactive for approximatly 5 seconds. Certain setting changes only take place when a reset has occurred.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

enableWatchdogTimer(*self*, *timeout*)

Enables the watchdog timer with the given timeout rate. If a frame takes more than this amount of time, the dongle will automatically reset. This is useful for dealing with dongle hangs or crashes, as the dongle would reset and continue normal operation.

Parameters

`timeout`: An integer indicating the length of time in microseconds to wait before timing out when the watchdog timer is active.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

disableWatchdogTimer(*self*)

Disable the watchdog timer.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

enterFirmwareUpdateMode(*self*)

Puts the dongle into firmware update mode. This will cease normal operation until the firmware update mode is instructed to return the dongle to normal operation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setUSBMode(*self*, *mode*)

Sets the communication mode for USB. All modes present a COM port with which to communicate with the USB device. FTDI and CDC each present a regular numbered port, whereas Unique presents a port named YEI_<serial number>. The dongle will change modes immediately.

Parameters

`mode`: An integer whose value represents the desired USB mode for the dongle. Valid values are: 2 for Unique mode, 1 for FTDI mode, and 0 for CDC mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getUSBMode(*self*)

Gets the communication mode for USB. All modes present a COM port with which to communicate with the USB device. FTDI and CDC each present a regular numbered port, whereas Unique presents a port named YEI_<serial number>. The dongle will change modes immediately.

Return Value

An integer whose value represents the current USB mode for the sensor. Expected values are: 2 for Unique mode, 1 for FTDI mode, and 0 for CDC mode.

setClockSpeed(*self*, *speed*)

Sets the clock speed to the given value. It has an upper limit of 60 MHz and a lower limit of 10 MHz. This setting does not need to be committed, but does not take effect until the dongle is reset.

Parameters

`speed`: An integer that expresses the desired clock speed of the MCU in Hz. As an example, if one desired to set the clock speed to 10 MHz, the speed argument should be set to 10000000.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getClockSpeed(*self*)

Reads the clock speed of the dongle's MCU. It has an upper limit of 60 MHz and a lower limit of 10 MHz. This setting does not need to be committed, but does not take effect until the dongle is reset.

Return Value

An integer that expresses the current clock speed of the MCU in Hz. As an example, if current clock speed of the MCU was 10 MHz, the return value from this call be 10000000.

getSerialNumber(*self*)

Reads the serial number of the dongle. The serial number is a unique number for any one dongle.

Return Value

An integer whose value is the serial number of the dongle.

setLEDColor(*self*, *color*)

Sets the color of the LED on the dongle to the given RGB color.

Parameters

`color:` A tuple of 3 floats. Each float has an expected range of 0.0 (no color contribution) to 1.0 (greatest color contribution). The first float is the red channel, the second the green channel, and the third float is the blue channel. *(red, green, blue)*

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getLEDColor(*self*)

Gets the color of the LED on the dongle as an RGB tuple.

Return Value

A tuple of 3 floats. Each float has an expected range of 0.0 (no color contribution) to 1.0 (greatest color contribution). The first float is the red channel, the second the green channel, and the third float is the blue channel. *(red, green, blue)*

setJoystickLogicalID(*self*, *id*)

Set the logical ID of the wireless unit will operate as the joystick or disables joystick data transmission. Disabling joystick data transmission allows normal communication to occur at a faster rate.

Parameters

`id`: The logical ID (index in the dongle's wireless hardware table) associated with the wireless sensor that is to act as the data source for joystick HID streaming. If this value is set to -1, there will be no joystick input from any wireless sensor.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMouseLogicalID(*self*, *id*)

Set the logical ID of the wireless unit will operate as the mouse or disables mouse data transmission. Disabling mouse data transmission allows normal communication to occur at a faster rate.

Parameters

`id`: The logical ID (index in the dongle's wireless hardware table) associated with the wireless sensor that is to act as the data source for mouse HID streaming. If this value is set to -1, there will be no mouse input from any wireless sensor.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getJoystickLogicalID(*self*)

Gets the logical ID of the wireless unit that operates as the joystick.

Return Value

An integer whose value is the logical ID (index in the dongle's wireless hardware table) associated with the wireless sensor that acts as the data source for joystick HID streaming. If this value is -1, there is no joystick input from any wireless sensor.

getMouseLogicalID(*self*)

Gets the logical ID of the wireless unit that operates as the mouse.

Return Value

An integer whose value is the logical ID (index in the dongle's wireless hardware table) associated with the wireless sensor that acts as the data source for mosue HID streaming. If this value is -1, there is no mouse input from any wireless sensor.

setMouseRelative(*self*, *is_relative*)

Puts the mouse in absolute or relative mode. Please note that this change does not take effect immediately, and the dongle's settings must be committed and the dongle must be reset before the mouse will enter this mode.

Parameters

`is_relative`: A boolean whose value expressed whether the mouse is to be relative or not. If the value is True, the mouse is relative. If the value is False, the mouse is in absolute mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getMouseRelative(*self*)

Reads the current mouse absolute/relative state. Note that if the dongle has not been reset since it has been put in this state, the mouse will not reflect this change yet, even though this command will.

Return Value

A boolean whose value expressed whether the mouse is relative or not. If the value is True, the mouse is relative. If the value is False, the mouse is in absolute mode.

setJoystickMousePresent(*self*, *is_joy_present*, *is_mouse_present*)

Sets whether the joystick and mouse are present or removed. If removed, they will not show up as devices on the target system at all. For these changes to take effect, the dongle driver may need to be reinstalled.

Parameters

`is_joy_present:` A boolean whose value expresses whether the joystick is present on the system at all. If True, the joystick is present. If False, the joystick is not present.

`is_mouse_present:` A boolean whose value expresses whether the mouse is present on the system at all. If True, the mouse is present. If False, the mouse is not present.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getJoystickMousePresent(*self*)

Gets whether the joystick and mouse are present or removed. If removed, they will not show up as devices on the target system at all. For these changes to take effect, the dongle driver may need to be reinstalled.

Return Value

A list of Two boolean values that express whenther the joystick and mouse are present or not. The first value expresses whether the joystick is present on the system at all. If True, the joystick is present. If False, the joystick is not present. The second value expresses whether the mouse is present on the system at all. If True, the mouse is present. If False, the mouse is not present.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

8.2 Properties

Name	Description
Inherited from object	
__class__	

9 Class threespace_api.TSEMSensor

The TSEMSensor class is an interface layer for 3-Space Sensor Embedded units that communicate through the USB port or RS-232 port. Functions available through this class are either pythonic versions of the sensor's available commands or convenience functions for handling the establishment and use of the class's serial port.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **device_type**: A string expressing the type of 3-Space Sensor associated with the class instance.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the sensor takes place on.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before the next command is sent.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.
- **error**: A TSSError instance that keeps track of the last known error the class encountered.

9.1 Methods

setInterruptType(*self*, *mode*, *pin*)

Configures how the sensor will generate interrupts.

Parameters

`mode:` An integer that specifies how the sensor will generate the interupt and for how long. If mode is 0, no generation will occur. If mode is 1, the interrupt generator will set the specified pin low for 5 microseconds when new data is available. If mode is 2, the interrupt generator will set the pin low until the interrupt status is read with the getInterruptStatus function.

`pin:` An integer that defines what pin on the sensor to generate the interupt on. If pin is 0, the TXD pin is used. If pin is 1, the MISO pin is used to generate the interrupt. The method of communication each pin belongs to cannot be uwed while that pin is being used for interrupt generation. Note that the pin cannot be changed to TXD over the UART, and cannot be changed to MISO over SPI.

Return Value

A boolean indicating whether the command was successfuly written to the serial port. True indicates a successful write. False indicates there was a problem with writing to the port.

getInterruptType(*self*)

Returns how the sensor currently generates interrupts.

Return Value

A list of two integers. The list is formatted as such: *[mode, pin]*. Mode is an integer that specifies how the sensor will generate the interupt and for how long. If mode is 0, no generation will occur. If mode is 1, the interrupt generator will set the specified pin low for 5 microseconds when new data is available. If mode is 2, the interrupt generator will set the pin low until the interrupt status is read with the getInterruptStatus function. Pin is an integer that defines what pin on the sensor to generate the interupt on. If pin is 0, the TXD pin is used. If pin is 1, the MISO pin is used to generate the interrupt. The method of communication each pin belongs to cannot be uwed while that pin is being used for interrupt generation. Note that the pin cannot be changed to TXD over the UART, and cannot be changed to MISO over SPI.

getInterruptStatus(*self*)

Read the current interrupt state. Calling this command while interrupts are in mode '2' (see definition for setInterruptType for more details) will cause the interrupt pin to return to high.

Return Value

An integer whose value can either be 0 or 1. If the value is 0, there is no new data since the last orientation read. If the value is 1, then new data is available. None is returned if there has been an issue with communication with the sensor.

Inherited from threespace_api.TSSensor(Section 10)

__init__(), __new__(), calibrateGyro(), close(), commitSettings(), disableAxis(), disableButton(), disableWatchdogTimer(), enableWatchdogTimer(), enterFirmwareUpdateMode(), getAccelerometerCalibrationParam(), getAccelerometerEnabled(), getAccelerometerNormalized(), getAccelerometerRange(), getAccelerometerRaw(), getAccelerometerUnfiltered(), getActualUpdateRate(), getAllSensorsNormalized(), getAllSensorsRaw(), getAvgPercent(), getAxisDirections(), getButtonGyroDisableLength(), getButtonState(), getClockSpeed(), getCompassCalibrationParam(), getCompassEnabled(), getCompassNormalized(), getCompassRange(), getCompassRaw(), getCompassUnfiltered(), getConfidence(), getControlData(), getControlMode(), getDesiredUpdateRate(), getFiltGyroRate(), getFiltNorthEarthVecSensFrame(), getFiltOrientAxisAngle(), getFiltOrientEuler(), getFiltOrientFwdDwn(), getFiltOrientMat(), getFiltOrientQuat(), getFiltTaredFwdDwnVecSensFrame(), getFiltTaredOrientAxisAngle(), getFiltTaredOrientEuler(), getFiltTaredOrientFwdDwn(), getFiltTaredOrientMat(), getFiltTaredOrientQuat(), getFilterMode(), getGyroCalibrationParam(), getGyroEnabled(), getGyroNormalized(), getGyroRaw(), getGyroscopeRange(), getJoystickEnabled(), getJoystickMousePresent(), getKalmanMat(), getLEDColor(), getLastError(), getLookupTblVertVal(), getMouseEnabled(), getMouseRelative(), getMultiRefChkVecAccelerometer(), getMultiRefChkVecCompass(), getMultiRefResolution(), getMultiRefVecAccelerometer(), getMultiRefVecCompass(), getMultiRefWeightPwr(), getNumMultiRefCells(), getOversampleRate(), getRefVecAccelerometer(), getRefVecCompass(), getRefVecMode(), getRhoDataAccelerometer(), getRhoDataCompass(), getRunningAverageMode(), getSerialNumber(), getTareOrientMat(), getTareOrientQuat(), getTemperatureCelsius(), getTemperatureFahrenheit(), getUARTRate(), getUSBMode(), isConnected(), read(), reconnect(), resetKalmanFilter(), restoreFactorySettings(), setAccelerometerCalibrationParam(), setAccelerometerEnabled(), setAccelerometerRange(), setAvgPercent(), setAxisDirections(), setButtonGyroDisableLength(), setClockSpeed(), setCompassCalibrationParam(), setCompassEnabled(), setCompassRange(), setConfidenceRhoModeAccel(), setConfidenceRhoModeCompass(), setControlData(), setControlMode(), setFilterMode(), setGlobalAxis(), setGyroCalibrationParam(), setGyroEnabled(), setGyroscopeRange(), setJoystickEnabled(), setJoystickMousePresent(), setLEDColor(), setLookupTblVertVal(), setMouseEnabled(), setMouseRelative(), setMul-

tiRefChkVecAccelerometer(), setMultiRefChkVecCompass(), setMultiRefResolution(), setMultiRefVecAccelerometer(), setMultiRefVecCompass(), setMultiRefVecZero(), setMultiRefWeightPwr(), setOrientationButton(), setOversampleRate(), setPhysicalButton(), setRefVecAccelerometer(), setRefVecCompass(), setRefVecCurrentOrient(), setRefVecMode(), setRunningAverageMode(), setScreenPointAxis(), setShakeButton(), setStaticRhoModeAccel(), setStaticRhoModeCompass(), setTareCurrentOrient(), setTareMatrix(), setTareQuaternion(), setUARTRate(), setUSBMode(), setUpdateRate(), setupSimpleJoystick(), setupSimpleLightgun(), setupSimpleMouse(), softwareReset(), write()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

9.2 Properties

Name	Description
Inherited from object	
__class__	

10 Class threespace_api.TSSensor

object
└ **threespace_api.TSSensor**

Known Subclasses: threespace_api.TSBTSensor, threespace_api.TSDLSensor, threespace_api.TSEMSens
threespace_api.TSWLSensor

The TSSensor class is an interface layer for 3-Space Sensor USB units that communicate through the USB port. Functions available through this class are either pythonic versions of the sensor's available commands or convenience functions for handling the establishment and use of the class's serial port.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **device_type**: A string expressing the type of 3-Space Sensor associated with the class instance.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the sensor takes place on.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before the next command is sent.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.
- **error**: A TSSError instance that keeps track of the last known error the class encountered.

10.1 Methods

__**new**__(*cls*, *com_port*=`None`, *axis_dir*=`None`, *read_timeout*=`0.5`, *write_timeout*=`0.5`, *baudrate*=`115200`)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__ extit(inherited documentation)

__init__(*self*, *com_port*, *axis_dir*=`None`, *read_timeout*=`0.5`, *write_timeout*=`0.5`, *baudrate*=`115200`)

Initializes the 3-Space Sensor.

Parameters

`com_port`: A string storing the name of the serial port intended for communication. The name of the port is dependent of the operating system.

`axis_dir`: An optional tuple argument that expresses a default axis directions configuration for the sensor. The first value of the tuple is a string expressing the ordering of the axis. The second value is a boolean expressing the negation of the X-Axis, the third value is a boolean expressing the negation of the Y-Axis, and the fourth value is a boolean expressing the negation of the Z-Axis. A value of True for any of the booleans indicates that the corresponding axis is to be negated. A value of False for any of the booleans indicates the axis is not to be negated. Valid string values for the first tuple value are:

- 'XYZ'
- 'XZY'
- 'YXZ'
- 'YZX'
- 'ZXY'
- 'ZYX'

The default value is None, indicating that no default axis direction should be set.

`read_timeout`: An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up. The default value is 2.0.

`write_timeout`: An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up. The default value is 2.0.

`baudrate`: An optional integer argument expressing the data transmission rate of the serial port. The default value is 115200 bits per second.

Overrides: object.__init__

close(*self*)

Closes the sensor's serial port, preventing any future communication.

read(*self*, *size*)

Reads a specificed number of bytes from the sensor's serial port.

Parameters

`size`: An integer expressing the number of bytes to attempt to read from the sensor's serial port.

Return Value

A binary string representing the bytes read from the serial port. If an error occurs, an empty string is returned.

write(*self*, *data*)

Writes a binary string of data to the sensor's serial port.

Parameters

`data`: A binary string of data.

Return Value

True is returned if the write was successful. False is returned if some sort of error occured. When a write error occurs, the sensor's serial port is also closed and set to None.

isConnected(*self*, *try_reconnect*=`False`)

Tests if the sensor's serial port if properly connected and functioning.

Parameters

`try_reconnect`: An optional boolean that when True, an attempt at reconnection is made if the sensor is not connected. The default value is False.

Return Value

True is returned if the serial port is connected and working properly. False is returned if the port is not connected or has otherwise failed.

reconnect(*self*)

Attempts to establish a functioning connection to the sensor on the serial port.

Return Value

True is returned if the reconnection was successful. False is returned if the serial port could not be reconnnected.

getLastError(*self*)

Gets the last known error that the 3-Space Sensor unit encountered.

Return Value

A tuple describing the error type, 3-Space Sensor device that the error occured on, 3-Space Sensor device's serial number, the data that caused the error, the data's length, an exception and a message of the error if available.

getFiltTaredOrientQuat(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the tare orientation. This version returns the data as a quaternion.

Return Value

A list of floats is returned. The list represents the elements of the quaternion. The ordering of the list is as such: (x, y, z, w). None is returned if no meaningful data could be retrieved.

getFiltTaredOrientEuler(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the tare orientation. This version returns the data as a set of Euler angles (in radians).

Return Value

A list of floats is returned. The list represents the set of Euler Angles. The ordering of the list is as such: [pitch, yaw, roll]. None is returned if no meaningful data could be retrieved.

getFiltTaredOrientMat(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the tare orientation. This version returns the data as a 3x3 rotation matrix (in row major).

Return Value

A list of floats is returned. The list is a 1D array of all elements of the matrix. The array is formatted in "row major", meaning that the first 3 values in the array correspond to the first row of the matrix, the second three values correspond to the second row of the matrix, and the third triplet with the third row. None is returned if no meaningful data could be read.

getFiltTaredOrientAxisAngle(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the tare orientation. This version returns the data as an Axis Angle construct (x, y, z, angle_in_radians).

Return Value

A list of floats is returned. Each element in the list corresponds to an element in the expressed Axis Angle construct. The list is formatted as such: (x, y, z, angle_in_radians). None is returned if no meaningful data could be read from the sensor.

getFiltTaredOrientFwdDwn(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the tare orientation. This version returns the data as the forward and down vectors of the coordinate system defined by the orientation.

Return Value

A list of 6 floats is returned. The first three floats correspond to the expressed "Forward" vector. The second three floats correspond to the expressed "Down" vector. The list is formatted as such: *(Fwd_x, Fwd_y, Fwd_z, Dwn_x, Dwn_y, Dwn_z)*. None is returned if no meaningful data could be read.

getFiltGyroRate(*self*)

Returns the gyro rates as determined by taking the difference between the current orientation and the previous one.

Return Value

A list of floats is returned. The list is a quaternion expressing the rate of change of the sensor's gyros. The quaternion is formatted as such: *(x, y, z, w)*. None is returned if no meaningful data could be read.

getFiltOrientQuat(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the reference vectors. This version returns the data as a quaternion.

Return Value

A list of floats is returned. The list is a quaternion expressing the filered orientation of the sensor. The quaternion is formatted as such: *(x, y, z, w)*. None is returned if no meaningful data could be read.

getFiltOrientEuler(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the reference vectors. This version returns the data as a set of Euler angles (in radians).

Return Value

A list of floats is returned. The list represents the set of Euler Angles. The ordering of the list is as such: *[pitch, yaw, roll]*. None is returned if no meaningful data could be retrieved.

getFiltOrientMat(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the reference vectors. This version returns the data as a 3x3 rotation matrix (in row major).

Return Value

A list of floats is returned. The list is a 1D array of all elements of the matrix. The array is formatted in "row major", meaning that the first 3 values in the array correspond to the first row of the matrix, the second three values correspond to the second row of the matrix, and the third triplet with the third row. None is returned if no meaningful data could be retrieved.

getFiltOrientAxisAngle(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the reference vectors. This version returns the data as an axis and an angle.

Return Value

A list of floats is returned. Each element in the list corresponds to an element in the expressed Axis Angle construct. The list is formatted as such: *(x, y, z, angle_in_radians)*. None is returned if no meaningful data could be read from the sensor.

getFiltOrientFwdDwn(*self*)

Returns the final orientation as determined by the Kalman filter, relative to the reference vectors. This version returns the data as the forward and down vectors of the coordinate system defined by the orientation.

Return Value

A list of 6 floats is returned. The first three floats correspond to the expressed "Forward" vector. The second three floats correspond to the expressed "Down" vector. The list is formatted as such: *(Fwd_x, Fwd_y, Fwd_z, Dwn_x, Dwn_y, Dwn_z)*. None is returned if no meaningful data could be read.

getFiltTaredFwdDwnVecSensFrame(*self*)

Returns the forward and down vectors as determined by the Kalman filter, relative to the tare orientation. This version returns the data as the forward and down vectors of the coordinate system defined by the sensor reference frame.

Return Value

A list of 6 floats is returned. The first three floats correspond to the expressed "Forward" vector. The second three floats correspond to the expressed "Down" vector. The list is formatted as such: *(Fwd_x, Fwd_y, Fwd_z, Dwn_x, Dwn_y, Dwn_z)*. None is returned if no meaningful data could be read.

getFiltNorthEarthVecSensFrame(*self*)

Returns the North and Earth vectors as determeined by the Kalman filter, relative to the global sensor references. Returned vectors are in the coordinate system defined by the sensor reference frame.

Return Value

A list of 6 floats is returned. The first three floats correspond to the expressed "North" vector. The second three floats correspond to the expressed "Earth" vector. The list is formatted as such: *(North_x, North_y, North_z, Earth_x, Earth_y, Earth_z)*. None is returned if no meaningful data could be read.

getAllSensorsNormalized(*self*)

Returns the normalized data of the 3-Space Sensor's on-board sensors.

Return Value

A tuple of 9 floats is returned. The first three elements are the normalized data from the sensor's gyro, the second three elements are the normalized data from the sensor's accelerometer, and the last three elements are the normalized data from the sensor's compass. *(gyro Vector 3, accelerometer Vector3, compass Vector3)* None is returned if no meaningful data could be read.

getGyroNormalized(*self*)

Returns the normalized gyro rates of the 3-Space Sensor.

Return Value

A tuple of 3 floats is returned. The three elements are the normalized data from the sensor's gyro, expressed as a Vector 3. The data is formatted as such: *(x, y, z)*. None is returned if no meaningful data could be read.

getAccelerometerNormalized(*self*)

Returns the normalized gravity vector of the 3-Space Sensor.

Return Value

A tuple of 3 floats is returned. The three elements are the normalized data from the sensor's accelerometer, expressed as a Vector 3. The data is formatted as such: *(x, y, z)*. None is returned if no meaningful data could be read.

getCompassNormalized(*self*)

Returns the normalized north vector of the 3-Space Sensor.

Return Value

A tuple of 3 floats is returned. The three elements are the normalized data from the sensor's compass, expressed as a Vector 3. The data is formatted as such: *(x, y, z)*. None is returned if no meaningful data could be read.

getTemperatureCelsius(*self*)

Returns the temperature of the sensor in Celsius.

Return Value

A float is returned. The value of the float is the current temperature of the 3-Space Sensor in Celsius. None is returned if no meaningful data could be read.

getTemperatureFahrenheit(*self*)

Returns the temperature of the sensor in Fahrenheit.

Return Value

A float is returned. The value of the float is the current temperature of the 3-Space Sensor in Fahrenheit. None is returned if no meaningful data could be read.

getConfidence(*self*)

Returns a value indicating how much the compass and accelerometer are to be trusted to be unit vectors pointing in the proper directions at the moment.

Return Value

A float is returned. This value ranges from 0 to 1, with 1 being fully trusted and 0 being not trusted at all. This will change based on if the sensor is being moved and if it is near a strong magnetic field. None is returned if no meaningful data could be read.

getAccelerometerUnfiltered(*self*)

Returns the unfiltered (but compensated) data of the 3-Space Sensor's on-board accelerometer.

Return Value

A tuple of 3 floats is returned. The three elements are the unfiltered data from the sensor's accelerometer. *(accelerometer Vector 3)* None is returned if no meaningful data could be read.

getCompassUnfiltered(*self*)

Returns the unfiltered (but compensated) data of the 3-Space Sensor's on-board compass.

Return Value

A tuple of 3 floats is returned. The three elements are the unfiltered data from the sensor's compass. *(compass Vector 3)* None is returned if no meaningful data could be read.

getAllSensorsRaw(*self*)

Returns the raw data of the 3-Space Sensor's on-board sensors.

Return Value

A tuple of 9 floats is returned. The first three elements are the raw data from the sensor's gyro, the second three elements are the raw data from the sensor's accelerometer, and the last three elements are the raw data from the sensor's compass. *(gyro Vector 3, accelerometer Vector3, compass Vector3)* None is returned if no meaningful data could be read.

getGyroRaw(*self*)

Returns the raw gyro value as a vector.

Return Value

A list of 3 floats is returned. The list's value is the current value of the 3-Space Sensor's gyro. None is returned if no meaningful data could be read.

getAccelerometerRaw(*self*)

Returns the raw accelerometer value as a vector.

Return Value

A list of 3 floats is returned. The list's value is the current value of the 3-Space Sensor's accelerometer. None is returned if no meaningful data could be read.

getCompassRaw(*self*)

Returns the raw north vector reading from the compass.

Return Value

Returns the raw compass value as a vector.

setTareCurrentOrient(*self*)

Sets current filtered orientation as the tare orientation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setTareQuaternion(*self*, *quat*)

Sets a passed in quaternion as the tare orientation.

Parameters

`quat`: A list of 4 floats. The list represents a quaternion to use as a tare orientation for the sensor. The list must be formatted as such: *(x, y, z, w)*.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setTareMatrix(*self*, *mat*)

Sets a passed in 3x3 matrix as the tare orientation.

Parameters

`mat`: A list of 9 floats. The list represents a 1D array of elements for a 3x3 matrix. The array is to be formatted such that the matrix is in "row major". In other words, the first 3 values in the array correspond to the first row of the matrix, the second three values correspond to the second row of the matrix, and the third triplet with the third row.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setStaticRhoModeAccel(*self*, *val*)

Sets the rho mode for the accelerometer to static, using the given value as rho.

Parameters

`val`: A float representing the value to be used as Rho.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setConfidenceRhoModeAccel(*self*, *min*, *max*)

Sets the rho mode for the accelerometer to confidence, using the given values (two floats) as the min and the max. The confidence factor will be used in real time to interpolate between these to determine what rho value is used.

Parameters

`min`: A float representing the minimum Rho value to be used for calculation.

`max`: A float representing the maximmum Rho value to be used for calculation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setStaticRhoModeCompass(*self*, *val*)

Sets the rho mode for the compass to static, using the given value as rho.

Parameters

`val`: A float representing the Rho value to be used for calculation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setConfidenceRhoModeCompass(*self*, *min*, *max*)

Sets the rho mode for the compass to confidence, using the given values as the min and the max. The confidence factor will be used in real time to interpolate between these to determine what rho value is used.

Parameters

`min`: A float representing the minimum Rho value to be used for calculation.

`max`: A float representing the maximmum Rho value to be used for calculation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setUpdateRate(*self*, *rate*)

Sets the target update rate to the given rate. If the filter takes less time to update during a given frame than this rate specifies, the frame will be padded out to take the specified amount of time. If the filter takes more time to update than this rate, this rate will be ignored.

Parameters

`rate`: An integer representing the desired update rate in microseconds.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setRefVecCurrentOrient(*self*)

Uses the current tared orientation to set up the reference vector for the nearest orthogonal orientation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setRefVecMode(*self*, *mode*)

Sets reference vector mode to either 'single static', 'single auto', 'single auto continuous', or 'multi'.

Single static mode uses a certain reference vector for the compass and another certain reference vector for the accelerometer at all times.

Single auto mode uses (0,-1,0) as the reference vector for the accelerometer at all times and uses the current average angle between the accelerometer and compass to calculate the compass reference vector. After that this mode acts like single static mode.

Single auto continuous mode uses (0,-1,0) as the reference vector for the accelerometer at all times and uses the average angle between the accelerometer and compass to constantly redetermine the compass reference vector.

Multi mode uses a collection of reference vectors for the compass and accelerometer, and selects which ones to use before each step of the filter.

Parameters

`mode`: An integer representing the desierd mode of operation. (0 for single static, 1 for single auto, 2 for single auto continuous, 3 for multi)

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setOversampleRate(*self*, *rate*)

Sets oversample rate of data reads from the 3-Space Sensor's sensors.

Parameters

`rate`: An integer prepresenting the amount of oversampling to perform. 1 turns off oversampling. 2+ sets the number of samples per frame.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setGyroEnabled(*self*, *do_enable*)

Enables or disables the gyros (will be replaced with a still gyro reading if disabled).

Parameters

`do_enable`: A boolean value expressing whether to enable or disable the gyro *(False - Disabled, True - Enabled)*.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setAccelerometerEnabled(*self*, *do_enable*)

Enables or disables the accelerometer (This filter will not use this data if disabled).

Parameters

`do_enable`: A boolean value expressing whether to enable or disable the accelerometer *(False - Disabled, True - Enabled)*.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setCompassEnabled(*self*, *do_enable*)

Enables or disables the compass(This filter will not use this data if disabled).

Parameters

`do_enable`: A boolean value expressing whether to enable or disable the compass *(False - Disabled, True - Enabled)*.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefVecZero(*self*)

Resets all reference vectors in the multi reference scheme to zero.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefResolution(*self*, *num_division*, *num_vecs*)

Sets number of cell divisions and number of nearby vectors per cell for the multi reference lookup table.

By default, the multiple reference vector mode only deals with orientations obtainable by successive rotations of 90 degrees about any of the three axes. This command can adjust it to accept orientations obtainable by 45 degree rotations. For 90 degrees, give a "number of cell divisions" of 4, and for 45 give 8. In addition, the number of vectors near to any given orientation which the scheme will check can be adjusted as well. If this value is 0, only the nearest orientation will be checked. The largest this value can be is 32. Also note that the larger this value is, the longer the multiple reference vector mode will take to run each cycle.

Parameters

`num_division:` An integer representing the number of cell divisions.

`num_vecs:` An integer representing the number of nearby vectors.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefVecCompass(*self*, *idx*, *vec*)

Directly set compass multi reference vector at the specified index to the specified vector.

Parameters

`idx:` An integer representing the index of the compass to set the vector to.

`vec:` A list of 3 floats expressing the vector to set at idx.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefChkVecCompass(*self*, *idx*, *vec*)

Directly set compass multi reference check vector at the specified index to the specified vector.

Parameters

`idx`: An integer representing the index of the compass to set the vector to.

`vec`: A list of 3 floats expressing the vector to set at idx.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefVecAccelerometer(*self*, *idx*, *vec*)

Directly set accel multi reference vector at the specified index to the specified vector.

Parameters

`idx`: An integer representing the index of the accelerometer to set the vector to.

`vec`: A list of 3 floats expressing the vector to set at idx.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefChkVecAccelerometer(*self*, *idx*, *vec*)

Directly set accel multi reference check vector at the specified index to the specified vector.

Parameters

`idx`: An integer representing the index of the accelerometer to set the vector to.

`vec`: A list of 3 floats expressing the vector to set at idx.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setAxisDirections(*self*, *axis_order*, *neg_x*=`False`, *neg_y*=`False`, *neg_z*=`False`)

Set which directions each of the natural axes of the sensor point. (For example, using ZXY means that whatever value appears as X on the natural axes will now be the Z component of any new data). This function also sets which axis are to be flipped (negated). If the axis's bool is False, the axis is pointing in the positive direction, and if the bool is True, the axis is flipped. (Note: these are applied to the axes after the 'axis_order' conversion takes place).

Parameters

`axis_order`: A string representing the intended assignment of axis calculating orientations. Valid configuration strings are:

- 'XYZ'
- 'XZY'
- 'YXZ'
- 'YZX'
- 'ZXY'
- 'ZYX'

`neg_x`: A boolean that when True negates the X axis.

`neg_y`: A boolean that when True negates the Y axis.

`neg_z`: A boolean that when True negates the Z axis.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setAvgPercent(*self*, *val*)

Sets what percentage of running average to use on the sensor's orientation. This is computed as follows:

total_orient = total_orient * percent total_orient = total_orient + current_orient * (1 - percent) current_orient = total_orient

If the percentage is 0, the running average will be shut off completely.

Parameters

`val`: A float representing the percentage of the running average to use for the sensor's orientation. Valid values are within the range 0.0 to 1.0.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setRefVecCompass(*self*, *vec*)

Sets the static compass reference vector.

Parameters

`vec`: A list of 3 floats expressing the vector to set the compass's reference vector to.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setRefVecAccelerometer(*self*, *vec*)

Sets the static accelerometer reference vector.

Parameters

`vec`: A list of 3 floats expressing the vector to set the accelerometer's reference vector to.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

resetKalmanFilter(*self*)

Resets Kalman filter's covariance and state matrices.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setAccelerometerRange(*self*, *range*)

Set which range of accelerometer data to use. Higher ranges can detect and report larger accelerations, but are not as accurate for smaller ones.

Parameters

`range`: An integer representing the intended range for the accelerometer. Valid values are: 0 for 2G, 1 for 4G, and 2 for 8G.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMultiRefWeightPwr(*self*, *pow*)

Set weighting power for multi reference vector weights. Multi reference vector weights are all raised to the weight power before they are summed and used in the calculation for the final reference vector. Setting this value nearer to 0 will cause the reference vectors to overlap more, and setting it nearer to infinity will cause the reference vectors to influence a smaller set of orientations.

Parameters

`pow:` A float representing the weight power to use.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setFilterMode(*self*, *mode*)

Used to disable the orientation filter or set the orientation filter mode. Changing this parameter can be useful for tuning filter-performance versus orientation-update rates. When using the sensor as an IMU, it will improve performance to disable the orientation filter. When using as an AHRS, where orientation is needed, full-Kalman, gyroless-filtered, or fast-filtered should be used.

Parameters

`mode:` An integer whose value expresses the mode to set the sensor's filter to. Possible values are:

- 0 - Disables filter
- 1 - Full Kalman Filter
- 2 - Gyroless Filtered Orientation (Not Yet Supported)
- 3 - Fast Filtered Orientation (Not Yet Supported)

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setRunningAverageMode(*self*, *mode*)

Selects the mode that the running-average method uses. 'Normal' uses a static running-average. 'Confidence' uses a running average that changes dynamically based upon the confidence factor.

Parameters

`mode`: An integer whose value expresses the mode to set the sensor's running average to. Possible values are:

- 0 - Normal mode
- 1 - Confidence Mode

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setGyroscopeRange(*self*, *range*)

Sets the measurment range used by the gyroscope sensor.

Parameters

`range`: An integer whose value expresses the range mode to set the gyroscope to. Possible values are:

- 0 - +/- 250dps mode
- 1 - +/- 500dps mode
- 2 - +/- 2000dps mode

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setCompassRange(*self*, *range*)

Sets the measurement range used by the compass sensor.

Parameters

`range`: An integer whose value expresses the range mode to set the compass to. Possible values are:

- 0 - +/- 0.88Ga mode
- 1 - +/- 1.30Ga mode
- 2 - +/- 1.90Ga mode
- 3 - +/- 2.50Ga mode
- 4 - +/- 4.00Ga mode
- 5 - +/- 4.70Ga mode
- 6 - +/- 5.60Ga mode
- 7 - +/- 8.10Ga mode

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getTareOrientQuat(*self*)

Reads the tare orientation as a quaternion.

Return Value

A list of 4 floats expressing a quaternion. The quaternion is formatted as such: *(x, y, z, w)*.

getTareOrientMat(*self*)

Reads the tare orientation as a rotation matrix.

Return Value

A list of 9 floats expressing a 3x3 matrix. The list is a 1D array listing the elements of the matrix is a 'row major' format *(The first 3 elements of the list are the first row of the matrix, the second 3 elements are the second row, and the third triplet is the last row)*.

getRhoDataAccelerometer(*self*)

Reads the rho mode and rho data for the accelerometer. Rho mode, min/static rho, and the max rho (if the rho mode is 'confidence') are returned.

Return Value

A list of either 2 or 3 floats. The first element is the rho mode. If the value is 0, the rho mode is 'static'. If the value is 1, the rho mode is 'confidence'. The second element is the static rho is the mode is 'static'. If the mode is 'confidence', the second value is the minimum rho value. If the mode is 'confidence', there will also be a third integer in the list. The third integer represents the maximum rho value.

getRhoDataCompass(*self*)

Reads the rho mode and rho data for the compass. Rho mode, min/static rho, and the max rho (if the rho mode is 'confidence') are returned.

Return Value

A list of either 2 or 3 floats. The first element is the rho mode. If the value is 0, the rho mode is 'static'. If the value is 1, the rho mode is 'confidence'. The second element is the static rho is the mode is 'static'. If the mode is 'confidence', the second value is the minimum rho value. If the mode is 'confidence', there will also be a third integer in the list. The third integer represents the maximum rho value.

getActualUpdateRate(*self*)

Reads the amount of time the last frame took.

Return Value

An integer representing the amount of time the last frame of the kalman filter took to process in microseconds.

getRefVecCompass(*self*)

Reads the single mode compass reference vector.

Return Value

A list of 3 floats expressing the reference vector. The vector is formatted as such: *(x, y, z)*.

getRefVecAccelerometer(*self*)

Reads the single mode accelerometer reference vector.

Return Value

A list of 3 floats expressing the reference vector. The vector is formatted as such: *(x, y, z)*.

getRefVecMode(*self*)

Reads the reference vector mode.

Single static mode uses a certain reference vector for the compass and another certain reference vector for the accelerometer at all times.

Single auto mode uses (0,-1,0) as the reference vector for the accelerometer at all times and uses the current average angle between the accelerometer and compass to calculate the compass reference vector. After that this mode acts like single static mode.

Single auto continuous mode uses (0,-1,0) as the reference vector for the accelerometer at all times and uses the average angle between the accelerometer and compass to constantly redetermine the compass reference vector.

Multi mode uses a collection of reference vectors for the compass and accelerometer, and selects which ones to use before each step of the filter.

Return Value

A single integer is returned. The integer will be one of 4 possible values. 0 for single static, 1 for single auto, 2 for single auto continuous, and 3 for multi.

getMultiRefVecCompass(*self*, *idx*)

Reads the multi mode compass reference vector at index.

Parameters

`idx:` An integer representing the index of the desired reference vector.

Return Value

A list of 3 floats is returned. The floats represent the reference vector found at the inputed index. The vector is formatted as such: *(x, y, z)*.

getMultiRefChkVecCompass(*self*, *idx*)

Reads the multi mode compass reference check vector at index.

Parameters

`idx:` An integer representing the index of the desired reference check vector.

Return Value

A list of 3 floats is returned. The floats represent the reference vector found at the inputed index. The vector is formatted as such: *(x, y, z)*.

getMultiRefVecAccelerometer(*self*, *idx*)

Reads the multi mode accelerometer reference vector at index.

Parameters

`idx:` An integer representing the index of the desired reference vector.

Return Value

A list of 3 floats is returned. The floats represent the reference vector found at the inputed index. The vector is formatted as such: *(x, y, z)*.

getMultiRefChkVecAccelerometer(*self*, *idx*)

Reads the multi mode accelerometer reference check vector at index.

Parameters

`idx:` An integer representing the index of the desired reference check vector.

Return Value

A list of 3 floats is returned. The floats represent the reference vector found at the inputed index. The vector is formatted as such: *(x, y, z)*.

getGyroEnabled(*self*)

Reads the enabled state of the gyros.

Return Value

A boolean. A value of True indicates that the gyro is enabled. A value of False indicates that the gyro is disabled.

getAccelerometerEnabled(*self*)

Reads the enabled state of the Accelerometer.

Return Value

A boolean. A value of True indicates that the accelerometer is enabled. A value of False indicates that the accelerometer is disabled.

getCompassEnabled(*self*)

Reads the enabled state of the Compass.

Return Value

A boolean. A value of True indicates that the compass is enabled. A value of False indicates that the compass is disabled.

getAxisDirections(*self*)

Gets which directions each of the natural axes of the sensor point. (For example, using ZXY means that whatever value appears as X on the natural axes will now be the Z component of any new data). This function also gets which axis are to be flipped (negated). If the axis's bool is False, the axis is pointing in the positive direction, and if the bool is True, the axis is flipped. (Note: these are applied to the axes after the 'axis_order' conversion takes place).

Return Value

A tuple of 4 values. The first is A string representing the intended assignment of axis calculating orientations. Expected string values are:

- 'XYZ'
- 'XZY'
- 'YXZ'
- 'YZX'
- 'ZXY'
- 'ZYX'

The other three values are booleans expressing the negation of axes. If a boolean is True, than its axis is flipped (negated). If the bool is False, the axis is not flipped. The first bool is for the X-Axis, the second is for the Y-Axis, and the third is for the Z-axis. *(axis_order, neg_x, neg_y, neg_z)*

getOversampleRate(*self*)

Read oversample rate.

Return Value

An integer representing the oversample rate. 1 means oversampling is off. 2+ indicated the number of samples per frame.

getAvgPercent(*self*)

Reads what percentage of running average to use on the sensor's orientation. This is computed as follows:

total_orient = total_orient * percent total_orient = total_orient + current_orient * (1 - percent) current_orient = total_orient

If the percentage is 0, the running average will be shut off completely.

Return Value

A float representing the percentage of the running average to use for the sensor's orientation. Expected values are within the range 0.0 to 1.0.

getDesiredUpdateRate(*self*)

Reads the current desired update rate. If the filter takes less time to update during a given frame than this rate specifies, the frame will be padded out to take the desired amount of time. If the filter takes more time to update than this rate, this rate will be ignored.

Return Value

An integer representing the amount of time desired between each frame of kalman filter calculations in microseconds.

getKalmanMat(*self*)

Read Kalman filter's covariance 3x3 matrix.

Return Value

A list of 9 floats. The list is a 1D array of the elements of a 3x3 matrix layed out in 'row major' ordering. In other words, the first 3 values in the array correspond to the first row of the matrix, the second three values correspond to the second row of the matrix, and the third triplet with the third row.

getAccelerometerRange(*self*)

Read accelerometer sensitivity range. Higher ranges can detect and report larger accelerations, but are not as accurate for smaller ones.

Return Value

An integer representing the intended range for the accelerometer. Expected values are:

- 0 for 2G
- 16 for 4G
- 48 for 8G.

getMultiRefWeightPwr(*self*)

Get the weighting power for multi reference vector weights. Multi reference vector weights are all raised to the weight power before they are summed and used in the calculation for the final reference vector. Setting this value nearer to 0 will cause the reference vectors to overlap more, and setting it nearer to infinity will cause the reference vectors to influence a smaller set of orientations.

Return Value

A float representing the weight power to use.

getMultiRefResolution(*self*)

Reads number of cell divisions and number of nearby vectors per cell for the multi reference lookup table.

By default, the multiple reference vector mode only deals with orientations obtainable by successive rotations of 90 degrees about any of the three axes. This command can adjust it to accept orientations obtainable by 45 degree rotations. For 90 degrees, give a "number of cell divisions" of 4, and for 45 give 8. In addition, the number of vectors near to any given orientation which the scheme will check can be adjusted as well. If this value is 0, only the nearest orientation will be checked. The largest this value can be is 32. Also note that the larger this value is, the longer the multiple reference vector mode will take to run each cycle.

Return Value

Two integers in a list. The first representing the number of cell divisions. The second representing the number of nearby vectors.

getNumMultiRefCells(*self*)

Read number of multi reference cells.

Return Value

An integer expressing the number of multi reference cells.

getFilterMode(*self*)

Reads the current orientation filter mode. Manipulating the filer mode can be useful for tuning filter-performance versus orientation-update rates. When using the sensor as an IMU, it will improve performance to disable the orientation filter. When using as an AHRS, where orientation is needed, full-Kalman, gyroless-filtered, or fast-filtered should be used.

Return Value

An integer whose value expresses the sensor filter's mode. Possible values are:

- 0 - Disables filter
- 1 - Full Kalman Filter
- 2 - Gyroless Filtered Orientation (Not Yet Supported)
- 3 - Fast Filtered Orientation (Not Yet Supported)

None is returned if there were any issues communicating with the sensor.

getRunningAverageMode(*self*)

Reads the mode that the running-average method uses. 'Normal' uses a static running-average. 'Confidence' uses a running average that changes dynamically based upon the confidence factor.

Return Value

An integer whose value expresses the mode to set the sensor's running average to. Possible values are:

- 0 - Normal mode
- 1 - Confidence Mode

None is returned if there were any issues communicating with the sensor.

getGyroscopeRange(*self*)

Sets the measurment range used by the gyroscope sensor.

Return Value

An integer whose value expresses the range mode to set the gyroscope to. Possible values are:

- 0 - +/- 250dps mode
- 1 - +/- 500dps mode
- 2 - +/- 2000dps mode

None is returned if there were any issues communicating with the sensor.

getCompassRange(*self*)

Sets the measurement range used by the compass sensor.

Return Value

An integer whose value expresses the range mode to set the compass to. Possible values are:

- 0 - +/- 0.88Ga mode
- 1 - +/- 1.30Ga mode
- 2 - +/- 1.90Ga mode
- 3 - +/- 2.50Ga mode
- 4 - +/- 4.00Ga mode
- 5 - +/- 4.70Ga mode
- 6 - +/- 5.60Ga mode
- 7 - +/- 8.10Ga mode

None is returned if there were any issues communicating with the sensor.

setCompassCalibrationParam(*self*, *mat*, *bias*)

Sets the compass calibration parameters to the given values. These consist of a bias which is applied to the raw data as a translation, and a matrix by which the value is multiplied by.

Parameters

`mat:` A list of 9 floats expressing the matrix to multiply the raw data by. The list is a 1D array listing the elements of the matrix is a 'row major' format *(The first 3 elements of the list are the first row of the matrix, the second 3 elements are the second row, and the third triplet is the last row).*

`bias:` A list of 3 floats. The list represents a vector indicating the bias of the calibration. The vector should be formatted as such: *(x, y, z).*

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setAccelerometerCalibrationParam(*self*, *mat*, *bias*)

Sets the accelerometer calibration parameters to the given values. These consist of a bias which is applied to the raw data as a translation, and a matrix by which the value is multiplied by.

Parameters

`mat:` A list of 9 floats expressing the matrix to multiply the raw data by. The list is a 1D array listing the elements of the matrix is a 'row major' format *(The first 3 elements of the list are the first row of the matrix, the second 3 elements are the second row, and the third triplet is the last row).*

`bias:` A list of 3 floats. The list represents a vector indicating the bias of the calibration. The vector should be formatted as such: *(x, y, z).*

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getCompassCalibrationParam(*self*)

Read compass calibration parameters Bias and 3x3 Matrix.

Return Value

A list of 12 floats. The first 3 values in the list represent a vector indicating the bias of the calibration. The vector is formatted as such: *(x, y, z)*. The next 9 floats express the matrix to multiply the raw data by. The list is a 1D array listing the elements of the matrix is a 'row major' format *(The first 3 elements of the list are the first row of the matrix, the second 3 elements are the second row, and the third triplet is the last row).*

getAccelerometerCalibrationParam(*self*)

Read accelerometer calibration parameters Bias and 3x3 Matrix.

Return Value

A list of 12 floats. The first 3 values in the list represent a vector indicating the bias of the calibration. The vector is formatted as such: i{(x, y, z)}. The next 9 floats express the matrix to multiply the raw data by. The list is a 1D array listing the elements of the matrix is a 'row major' format *(The first 3 elements of the list are the first row of the matrix, the second 3 elements are the second row, and the third triplet is the last row).*

getGyroCalibrationParam(*self*)

Read gyro calibration parameters. These consist of a bias which is applied to the raw data as a translation, and a matrix by which the value is multiplied by.

Return Value

A list of 12 floats. The first 3 floats are a vector representing the bias parameter. The second 9 floats represent a 3x3 rotation matrix that all raw data is multiplied by in row-major form.

calibrateGyro(*self*)

Puts the sensor in gyro calibration mode. It will collect a few frames of data from the gyro to determine its bias. It will return to normal operation after this or if the sensor is reset.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setGyroCalibrationParam(*self*, *mat*, *bias*)

Sets the gyro calibration parameters. These consist of a bias which is applied to the raw data as a translation, and a matrix by which the value is multiplied by.

Parameters

`mat:` A list of 9 floats. The list is a 3x3 rotation matrix and is formatted in row-major.

`bias:` A list of 3 floats. The list is a vector and formatted as such: *(x, y, z)*.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setLookupTblVertVal(*self*, *type*, *idx*, *val*)

Set a vector entry in a 3-Space Sensor's vertex look-up table. This function is an experimental calibration function that is not guaranteed to be supported.

Parameters

`type:` An integer expressing the type of on-board sensor data to log. Possible values are:

- 0 - Compass
- 1 - Accelerometer

`idx:` An integer expressng the index into the lookup table to write to. The value may range from 0 to 1352.

`val:` A normalized Vector3 (list of 3 floats) to store in the table.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getLookupTblVertVal(*self*, *type*, *idx*)

Reads a vector entry in a 3-Space Sensor's vertex look-up table. This function is an experimental calibration function that is not guaranteed to be supported.

Parameters

`type`: An integer expressing the type of on-board sensor data to log. Possible values are:

- 0 - Compass
- 1 - Accelerometer

`idx`: An integer expressng the index into the lookup table to write to. The value may range from 0 to 1352.

Return Value

A normalized Vector3 (list of 3 floats) stored at idx in the table. None is returned if there was an issue communicating with the sensor.

restoreFactorySettings(*self*)

Restores all settings to factory settings. The settings are not committed to non-volatile memory by this command, so the commit settings command will have to be used if this is desired.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

commitSettings(*self*)

Commits settings to non-volatile memory. This will cause them to persist even if the sensor is reset.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

softwareReset(*self*)

Resets the sensor without powering it down. Any settings saved to non-volatile memory will be restored and the sensor. During the reset, the sensor will be inactive for approximatly 5 seconds. Certain setting changes only take place when a reset has occurred.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

enableWatchdogTimer(*self*, *timeout*)

Enables the watchdog timer with the given timeout rate. If a frame takes more than this amount of time, the sensor will automatically reset. This is useful for dealing with sensor hangs or crashes, as the sensor would reset and continue normal operation.

Parameters

`timeout`: An integer indicating the length of time in microseconds to wait before timing out when the watchdog timer is active.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

disableWatchdogTimer(*self*)

Disable the watchdog timer.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

enterFirmwareUpdateMode(*self*)

Puts the sensor into firmware update mode. This will cease normal operation until the firmware update mode is instructed to return the sensor to normal operation.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setUARTRate(*self*, *rate*)

Sets the baud rate of the physical UART. This setting does not need to be committed, but does not take effect until the sensor is reset. The baud rate will be set to the valid value nearest the requested value.

Parameters

`rate`: An integer whose value is the desired baud rate of the UART.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getUARTRate(*self*)

Reads the baud rate of the physical UART. If the UART rate has been changed, but the sensor has not been reset, the returned rate will reflect the newly set rate as opposed to the current operating rate.

Return Value

An integer whose value is the actual, current baud rate for the physical UART.

setUSBMode(*self*, *mode*)

Sets the communication mode for USB. All modes present a COM port with which to communicate with the USB device. FTDI and CDC each present a regular numbered port, whereas Unique presents a port named YEI_<serial number>. The sensor will change modes immediately.

Parameters

`mode`: An integer whose value represents the desired USB mode for the sensor. Valid values are: 2 for Unique mode, 1 for FTDI mode, and 0 for CDC mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getUSBMode(*self*)

Gets the communication mode for USB. All modes present a COM port with which to communicate with the USB device. FTDI and CDC each present a regular numbered port, whereas Unique presents a port named YEI_<serial number>. The sensor will change modes immediately.

Return Value

An integer whose value represents the current USB mode for the sensor. Expected values are: 2 for Unique mode, 1 for FTDI mode, and 0 for CDC mode.

setClockSpeed(*self*, *speed*)

Sets the clock speed to the given value. It has an upper limit of 60 MHz and a lower limit of 10 MHz. This setting does not need to be committed, but does not take effect until the sensor is reset.

Parameters

`speed:` An integer that expresses the desired clock speed of the MCU in Hz. As an example, if one desired to set the clock speed to 10 MHz, the speed argument should be set to 10000000.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getClockSpeed(*self*)

Reads the clock speed of the sensor's MCU. It has an upper limit of 60 MHz and a lower limit of 10 MHz. This setting does not need to be committed, but does not take effect until the sensor is reset.

Return Value

An integer that expresses the current clock speed of the MCU in Hz. As an example, if current clock speed of the MCU was 10 MHz, the return value from this call be 10000000.

getSerialNumber(*self*)

Reads the serial number of the sensor. The serial number is a unique number for any one sensor.

Return Value

An integer whose value is the serial number of the sensor.

setLEDColor(*self*, *color*)

Sets the color of the LED on the sensor to the given RGB color.

Parameters

`color:` A tuple of 3 floats. Each float has an expected range of 0.0 (no color contribution) to 1.0 (greatest color contribution). The first float is the red channel, the second the green channel, and the third float is the blue channel. *(red, green, blue)*

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getLEDColor(*self*)

Gets the color of the LED on the sensor as an RGB tuple.

Return Value

A tuple of 3 floats. Each float has an expected range of 0.0 (no color contribution) to 1.0 (greatest color contribution). The first float is the red channel, the second the green channel, and the third float is the blue channel. *(red, green, blue)*

setJoystickEnabled(*self*, *is_enabled*)

Turns the data feed to the joystick on and off. If disabled, the sensor will still enumerate as a joystick, but the joystick will not function. This allows normal communication to occur at a faster rate.

Parameters

`is_enabled`: A boolean. If True, the joystick is enabled. If False, the joystick is disabled.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setMouseEnabled(*self*, *is_enabled*)

Turns the data feed to the mouse on and off. If disabled, the sensor will still enumerate as a mouse, but the mouse will not function. This allows normal communication to occur at a faster rate.

Parameters

`is_enabled`: A boolean. If True, the mouse is enabled. If False, the mouse is disabled.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getJoystickEnabled(*self*)

Read whether the data feed to the joystick on or off. If disabled, the sensor will still enumerate as a joystick, but the joystick will not function. This allows normal communication to occur at a faster rate.

Return Value

A boolean. If True, the joystick is enabled. If False, the joystick is disabled.

getMouseEnabled(*self*)

Read whether the data feed to the mouse on or off. If disabled, the sensor will still enumerate as a mouse, but the mouse will not function. This allows normal communication to occur at a faster rate.

Return Value

A boolean. If True, the mouse is enabled. If False, the mouse is disabled.

setControlMode(*ctrl_class*, *ctrl_idx*, *hdl_idx*)

setControlData(*ctrl_class*, *ctrl_idx*, *data_idx*, *data*)

getControlMode(*ctrl_class*, *ctrl_idx*)

getControlData(*ctrl_class*, *ctrl_idx*, *data_idx*)

setButtonGyroDisableLength(*self*, *length*)

Sets how long, in frames, the gyros should be disabled after one of the physical buttons on the sensor is pressed. This setting helps to alleviate gyro disturbances caused by the buttons causing small shockwaves in the sensor.

Parameters

`length`: An integer whose value expresses how many frames to disable the gyros for when a button is pressed. A setting of 0 means the gyros won't be disabled at all.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getButtonGyroDisableLength(*self*)

Returns how long, in frames, the gyros should be disabled after one of the physical buttons on the sensor is pressed. This setting helps to alleviate gyro disturbances caused by the buttons causing small shockwaves in the sensor.

Return Value

An integer whose value expresses how many frames to disable the gyros for when a button is pressed. A setting of 0 means the gyros won't be disabled at all.

getButtonState(*self*)

Reads the current state of the sensor's physical buttons. Designation of 'left' and 'right' assumes the sensor to be oriented such that the LED side of the sensor is facing up and the side of the sensor that contains the USB port is facing towards the user.

Return Value

Two boolean values representing the states of the two physical sensor buttons. A value of True indicates that the button is being pressed. A value of False indicates the button is not currently being pressed. The first boolean is the 'left' button's state and the second boolean is the 'right' button's state. If there is a problem with communication, None is returned.

setMouseRelative(*self*, *is_relative*)

Puts the mouse in absolute or relative mode. Please note that this change does not take effect immediately, and the sensor's settings must be committed and the sensor must be reset before the mouse will enter this mode.

Parameters

`is_relative`: A boolean whose value expressed whether the mouse is to be relative or not. If the value is True, the mouse is relative. If the value is False, the mouse is in absolute mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getMouseRelative(*self*)

Reads the current mouse absolute/relative state. Note that if the sensor has not been reset since it has been put in this state, the mouse will not reflect this change yet, even though this command will.

Return Value

A boolean whose value expressed whether the mouse is relative or not. If the value is True, the mouse is relative. If the value is False, the mouse is in absolute mode.

setJoystickMousePresent(*self*, *is_joy_present*, *is_mouse_present*)

Sets whether the joystick and mouse are present or removed. If removed, they will not show up as devices on the target system at all. For these changes to take effect, the sensor driver may need to be reinstalled.

Parameters

`is_joy_present:` A boolean whose value expresses whether the joystick is present on the system at all. If True, the joystick is present. If False, the joystick is not present.

`is_mouse_present:` A boolean whose value expresses whether the mouse is present on the system at all. If True, the mouse is present. If False, the mouse is not present.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getJoystickMousePresent(*self*)

Gets whether the joystick and mouse are present or removed. If removed, they will not show up as devices on the target system at all. For these changes to take effect, the sensor driver may need to be reinstalled.

Return Value

A list of Two boolean values that express whenther the joystick and mouse are present or not. The first value expresses whether the joystick is present on the system at all. If True, the joystick is present. If False, the joystick is not present. The second value expresses whether the mouse is present on the system at all. If True, the mouse is present. If False, the mouse is not present.

setGlobalAxis(*self*, *joy_or_mouse*, *device_axis*, *local_axis*, *global_axis*, *deadzone*, *scale*, *power*)

Sets an axis of the desired emulated input device as a 'Global Axis' style axis. Axis operating under this style use a reference vector and a consitent local vector to determine the state of the device's axis. As the local vector rotates, it is projected onto the global vector. Once the distance of that projection on the global vector exceeds the inputted "deadzone", the device will begin tranmitting non-zero values for the device's desired axis.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`device_axis:` A string whose value may be either 'X' or 'Y'. This string defines what axis of the device is to be configured.

`local_axis:` A list of 3 Floats whose value is a normalized Vector3. This vector represents the sensor's local vector to track.

`global_axis:` A list of 3 Floats whose value is a normalized Vector3. This vector represents the global vector to project the local vector onto (should be orthoginal to the local vector).

`deadzone:` A float that defines the minimum distance necessary for the device's axis to read a non-zero value.

`scale:` A float that defines the linear scale for the values being returned for the axis.

`power:` A float whose value is an exponental power used to further modify data being returned from the sensor.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setScreenPointAxis(*self*, *joy_or_mouse*, *device_axis*, *dist_from_screen*, *dist_on_axis*, *collision_component*, *sensor_dir*, *button_halt*)

Sets an axis of the desired emulated input device as a 'Screen Point Axis' style axis. An axis operating under this style projects a vector along the sensor's direction vector into a mathmatical plane. The collision point on the plane is then used to determine what the device's axis's current value is. The direction vector is rotated based on the orientation of the sensor.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`device_axis:` A string whose value may be either 'X' or 'Y'. This string defines what axis of the device is to be configured.

`dist_from_screen:` A float whose value is the real world distance the sensor is from the user's screen. Must be the same units as dist_on_axis.

`dist_on_axis:` A float whose value is the real world length of the axis along the user's screen (width of screen for x-axis, height of screen for y-axis). Must be the same units as dist_from_screen.

`collision_component:` A string whose value may be 'X', 'Y', or 'Z'. This string defines what component of the look vector's collision point on the virtual plane to use for manipulating the device's axis.

`sensor_dir:` A string whose value may be 'X', 'Y', or 'Z'. This string defines which of the sensor's local axis to use for creating the vector to collide with the virtual plane.

`button_halt:` A float whose value is a pause time in milliseconds. When a button is pressed on the emulated device, transmission of changes to the axis is paused for the inputted amount of time to prevent undesired motion detection when pressing buttons.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

disableAxis(*self*, *joy_or_mouse*, *device_axis*)

Disables an axis on the passed in device.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`device_axis:` A string whose value may be either 'X' or 'Y'. This string defines what axis of the device is to be configured.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setPhysicalButton(*self*, *joy_or_mouse*, *but_idx*, *button_bind*)

Binds a sensor's physical button to an emulated device's button.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`but_idx:` An integer whose value is in the range 0 - 7. This value defines which button on the emulated device to configure.

`button_bind:` A string whose value may be 'left' or 'right'. This string defines which physical button to bind to the emulated device's button to as defined by but_idx. Designation of 'left' and 'right' assumes the sensor to be oriented such that the LED side of the sensor is facing up and the side of the sensor that contains the USB port is facing towards the user.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setShakeButton(*self*, *joy_or_mouse*, *but_idx*, *threshold*)

Sets up an emulated device's button such that it is 'pressed' when the sensor is shaken.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`but_idx:` An integer whose value is in the range 0 - 7. This value defines which button on the emulated device to configure.

`threshold:` A float whose value defines how many Gs of force must be experienced by the sensor before the button is 'pressed'.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setOrientationButton(*self*, *joy_or_mouse*, *but_idx*, *local_axis*, *global_axis*, *max_dist*)

Sets up a device's button such that it is 'pressed' when a reference vector aligns itself with a local vector.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`but_idx:` An integer whose value is in the range 0 - 7. This value defines which button on the emulated device to configure.

`local_axis:` A list of 3 floats whose value represents a normalized Vector3. This vector represents the sensor's local vector to track.

`global_axis:` A list of 3 floats whose value is a normalized Vector3. This vector represents the global vector to move the local vector towards for "pressing" (should not be colinear to the local vector).

`max_dist:` A float whose value defines how close the local vector's orientation must be to the global vector for the button to be 'pressed'.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

disableButton(*self*, *joy_or_mouse*, *but_idx*)

Disables a button on the passed in emulated device.

Parameters

`joy_or_mouse:` A string whose value may be either 'mouse' or 'joystick'. This string defines whether the device in question is a mouse or joystick.

`but_idx:` An integer whose value is in the range 0 - 7. This value defines which button on the emulated device to configure.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setupSimpleMouse(*self*, *dist_from_screen*, *diag_size*, *aspect_ratio*, *is_relative*=`True`)

Creates a simple emulated mouse device using the features of the sensor. Left button and right button emulate the mouse's left and right buttons respectivly and using the sensor as a pointing device with the front of the device facing towards the screen will move the mouse cursor.

Parameters

`dist_from_screen:` A float whose value is the real world distance the sensor is from the user's screen. Must be the same units as diag_size.

`diag_size:` A float whose value is the real world diagonal size of the user's screen. Must be the same units as dist_from_screen.

`aspect_ratio:` A float whose value is the real world aspect ratio of the user's screen. For instance, if the user had a screen whose aspect ratio was 16:9, the value of this argument should then be 16.0 / 9.0, or 1.777.

`is_relative:` A boolean whose value expresses whether the mouse is to operate in relative mode (True) or absolute mode (False).

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setupSimpleJoystick(*self*, *deadzone*, *scale*, *power*, *shake_threshold*, *max_dist*)

Creates a simple emulated joystick device using the features of the sensor. The left and right physical buttons on the sensor act as buttons 0 and 1 for the joystick. Button 2 is a shake button. Buttons 3 and 4 are pressed when the sensor is rotated +-90 degrees on the Z-axis. Rotations on the sensor's Y and X axis correspond to movements on the joystick's X and Y axis.

Parameters

`deadzone:`	A float that defines the minimum distance necessary for the device's axis to read a non-zero value.
`scale:`	A float that defines the linear scale for the values being returned for the axis.
`power:`	A float whose value is an exponental power used to further modify data being returned from the sensor.
`shake_threshold:`	A float whose value defines how many Gs of force must be experienced by the sensor before the button 2 is 'pressed'.
`max_dist:`	A float whose value defines how close the local vector's orientation must be to the global vector for buttons 3 and 4 are "pressed".

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setupSimpleLightgun(*self*, *dist_from_screen*, *diag_size*, *aspect_ratio*, *is_relative*=`True`)

Creates a simple emulated mouse based lightgun device using the features of the sensor. Left button of the sensor emulates the mouse's left button. Pointing the sensor 'Up' pressed the mouse's right button. This configuration uses the sensor as a pointing device with the front of the device facing forward the screen will move the mouse cursor.

Parameters

`dist_from_screen:` A float whose value is the real world distance the sensor is from the user's screen. Must be the same units as diag_size.

`diag_size:` A float whose value is the real world diagonal size of the user's screen. Must be the same units as dist_from_screen.

`aspect_ratio:` A float whose value is the real world aspect ratio of the user's screen. For instance, if the user had a screen whose aspect ratio was 16:9, the value of this argument should then be 16.0 / 9.0, or 1.777.

`is_relative:` A boolean whose value expresses whether the mouse is to operate in relative mode (True) or absolute mode (False).

Return Value

True if the command was successfuly written to the device. False if the command was not written.

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

10.2 Properties

Name	Description
Inherited from object	
__class__	

11 Class threespace_api.TSWLSensor

The TSWLSensor class is an interface layer for 3-Space Sensor Wireless units that communicate through the USB port or a 3-Space Sensor Dongle. Functions available through this class are either pythonic versions of the sensor's available commands or convenience functions for determining the status of the sensor's connection.

Attributes:

- **port_name**: A string storing the name of the serial port active communication occurs on. The name of the port is determined by the operating system.
- **read_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting for a response when reading before giving up.
- **write_timeout**: A float expressing the amount of time (in seconds) the serial port blocks, waiting to complete writting before giving up.
- **baudrate**: An integer expressing the data transmission rate of the serial port.
- **serial_port**: A PySerial Serial object instance. This is the serial port that all communication with the sensor takes place on.
- **friendly_name**: A python string storing the OS assigned human readable description of the serial port.
- **is_asynchronous**: A boolean that whose value indicates whether the wireless sensor is asynchronously transmitting or not. A value of True means that the sensor is currently transmitting asynchronously. A value of False indicates that the sensor is not communicating asynchronously. This variable is protected by thread locks and all queries for asynchronous status should be attained by a call to isAsynchronous.
- **asynch_bool_lock**: A threading lock that protects the is_asynchronous variable from corruption due to threaded access.
- **last_asynch_time**: A float whose value is a time stamp of the last successfuly recieved asynchronous data packet.
- **parse_func**: A function reference used for parsing binary data recieved from an asynchronous packet. The function getAsynchronousData uses this function to convert the binary data to python formatted data.
- **call_lock**: A threading lock that prevents mutliple threads from attempting to call 3-Space Sensor Wireless commands at once. This lock is set up such that if multiple threads attempt to send commands, the first command to be called will guarentee complete writing and return of any relevant data before

the next command is sent.

- **dongle**: A reference to the dongle this wireless sensor is currently communicating through.
- **address**: The logical ID that this sensor is listed under in the dongle's wireless hardware table. This value is used in the construction of serial commands sent to the dongle.
- **device_type**: A string expressing the type of 3-Space Sensor associated with the class instance. "WL_W" is the value of this variable, indicating that it is a wireless sensor communicating wirelessly.
- **send_retries**: An integer expressing the number of times a send or other state-pertinent command will be attempted (and fail) before notifying the caller that the command has failed.
- **serial_number**: An integer that stores the serial number of the associated 3-Space Sensor Wireless.
- **serial_number_hex**: A string that stores the hex number of the serial number.
- **cur_asynch_start_time**: The starting time of the last asynchronous command issued. This is used for estimating when a timed asynchronous session has ended.
- **cur_asynch_duration**: The duration parameter of the last asynchronous command issued. This is used for estimating when a timed asynchronous session has ended.
- **last_timestamp**: A float that denotes the timestamp of the last asynchronous data received.
- **last_data**: A reference to the last asynchronous data received.
- **error**: A TSSError instance that keeps track of the last known error the class encountered.

11.1 Methods

__new__(*cls*, *dongle*=`None`, *address*=`None`, *serial_number*=`None`, *axis_dir*=`None`, *com_port*=`None`, *read_timeout*=`2`, *write_timeout*=`2`, *baudrate*=`115200`)

Return Value

a new object with type S, a subtype of T

Overrides: object.__new__ extit(inherited documentation)

__init__(*self*, *dongle*=`None`, *address*=`None`, *serial_number*=`None`, *axis_dir*=`None`, *com_port*=`None`, *read_timeout*=`2`, *write_timeout*=`2`, *baudrate*=`115200`)

Intializes the 3-Space Wireless sensor.

Parameters

`dongle:` A reference to a TSDongle object. The wireless sensor will be communicating through this object's serial port.

`address:` An integer whose value is the logical ID that this sensor is listed under in the dongle's wireless hardware table. This value is used in the construction of serial commands sent to the dongle.

`serial_number:` An optional integer whose value is the serial number of the wireless sensor associated with this class. The default value of this argument is None. If None, the function will call the sensor's getSerialNumber command and get the serial number dynamicaly.

`axis_dir:` An optional tuple argument that expresses a default axis directions configuration for the sensor. The first value of the tuple is a string expressing the ordering of the axis. The second value is a boolean expressing the negation of the X-Axis, the third value is a boolean expressing the negation of the Y-Axis, and the fourth value is a boolean expressing the negation of the Z-Axis. A value of True for any of the booleans indicates that the corresponding axis is to be negated. A value of False for any of the booleans indicates the axis is not to be negated. Valid string values for the first tuple value are:

- 'XYZ'
- 'XZY'
- 'YXZ'
- 'YZX'
- 'ZXY'
- 'ZYX'

The default value is None, indicating that no default axis direction should be set. *(axis_order_string, neg_x_bool, neg_y_bool, neg_z_bool)*

`com_port:` A string storing the name of the serial port intended for communication. The name of the port is dependent of the operating system.

`read_timeout:` An optional float argument expressing the amount of time (in seconds) the serial port blocks, waiting

wirelessClose(*self*)

When called, the sensor removes itself from the dongle's list of sensors and removes its reference to the dongle. Any reference to this object should be deleted once close is called, as the object is no longer useful.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

wirelessRead(*self*, *read_amount*)

Returns the latest binary data recieved from the wireless sensor.

Parameters

`read_amount`: A placer argument that does not get used. Only present for matching the prototype with the TSSensor's read().

Return Value

A binary string representing the bytes read from the serial port. If an error occurs, an empty string is returned.

wirelessWrite(*self*, *command_str*)

Writes a binary command to the wireless sensor via the sensor's dongle.

Parameters

`command_str`: A string of binary data to be written to the wireless sensor.

Return Value

A boolean expressing the success of the command. If True, the command was successfuly sent. If False, the command failed to transmit.

wirelessIsConnected(*self*, *try_reconnect*=`False`)

Tests if the wireless sensor is available for communication. This testing also includes checking if the dongle associated with the sensor is connected.

Parameters

`try_reconnect`: An optional boolean that when True, an attempt at reconnection is made if the sensor's dongle is not connected. The default value is False.

Return Value

True is returned if the sensor is available for communication. False is returned if either the sensor's dongle is not connected or the wireless sensor is not capable of recieving/transmitting wireless packets.

wirelessReconnect(*self*)

Attempts to establish a functioning connection to the dongle on the serial port if it is not connected.

Return Value

True is returned if the dongle is now connected. False is returned if the serial port could not be reconnnected.

getPanID(*self*)

Returns the PanID of the sensor (if it is a wireless one).

Return Value

An integer whose value is the PanID for the sensor is returned.

setPanID(*self*, *id*)

Set the sensor's PanID (if it is a wireless sensor). Only devices with the same PanID will be able to communicate with it. For wireless sensors, this command will only work when issued to a device plugged in via USB.

Parameters

`id:` An integer whose value is the intended PanID.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getChannel(*self*)

Return this sensor's wireless channel (if a wireless sensor).

Return Value

An integer whose range is from 11-26 (decimal), inclusive. This value is the wireless channel.

setChannel(*self*, *channel*)

Sets this sensor's wireless channel (if a wireless sensor) to the passed in value. Only devices on the same channel will be able to communicate.

Parameters

`channel:` An integer whose value is in the range 11-26 (decimal), inclusive. This value is the channel to assign to the sensor. An attempt to set the channel to a value outside this range will have no effect.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

setLEDMode(*self*, *mode*)

Sets the mode of the sensor's LED (if wireless). The LED has two possible mode, 'static' and standard.

- If the LED is in 'static' mode, this means that it will only display the color set by the command setLEDColor.
- If the LED is in 'standard' mode, it will display the standard LED colors as described below:
  - Upon receipt of a packet, the wireless unit will flash green temporarily. This will occur regardless of whether the wireless unit is plugged in or not.
  - When the wireless unit is plugged in and charging, the sensor will flash orange every second.
  - When the wireless unit is plugged in and fully charged, the sensor will flash green every second.
  - When the wireless unit falls below a certain battery life level, it will flash red in increasingly quicker intervals. *(Note that this does not happen if the sensor is plugged in.)*

Parameters

`mode`: An integer of two valid values. If 0, the sensor will be set to 'standard' LED mode. If 1, the sensor will be set to 'static' LED mode.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

commitWirelessSettings(*self*)

Commits wireless configuration settings to the wireless sensor's non-volitile memory. The settings committed with this command are:

- PanID
- Wireless Channel
- Wireless Address

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getWirelessAddress(*self*)

Reads the wireless address of the wireless sensor.

Return Value

If the sensor is a wireless sensor, an integer whose value is the sensor's wireless address is returned. If the sensor is a wired sensor, None is returned.

setWirelessAddress(*self*, *addr*)

Sets the wireless address of the wireless sensor.

Parameters

`addr:` An integer whose value is the desired wireless address for the wireless sensor.

Return Value

True if the command was successfuly written to the device. False if the command was not written.

getLEDMode(*self*)

Reads the mode of the sensor's LED (if wireless). The LED has two possible mode, 'static' and standard.

- If the LED is in 'static' mode, this means that it will only display the color set by the command setLEDColor.
- If the LED is in 'standard' mode, it will display the standard LED colors as described below:
  - Upon receipt of a packet, the wireless unit will flash green temporarily. This will occur regardless of whether the wireless unit is plugged in or not.
  - When the wireless unit is plugged in and charging, the sensor will flash orange every second.
  - When the wireless unit is plugged in and fully charged, the sensor will flash green every second.
  - When the wireless unit falls below a certain battery life level, it will flash red in increasingly quicker intervals. *(Note that this does not happen if the sensor is plugged in.)*

Return Value

An integer of two valid values. If 0, the sensor will be set to 'standard' LED mode. If 1, the sensor will be set to 'static' LED mode.

getBatteryVoltage(*self*)

Return the voltage of the battery as measured by the wireless sensor.

Return Value

If the sensor is a wireless sensor, a float whose value is the current voltage of the battery. If the sensor is a wired sensor, None is returned.

getBatteryLife(*self*)

Return the percentage of battery life left in the wireless sensor's battery.

Return Value

If the sensor is a wireless sensor, an integer whose value is the current percentage of the battery's life. If the sensor is a wired sensor, None is returned.

getBatteryStatus(*self*)

Return a status number indicating the state of the wireless sensor's battery.

Return Value

If the sensor is a wireless sensor, an integer whose value is the current status of the battery. 1 represents fully charged, 2 represents charging. If the sensor is a wired sensor, None is returned.

setUARTRate(*self*, *rate*)

A placer function since the parent class may perform this operation, but this class cannot. Will just return False.

Parameters

`rate`: An integer whose value is the desired baud rate of the UART.

Return Value

False

Overrides: threespace_api.TSSensor.setUARTRate

getUARTRate(*self*)

A placer function since the parent class may perform this operation, but this class cannot. Will just return None.

Return Value

None

Overrides: threespace_api.TSSensor.getUARTRate

isAsynchronous(*self*)

Returns whether the wireless sensor is communicating asynchronously or not.

Return Value

A boolean that if True means the sensor is currently asynchronously communicating. If False, then the sensor is not asynchronously communicating.

writeAsynchronous(*self*, *command_str*, *interval*, *duration*)

Writes a binary aynchronous command to the wireless sensor via the sensor's dongle.

Parameters

`command_str`: A string of binary data to be written to the wireless sensor.

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean expressing the success of the command. If True, the command was successfuly sent. If False, the command failed to transmit.

getAsynchronousData(*self*)

Returns the latest python typed data recieved from an asynchronous communication from the wireless sensor. The data recieved is also time stamped to allow for recieve delay compensation.

Return Value

A tuple of a python typed value of some sort and a float. The typing of the tuple's first value depends on the asynchronous communication currently active. The tuple's second value is a time stamp of when the data was recieved by the dongle relative to the start of the asynchronous session. A tuple of None and -1 is returned if no asynchronous packets have been recieved yet or the dongle is disconnected.

- *(asynch_data, time_stamp)* If a packet had been recieved.
- *(None, -1)* If no packet has been recieved yet or the dongle is disconnected.

stopAsynchronous(*self*)

Stops all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltTaredOrientQuat(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Tared Orientation of the sensor as a quaternion. getAsynchronousData will return data as a list of 4 floats representing the quaternion. The list will be formatted as such: *(x, y, z, w)*.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltTaredOrientEuler(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Tared Orientation of the sensor as a set of euler angles. getAsynchronousData will return data as a list of 3 floats representing the set of euler angles. The list will be formatted and order as such: *(pitch, yaw, roll)* in radians.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltTaredOrientMat(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Tared Orientation of the sensor as 3x3 matrix. getAsynchronousData will return data as a list of 9 floats representing the matrix. The list will be formatted as a 1D array containing the elements of the matrix in a 'row major' format. In other words, The first three elements of the list will be the first row of the matrix, the next three the second row, and the last triplet the third row.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltTaredOrientAxisAngle(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Tared Orientation of the sensor as an axis angle construct. getAsynchronousData will return data as a list of 4 floats representing the axis angle construct. The list will be formatted as such: *(x, y, z, angle)*, with the angle being in radians.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltTaredOrientFwdDwn(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Tared Orientation of the sensor as a pair of 3D vectors. getAsynchronousData will return data as a list of 6 floats representing a normalized 'forward' vector and a normalized 'down' vector. The list will be formatted as such: *(forward_x, forward_y, forward_z, down_x, down_y, down_z).*

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltOrientQuat(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Un-Tared Orientation of the sensor as a quaternion. getAsynchronousData will return data as a list of 4 floats representing the quaternion. The list will be formatted as such: *(x, y, z, w)*.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltOrientEuler(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Un-Tared Orientation of the sensor as a set of euler angles. getAsynchronousData will return data as a list of 3 floats representing the set of euler angles. The list will be formatted and order as such: *(pitch, yaw, roll)* in radians.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltOrientMat(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Un-Tared Orientation of the sensor as 3x3 matrix. getAsynchronousData will return data as a list of 9 floats representing the matrix. The list will be formatted as a 1D array containing the elements of the matrix in a 'row major' format. In other words, The first three elements of the list will be the first row of the matrix, the next three the second row, and the last triplet the third row.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltOrientAxisAngle(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Un-Tared Orientation of the sensor as an axis angle construct. getAsynchronousData will return data as a list of 4 floats representing the axis angle construct. The list will be formatted as such: *(x, y, z, angle)*, with the angle being in radians.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltOrientFwdDwn(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Un-Tared Orientation of the sensor as a pair of 3D vectors. getAsynchronousData will return data as a list of 6 floats representing a normalized 'forward' vector and a normalized 'down' vector. The list will be formatted as such: *(forward_x, forward_y, forward_z, down_x, down_y, down_z).*

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltTaredFwdDwnVecSensFrame(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Tared Orientation of the sensor as a pair of 3D vectors in the coordinate system defined by the sensor reference frame. getAsynchronousData will return data as a list of 6 floats representing a normalized 'forward' vector and a normalized 'down' vector. The list will be formatted as such: *(forward_x, forward_y, forward_z, down_x, down_y, down_z)*.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetFiltNorthEarthVecSensFrame(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current Filtered Orientation of the sensor relative to the global sensor reference as a pair of 3D vectors. getAsynchronousData will return data as a list of 6 floats representing a normalized 'north' vector and a normalized 'earth' vector. The list will be formatted as such: *(north_x, north_y, north_z, earth_x, earth_y, earth_z).*

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetAllSensorsNormalized(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current normalized data being read from the collection of on-board sensors on the wireless sensor. getAsynchronousData will return data as a list of 9 floats. The list is formatted such that The first three elements are the normalized data from the sensor's gyro, the second three elements are the normalized data from the sensor's accelerometer, and the last three elements are the normalized data from the sensor's compass.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetGyroNormalized(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current normalized data being read from the on-board gyroscope on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that The three elements are the normalized data from the gyro's 3 axes.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetAccelerometerNormalized(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current normalized data being read from the on-board accelerometer on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that The three elements are the normalized data from the accelerometer's 3 axes.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetCompassNormalized(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current normalized data being read from the on-board compass on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that The three elements are the normalized north vector detected by the compass.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetAccelerometerUnfiltered(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current unfiltered (but compensated) data being read from the accelerometer on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that the three elements are the unfiltered data from the sensor's accelerometer.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetCompassUnfiltered(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current unfiltered (but compensated) data being read from the compass on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that the first three elements is the unfiltered data from the sensor's compass.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetAllSensorsRaw(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current raw data being read from the collection of on-board sensors on the wireless sensor. getAsynchronousData will return data as a list of 9 floats. The list is formatted such that The first three elements are the raw data from the sensor's gyro, the second three elements are the raw data from the sensor's accelerometer, and the last three elements are the raw data from the sensor's compass.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetGyroRaw(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current raw data being read from the on-board gyroscope on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that The three elements are the raw data from the gyro's 3 axes.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetAccelerometerRaw(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current raw data being read from the on-board accelerometer on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that The three elements are the raw data from the accelerometer's 3 axes.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetCompassRaw(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current raw data being read from the on-board compass on the wireless sensor. getAsynchronousData will return data as a list of 3 floats. The list is formatted such that The three elements make up the raw un-normalized north vector being read from the compass.

Parameters

`interval:` An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration:` An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

asynchGetButtonState(*self*, *interval*, *duration*)

Starts an asynchronous session with the wireless sensor. This session will be transmitting the current state of the physical buttons on the sensor. getAsynchronousData will return data as a list of 2 booleans. A value of True indicates that the button is being pressed. A value of False indicates the button is not currently being pressed. The first boolean is the 'left' button's state and the second boolean is the 'right' button's state. Designation of 'left' and 'right' assumes the sensor to be oriented such that the LED side of the sensor is facing up and the side of the sensor that contains the USB port is facing towards the user.

Parameters

`interval`: An integer whose value expressed how long in milliseconds the wireless sensor should wait before attempting to transmit an asynchronous packet.

`duration`: An integer whose value expressed how long in milliseconds the wireless sensor should transmit asynchronous packets. Once the elapsed time since the start of the asynchronous session exceeds duration, the sensor will stop transmitting asynchronous packets on its own. If duration is 0xffff (65535 in decimal), the sensor will tranmit asynchronous packets until either the sensor is powered down or another asynchronous command is sent with a finite duration value. A duration of 0 will effectively stop all asynchronous transmissions from the wireless sensor.

Return Value

A boolean whose value indicates the success of the command. If True, the command successfuly sent. If False, the command failed to transmit.

Inherited from threespace_api.TSSensor(Section 10)

calibrateGyro(), close(), commitSettings(), disableAxis(), disableButton(), disableWatchdogTimer(), enableWatchdogTimer(), enterFirmwareUpdateMode(), getAccelerometerCalibrationParam(), getAccelerometerEnabled(), getAccelerometerNormalized(), getAccelerometerRange(), getAccelerometerRaw(), getAccelerometerUnfiltered(), getActualUpdateRate(), getAllSensorsNormalized(), getAllSensorsRaw(), getAvgPercent(), getAxisDirections(), getButtonGyroDisableLength(), getButtonState(), getClockSpeed(), getCompassCalibrationParam(), getCompassEnabled(), getCompassNormalized(), getCompassRange(), getCompassRaw(), getCompassUnfiltered(), getConfidence(), getControlData(), getControlMode(), getDesiredUpdateRate(), getFiltGyroRate(), getFiltNorthEarthVecSensFrame(), getFiltOrientAxisAngle(), getFiltOrientEuler(), getFiltOrientFwdDwn(), getFiltOrientMat(),

getFiltOrientQuat(), getFiltTaredFwdDwnVecSensFrame(), getFiltTaredOrientAxisAngle(), getFiltTaredOrientEuler(), getFiltTaredOrientFwdDwn(), getFiltTaredOrientMat(), getFiltTaredOrientQuat(), getFilterMode(), getGyroCalibrationParam(), getGyroEnabled(), getGyroNormalized(), getGyroRaw(), getGyroscopeRange(), getJoystickEnabled(), getJoystickMousePresent(), getKalmanMat(), getLEDColor(), getLastError(), getLookupTblVertVal(), getMouseEnabled(), getMouseRelative(), getMultiRefChkVecAccelerometer(), getMultiRefChkVecCompass(), getMultiRefResolution(), getMultiRefVecAccelerometer(), getMultiRefVecCompass(), getMultiRefWeightPwr(), getNumMultiRefCells(), getOversampleRate(), getRefVecAccelerometer(), getRefVecCompass(), getRefVecMode(), getRhoDataAccelerometer(), getRhoDataCompass(), getRunningAverageMode(), getSerialNumber(), getTareOrientMat(), getTareOrientQuat(), getTemperatureCelsius(), getTemperatureFahrenheit(), getUSBMode(), isConnected(), read(), reconnect(), resetKalmanFilter(), restoreFactorySettings(), setAccelerometerCalibrationParam(), setAccelerometerEnabled(), setAccelerometerRange(), setAvgPercent(), setAxisDirections(), setButtonGyroDisableLength(), setClockSpeed(), setCompassCalibrationParam(), setCompassEnabled(), setCompassRange(), setConfidenceRhoModeAccel(), setConfidenceRhoModeCompass(), setControlData(), setControlMode(), setFilterMode(), setGlobalAxis(), setGyroCalibrationParam(), setGyroEnabled(), setGyroscopeRange(), setJoystickEnabled(), setJoystickMousePresent(), setLEDColor(), setLookupTblVertVal(), setMouseEnabled(), setMouseRelative(), setMultiRefChkVecAccelerometer(), setMultiRefChkVecCompass(), setMultiRefResolution(), setMultiRefVecAccelerometer(), setMultiRefVecCompass(), setMultiRefVecZero(), setMultiRefWeightPwr(), setOrientationButton(), setOversampleRate(), setPhysicalButton(), setRefVecAccelerometer(), setRefVecCompass(), setRefVecCurrentOrient(), setRefVecMode(), setRunningAverageMode(), setScreenPointAxis(), setShakeButton(), setStaticRhoModeAccel(), setStaticRhoModeCompass(), setTareCurrentOrient(), setTareMatrix(), setTareQuaternion(), setUSBMode(), setUpdateRate(), setupSimpleJoystick(), setupSimpleLightgun(), setupSimpleMouse(), softwareReset(), write()

Inherited from object

__delattr__(), __format__(), __getattribute__(), __hash__(), __reduce__(), __reduce_ex__(), __repr__(), __setattr__(), __sizeof__(), __str__(), __subclasshook__()

11.2 Properties

Name	Description
Inherited from object	
__class__	

Index