Blatann Documentation

Release v0.5.0

Thomas Gerstenberg

Mar 02, 2023

CONTENTS:

blåtann: Norwegian word for “blue tooth”

Blatann aims to provide a high-level, object-oriented interface for interacting with bluetooth devices through python. It operates using a Nordic nRF52 through Nordic’s `pc-ble-driver-py` library and the associated Connectivity firmware for the device.

CHAPTER

ONE

GETTING STARTED

As of v0.3.0, blatann will only support Python 3.7+. v0.2.x will be partially maintained for Python 2.7 by backporting issues/bugs found in 0.3.x.

1.1 Introduction

This library relies on a Nordic nRF52 connected via USB to the PC and flashed with the Nordic Connectivity firmware in order to operate.

Note: This library will not work as a driver for any generic Bluetooth HCI USB device nor built-in Bluetooth radios. The driver is very specific to Nordic and their provided connectivity firmware, thus other Bluetooth vendors will not work. (BLE communications with non-Nordic devices is not affected.)

Below are the known supported devices:

- nRF52832 Development Kit (PCA10040)
- nRF52840 Development Kit (PCA10056)
- nRF52840 USB Dongle (PCA10059)

1.2 Install

Blatann can be installed through pip: `pip install blatann`

1.3 Running with macOS brew python

`pc-ble-driver-py` consists of a shared object which is linked to mac's system python. In order to use it with brew's python install, you'll need to run `install_name_tool` to modify the `.so` to point to brew python instead of system python.

Example shell script to do so (with more info) can be found here: macos script

1.4 Setting up Hardware

Once one of the hardware devices above is connected via USB, the Nordic Connectivity firmware can be flashed using Nordic's nRF Connect Application. There are other methods you can use (such as `nrfutil`), however this is the least-complicated way. Further instructions for using nRF Connect are out of scope for this as Nordic has great documentation for using their app already.

The firmware image to use can be found within the installed `pc-ble-driver-py` python package under the `hex/` directory. From there, it's a drag and drop operation to get the fimrware image onto the hardware.

See the *Compatibility Matrix* which lists what software, firmware, and hardware components work together.

1.5 Smoke Test the Setup

Once the hardware is flashed and Blatann is installed, the Scanner example can be executed to ensure everything is working. Blatann's examples can be executed from the command line using

```
python -m blatann.examples <example_name> <comport>
```

For the smoke test, use the `scanner` example which will stream any advertising packets found for about 4 seconds:
`python -m blatann.examples scanner <comport>`

If everything goes well, head on over to *Examples* to look at the library in action or visit *Library Architecture* to get an overview of the library. If things do not seem to be working, check out the *Troubleshooting* page.

CHAPTER

TWO

CORE CLASSES

Below are quick descriptions and links to the primary/core classes that are used to perform the various Bluetooth operations. This information, including more thorough usage, can be found within example code (*Examples*)

2.1 Events

The *Event* type is the basic building block of the blatann library.

Bluetooth operations are inherently asynchronous, thus asynchronous events must be used in order to communicate when things happen.

Almost all of the classes below implement one or more Events which can have multiple handler functions registered to process incoming data. Event properties are commonly named in the format of `on_*`, such as `on_timeout` or `on_read_complete`. The event properties also document the parameter types that the handler should accept. Majority of the events emit two parameters, a sender parameter, which provides the event source, and an event_args parameter, which provides the data associated with the event. Those familiar with C#/.NET, this should look very similar.

```
def my_handler(sender, event_args):
    # Handle the event
some_object.on_some_event.register(my_handler)
```

2.2 Waitables

Waitable, *EventWaitable*

Waitables are the solution to providing an API which supports synchronous, procedural code given the asynchronous nature of Bluetooth. For every asynchronous Bluetooth operation that is performed a `Waitable` object is returned which the user can then *wait()* on to block the current thread until the operation completes.

```
sender, event_args = characteristic.read().wait(timeout=5)
```

Note: Take care to not call `wait()` within an event handler as the system will deadlock (see Threading section under *Library Architecture* for more info).

Asynchronous paradigms are also supported through waitables where the user can register a handler to be called when the operation completes:

```
def my_characteristic_read_handler(sender, event_args):
    # Handle read complete
characteristic.read().then(my_characteristic_read_handler)
```

2.3 BLE Device

The *BleDevice* represents Nordic Bluetooth microcontroller itself. It is the root object of everything within this library.

To get started, instantiate a `BleDevice` and open it:

```
from blatann import BleDevice

ble_device = BleDevice("COM1")
ble_device.configure()
ble_device.open()
# Ready to use
```

The BLE Device is also responsible for initiating connections to peripheral devices and managing the local GATT database.

2.4 Advertising

The *Advertiser* component is accessed through the `ble_device.advertiser` attribute. It is configured using *AdvertisingData* objects to set the payloads to advertise

```
from blatann.gap.advertising import AdvertisingData
adv_data = AdvertisingData(flags=0x06, local_name="My Name")
scan_data = AdvertisingData(service_uuid16s="123F")
ble_device.advertiser.set_advertise_data(adv_data, scan_data)
ble_device.advertiser.start(adv_interval_ms=50)
```

2.5 Scanning

The *Scanner* component is accessed through the `ble_device.scanner` attribute.

The scanner output consists of a *ScanReportCollection*, which is comprised of *ScanReport* objects that represent advertising packets discovered.

```
scan_report_collection = ble_device.scanner.start_scan().wait(timeout=20)
```

2.6 Peer

The *Peer* class represents a Bluetooth connection to another device.

For connections as a peripheral to a central device, this peer object is static and accessed via the `ble_device.client` attribute. For connections as a central to a peripheral device, the peer is created as a result of *BleDevice.connect*.

Regardless of the connection type, the Peer is the basis for any connection-oriented Bluetooth operation, such as configuring the MTU, discovering databases, reading/writing characteristics, etc.

```
# Connect to a peripheral and exchange MTU
peer = ble_device.connect(peer_address).wait()
peer.exchange_mtu(144).wait()
# Exchange the MTU with a client
ble_device.client.exchange_mtu(183).wait()
```

2.7 Security

The processes for pairing and bonding is managed by a peer's *SecurityManager*, accessed via the `peer.security` attribute.

2.8 Local GATT Database

The *GattsDatabase* is accessed through the `ble_device.database` attribute. The database holds all of the services and characteristics that can be discovered and interacted with by a client.

GattsService s can be added to the database and *GattsCharacteristic* s are added to the services. The primary interaction point is through characteristics, which provides methods for setting values, handling writes, and notifying values to the client.

2.9 Remote GATT Database

The peer's *GattcDatabase* is accessed through the `peer.database` attribute. The database is populated through the *peer.discover_services* procedure. From there, the Peer's *GattcCharacteristic* s can be read, written, and subscribed to.

CHAPTER

THREE

EXAMPLES

This section is a work in progress. Still need to add specific sections giving an overview for each example

Examples can be found here: Blatann examples

CHAPTER

FOUR

COMPATIBILITY MATRIX

Table 1: Software/Firmware Compatibility Matrix

Blatann Version	Python Version Version	Connectivity Firmware Version	SoftDevice Version	pc-ble-driver-py Version
v0.2.x	2.7 Only	v1.2.x	v3	<= 0.11.4
v0.3+	3.7+	v4.1.x	v5	>= 0.12.0

Firmware images are shipped within the `pc-ble-driver-py` package under the `hex/` directory. Below maps which firmware images to use for which devices. For Blatann v0.2.x, firmware images are under subdir `sd_api_v3`. For Blatann v0.3+, firmware images are under subdir `sd_api_v5`.

Table 2: Firmware/Hardware Compatibility Matrix

Hardware	Firmware Image
nRF52832 Devkit	connectivity_x.y.z_<baud>_with_s132_x.y.hex (note the baud rate in use!)
nRF52840 Devkit	connectivity_x.y.z_<baud>_with_s132_x.y.hex (note the baud rate in use!) **or** connectivity_x.y.z_usb_with_s132_x.y.hex if using the USB port on the side
nRF52840 USB Dongle	connectivity_x.y.z_usb_with_s132_x.y.hex

Note: Blatann provides a default setting for the baud rate to use with the device. For 0.2.x, the default baud is 115200 whereas 0.3+ the default is 1M (and USB doesn't care). This is only an issue when running examples through the command line as it doesn't expose a setting for the baud rate. When writing your own script, it can be configured however it's needed.

CHAPTER

FIVE

TROUBLESHOOTING

This section is a work in progress

General Debugging

Blatann uses the built-in `logging` module to log all events and driver calls. The library also contains a helper function to configure/enable: *blatann.utils.setup_logger()*.

When submitting an issue, please include logs of the behavior at the `DEBUG` level.

Specific Error Messages

Error message `Failed to open. Error code: 0x8029` - Check your comport settings (baud, port, etc.).

Note that the nRF52840 USB dongle will enumerate 2 separate ports: one for the bootloader during flashing and one for the application. Make sure that you check the port number after the bootloader exits and application starts.

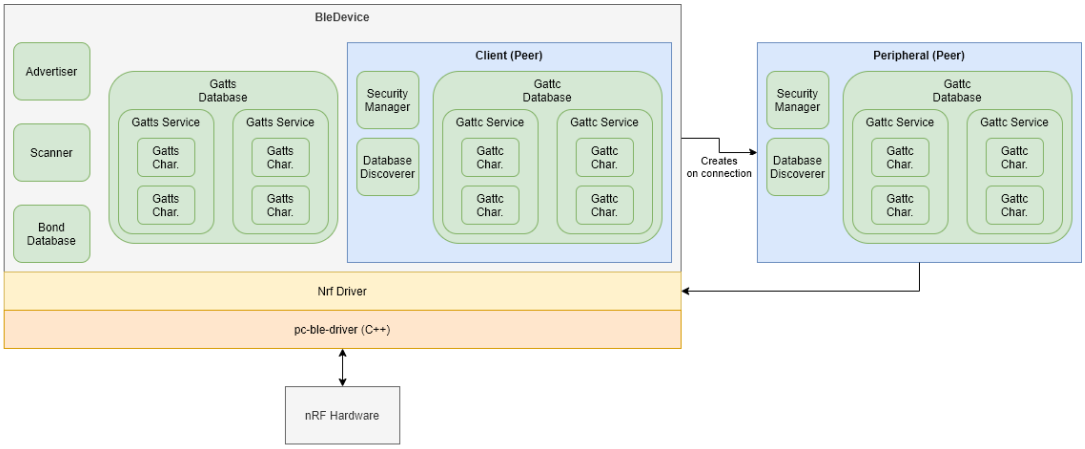
CHAPTER

SIX

LIBRARY ARCHITECTURE

6.1 Class Hierarchy (WIP)

Very high-level diagram outlining major/public components



BleDevice

The `BleDevice` class represents the Nordic hardware itself and is the primary entry point for the library. It provides the high-level APIs for all BLE operations, such as advertising, scanning, and connections. It also manages the device's configuration and bonding database.

Advertiser

The `Advertiser` class provides the API for setting advertising data, intervals, and starting/stopping of advertising. It is accessed via BleDevice.:attr:*~blatann.device.BleDevice.advertiser* attribute.

Scanner

The `Scanner` class provides the API for scanning for advertising packets. Scan reports are emitted during scanning and can be used to initiate connections with the advertising peripherals. It is accessed via BleDevice.:attr:*~blatann.device.BleDevice.scanner* attribute.

Peer

The `Peer` class represents a connection with another device over Bluetooth. The BLE Device contains a single `Client` object, which is the connection with a Client/Central device. When connecting to Server/Peripheral devices as a

Client/Central via *BleDevice.connect()*, a *Peripheral* object is created and returned as a result of the `connect()` call.

The Peer object provides everything necessary for communications with the device, including security, database creation (as a server) and discovery (as a client), connection parameters, etc.

6.2 Threading Model

Most BLE operations are inherently asynchronous. The Nordic has a combination of synchronous function calls and asynchronous events. Synchronous function calls may not return the result immediately and instead return within an asynchronous event as to not block the main context.

Blatann uses a second python thread for handling asynchronous events received over BLE. This event thread (named "<compport>Thread") handles a queue of events received from the C++ Driver and dispatches them to the registered callbacks. These callbacks are the triggers for the various *Event* objects that exist throughout the `Peer`, `Scanner`, `Advertiser`, and other objects.

In order to support both both event-driven and procedural styles of programming, a mechanism needs to exist in order to communicate events back to the main thread so asynchronous functionality (such as characteristic reads/writes) can be made synchronous. The result of this is the *Waitable* class.

Asynchronous method calls in the library will return a `Waitable` object which can either then have callbacks registered (to keep things asynchronous) or waited on (with or without timeout) from the main thread to make it synchronous. This is a very similar concept to `concurrent.futures.Future`, just a different implementation.

Since there is only a single thread which handles all events, **do not call *Waitable.wait()* within an event handler as it will cause a deadlock.** Calling BLE methods from the event handler context is perfectly fine and can use *Waitable.then(callback)* to handle the result of the operation asynchronously.

CHAPTER

SEVEN

CHANGELOG

7.1 v0.5.0

v0.5.0 reworks bonding database to JSON, adds a few features, and fixes a few bugs. Full list of issues and PRs for this release can be found here: 0.5.0 Milestone

Highlights

- Adds support for the scanner to resolve peer addresses.
  - *ScanReport* has 2 new properties: `is_bonded_device: bool` and `resolved_address: Optional[PeerAddress]`
- Adds support for privacy settings to advertise with a private resolvable or non-resolvable address
- Adds parameter to device configuration to set a different connection event length

Fixes

- Fixes incorrect variable name when a queued GATT operation times out (thanks @klow68)
- Fixes incorrect key length when converting an LESC private key to a raw bytearray (thanks @klow68). Function is unused within blatann
- Fixes issue where the service changed characteristic was not correctly getting added to the GATT server when configured

Changes

- Reworks the bond database to be saved using JSON instead of pickle.
  - Existing `"system"` and `"user"` database files configured in the BLE device will automatically be migrated to JSON
  - Other database files configured by filepath will continue to use pickle and can be updated manually using *migrate_bond_database()*
- Bond DB entries will now save the local BLE address that was used to generate the bonding data.
  - This will allow multiple nRF boards to use the same DB file and not resolve bond entries if it was not created with that board/address. This fixes potential issues where restoring a connection to a peer that was bonded to a different nRF board can cause the local device to think it has a bond, however the peer has bond info with a different, mismatched address.
- Moves bond-related resolve logic out of the security manager and into the bond database

7.2 v0.4.0

v0.4.0 introduces some new features and fixes a couple of issues. Full list of issues and PRs for this release can be found here: 0.4.0 Milestone

Highlights

- Adds support for reading RSSI of an active connection, plus example usage of API
- Adds Event+Waitable for Connection Parameter Update procedures
  - Additionally adds support for accepting/rejecting update requests as a central
- Adds support for setting the device's transmit power
- Adds support for setting advertising channel masks

Fixes

- Fixes issues seen when performing certain pairing routines on linux
- Fixes for misc. advertising corner cases
- Fixes an issue with `BasicGlucoseDatabase` introduced in the python 3 migration

7.3 v0.3.6

v0.3.6 is a minor bugfix update and some small improvements

Fixes

- Fixes an uncaught exception caused when handling a failed bond database load (thanks @dkkeller)
- Fixes an issue where waiting on indications to be confirmed did not work. Regression introduced in v0.3.4

Changes

- Updates the descriptor discovery portion of service discovery to be more efficient, speeding up service discovery times
- Updates the API lock at the driver layer to be per-device. This will reduce lock contention when using multiple BLE Devices in different threads

7.4 v0.3.5

v0.3.5 is a small update that primarily provides some bug fixes and cleanup to the bonding process.

Highlights

- Overall increased stability when restoring encryption using long-term keys for a previously-bonded device
- Adds param to set the CCCD write security level for a characteristic

Fixes

- Restoring legacy bonding LTKs as a central now works correctly

Changes

- Issue 60 - The default bonding database file has been moved into the user directory instead of within the package contents (`~/.blatann/bonding_db.pkl`).

  - An optional parameter has been added to the *BleDevice* constructor for specifying the file to use for convenience
  - To revert to the previous implementation, specify `bond_db_filename="system"` when creating the BleDevice object
  - To use the new storage location but keep the bonding data from previous version, copy over the database file from `<blatann_install_loc>/.user/bonding_db.pkl` to the location noted above

7.5 v0.3.4

v0.3.4 brings several new features (including characteristic descriptors) and a couple bug fixes. A fairly large refactoring of the GATT layer took place ot make room for the descriptors, however no public-facing APIs were modified.

Highlights

- Issue 11 - Adds support for adding descriptor attributes to characteristics
  - See the Central Descriptor Example and Peripheral Descriptor Example for how they can be used
- Adds a new `bt_sig` sub-package which provides constants and UUIDs defined by Bluetooth SIG.
- Adds visibility to the device's Generic Access Service: *BleDevice.generic_access_service*
  - Example usage has been added to the peripheral example
- Adds support for performing PHY channel updates
  - **Note**: Coded PHY is currently not supported, only 1Mbps and 2Mbps PHYs
- Adds a description attribute to the UUID class. The standard UUIDs have descriptions filled out, custom UUIDs can be set by the user.

Fixes

- Fixes an issue with bonding failing on linux
- Fixes an issue where the `sys_attr_missing` event was not being handled
- Adds missing low-level error codes for the RPC layer
- Fixes race condition when waiting on ID-based events causing an `AttributeError`. Event subscription previously occurred before the ID was set and there was a window where the callback could be triggered before the ID was set in the object instance. This issue was most prominent after introducing the write/notification queuing changes in combination with a short connection interval.

Changes

- The `device_name` parameter has been removed from *BleDevice.configure()*. This wasn't working before and has been added into the Generic Access Service.
- Write, notification, and indication queuing has been tweaked such that non-ack operations (write w/o response, notifications) now take advantage of a hardware queue independent of the acked counterparts (write request, indications)
- Service discovery was modified to allow descriptor discovery and in some cases (depending on peripheral stack) run faster
- `DecodedReadWriteEventDispatcher` has been moved from `blatann.services` to `blatann.services.decoded_event_dispatcher`. This was to solve a circular dependency issue once new features were added in.

- The glucose service has been updated to make better use of the notification queuing mechanism. Glucose record transmission is sped up greatly

7.6 v0.3.3

v0.3.3 fixes a couple issues and adds some enhancements to the security manager.

Highlights

- Adds handling for peripheral-initiated security/pairings
- Adds finer control over accepting/rejecting pairing requests based on the peer's role, whether or not it's already bonded, etc.
- Adds more events and properties to expose the connection's security state
- Adds method to delete a connected peer's bonding data for future connections

Fixes

- Fixes issue where the length of the scan response payload was not correctly being checked against the maximum 31-byte length
- Fixes issue that was not allowing central devices to initiate encryption to an already-bonded peripheral device
- Fixes issue that wasn't allowing time to be read from the Current Time service as a client

Changes

- Advertising payloads received that are padded with 0's at the end are now ignored and do not produce spammy logs
- Adds a device-level method to set the default security level to use for all subsequent connections to peripheral devices
- Adds a `name` property to the `Peer` class. This is auto-populated from the scan report (if connecting to a peripheral) and can be set manually if desired.

7.7 v0.3.2

v0.3.2 is a bug fix release

Fixes

- Issue 40 - Fixes issue where service discovery fails if the server returns `attribute_not_found` while discovering services
- Issue 42 - Fixes issue where *`Advertiser.is_advertising`* could return false if `auto_restart` is enabled and advertising times out

Added Features

- Exposes a new *`Advertiser.auto_restart`* property so it can be get/set outside of *`Advertiser.start()`*

7.8 v0.3.1

v0.3.1 provides a few enhancements and features from the previous release.

Highlights

- Adds the ability to discover, read, and write a connected central device's GATT database as a peripheral.
  - Example usage has been added to the peripheral example where it will discover the connected device's database after pairing completes
  - **NOTE:** The inverse of this should be considered experimental (i.e. acting as a central and having a peripheral read/write the local database).
- Adds the ability to perform writes without responses, both as a client and as a peripheral
  - New APIs have been added to the *GattcCharacteristic* class: *write_without_response()* and *writable_without_response*
- Adds API to trigger data length update procedures (with corresponding event) on the *Peer* class
  - The API does not allow the user to select a data length to use, i.e. the optimal data length is chosen by the SoftDevice firmware

Changes

- The connection event length has been updated to support the max-length DLE value (251bytes) at the shortest connection interval (7.5ms)
- Updates to documentation and type hinting
- Minor changes to logging, including removing spammy/duplicate logs when numerous characteristics exist in the GATT database

Fixes

- Fixes issue where iterating over the scan report in real-time was not returning the recently read packet and instead was returning the combined packet for the device's address. This was causing duplicate packets to not be marked in the scanner example.

7.9 v0.3.0

v0.3.0 marks the first stable release for Python 3.7+.

Unfortunately a comprehensive changelog is not available for this release as a lot went in to migrate to Py3/Softdevice v5. That said, public API should be mostly unchanged except for the noted changes below.

Highlights

- Python 3.7+ only
- Requires `pc-ble-driver-py` v0.12.0+
- Requires Nordic Connectivity firmware v4.1.1 (Softdevice v5)

Changes

- `Scanner.scanning` field was replaced with read-only property `Scanner.is_scanning`
- Parameter validation was added for Advertising interval, Scan window/interval/timeout, and connection interval/timeout.
  - Will raise `ValueError` exceptions when provided parameters are out of range

- With Python 3, converting from `bytes` to `str` (and vice-versa) requires an encoding format. By default, the encoding scheme is `utf-8` and can be set per-characteristic using the `string_encoding` property
- `peer.disconnect()` will now always return a `Waitable` object. Before it would return `None` if not connected to the peer. If `disconnect()` is called when the peer is not connected, it will return a Waitable object that expires immediately

Fixes

- Fixes an issue where unsubscribing from a driver event while processing the event was causing the the next handler for the driver event to be skipped
  - Back-ported to v0.2.9

Features

(This list is not comprehensive)

- Driver now property works with 2 devices simultaneously
- Event callbacks can now be used in a `with` context so the handler can be deregistered at the end of a block
  - Event callback example
- The `ScanFinishedWaitable` now provides a `scan_reports` iterable which can be used to iterate on advertising packets as they're seen in real-time
  - ScanFinishedWaitable example
- The `Peer` object now exposes properties for the active connection parameters and configured/preferred connection parameters
- The `Peripheral` object exposes an `on_service_discovery_complete` event
- Added `AdvertisingData.to_bytes()` to retrieve the data packet that will be advertised over the air

CHAPTER

EIGHT

API REFERENCE

8.1 blatann

8.1.1 blatann package

Subpackages

blatann.bt_sig package

Submodules

blatann.bt_sig.assigned_numbers module

class blatann.bt_sig.assigned_numbers.**Format**(_, *description=''*)

Bases: *IntEnumWithDescription*

Format enumeration for use with the *blatann.gatt.PresentationFormat* class

rfu = 0

boolean = 1

twobit = 2

nibble = 3

uint8 = 4

uint12 = 5

uint16 = 6

uint24 = 7

uint32 = 8

uint48 = 9

uint64 = 10

uint128 = 11

sint8 = 12

sint12 = 13

sint16 = 14

sint24 = 15

sint32 = 16

sint48 = 17

sint64 = 18

sint128 = 19

float32 = 20

float64 = 21

sfloat = 22

float = 23

duint16 = 24

utf8s = 25

utf16s = 26

struct = 27

class blatann.bt_sig.assigned_numbers.**Namespace**(_, *description=''*)

Bases: *IntEnumWithDescription*

Namespace enumeration for use with the *blatann.gatt.PresentationFormat* class

unknown = 0

bt_sig = 1

class blatann.bt_sig.assigned_numbers.**NamespaceDescriptor**(_, *description=''*)

Bases: *IntEnumWithDescription*

Namespace descriptor enumeration for use with the *blatann.gatt.PresentationFormat* class

auxiliary = 264

back = 257

backup = 263

bottom = 259

external = 272

flash = 266

front = 256

inside = 267

internal = 271

left = 269

lower = 261

main = 262

outside = 268

right = 270

supplementary = 265

top = 258

unknown = 0

upper = 260

class blatann.bt_sig.assigned_numbers.**Units**(_, *description=''*)

Bases: *IntEnumWithDescription*

Units enumeration for use with the *blatann.gatt.PresentationFormat* class

unitless = 9984

absorbed_dose_gray = 10035

absorbed_dose_rate_gray_per_second = 10068

acceleration_metres_per_second_squared = 10003

activity_referred_to_a_radionuclide_becquerel = 10034

amount_concentration_mole_per_cubic_metre = 10010

amount_of_substance_mole = 9990

angular_acceleration_radian_per_second_squared = 10052

angular_velocity_radian_per_second = 10051

angular_velocity_revolution_per_minute = 10152

area_barn = 10116

area_hectare = 10086

area_square_metres = 10000

capacitance_farad = 10025

catalytic_activity_concentration_katal_per_cubic_metre = 10071

catalytic_activity_katal = 10037

concentration_count_per_cubic_metre = 10165

concentration_parts_per_billion = 10181

concentration_parts_per_million = 10180

current_density_ampere_per_square_metre = 10008

density_kilogram_per_cubic_metre = 10005

dose_equivalent_sievert = 10036

dynamic_viscosity_pascal_second = 10048

electric_charge_ampere_hours = 10160

electric_charge_coulomb = 10023

electric_charge_density_coulomb_per_cubic_metre = 10060

electric_conductance_siemens = 10027

electric_current_ampere = 9988

electric_field_strength_volt_per_metre = 10059

electric_flux_density_coulomb_per_square_metre = 10062

electric_potential_difference_volt = 10024

electric_resistance_ohm = 10026

energy_density_joule_per_cubic_metre = 10058

energy_gram_calorie = 10153

energy_joule = 10021

energy_kilogram_calorie = 10154

energy_kilowatt_hour = 10155

exposure_coulomb_per_kilogram = 10067

force_newton = 10019

frequency_hertz = 10018

heat_capacity_joule_per_kelvin = 10054

heat_flux_density_watt_per_square_metre = 10053

illuminance_lux = 10033

inductance_henry = 10030

irradiance_watt_per_square_metre = 10166

length_foot = 10147

length_inch = 10146

length_metre = 9985

length_mile = 10148

length_nautical_mile = 10115

length_parsec = 10145

length_yard = 10144

length_angstrom = 10114

logarithmic_radio_quantity_bel = 10119

logarithmic_radio_quantity_neper = 10118

luminance_candela_per_square_metre = 10012

luminous_efficacy_lumen_per_watt = 10174

luminous_energy_lumen_hour = 10175

luminous_exposure_lux_hour = 10176

luminous_flux_lumen = 10032

luminous_intensity_candela = 9991

magnetic_field_strength_ampere_per_metre = 10009

magnetic_flux_density_tesla = 10029

magnetic_flux_weber = 10028

mass_concentration_kilogram_per_cubic_metre = 10011

mass_density_milligram_per_decilitre = 10161

mass_density_millimole_per_litre = 10162

mass_flow_gram_per_second = 10177

mass_kilogram = 9986

mass_pound = 10168

mass_tonne = 10088

metabolic_equivalent = 10169

molar_energy_joule_per_mole = 10065

molar_entropy_joule_per_mole_kelvin = 10066

moment_of_force_newton_metre = 10049

per_mille = 10158

percentage = 10157

period_beats_per_minute = 10159

permeability_henry_per_metre = 10064

permittivity_farad_per_metre = 10063

plane_angle_degree = 10083

plane_angle_minute = 10084

plane_angle_radian = 10016

plane_angle_second = 10085

power_watt = 10022

pressure_bar = 10112

pressure_millimetre_of_mercury = 10113

pressure_pascal = 10020

pressure_pound_force_per_square_inch = 10149

radiance_watt_per_square_metre_steradian = 10070

radiant_intensity_watt_per_steradian = 10069

refractive_index = 10013

relative_permeability = 10014

solid_angle_steradian = 10017

sound_pressure_decibel_spl = 10179

specific_energy_joule_per_kilogram = 10056

specific_heat_capacity_joule_per_kilogram_kelvin = 10055

specific_volume_cubic_metre_per_kilogram = 10007

step_per_minute = 10170

stroke_per_minute = 10172

surface_charge_density_coulomb_per_square_metre = 10061

surface_density_kilogram_per_square_metre = 10006

surface_tension_newton_per_metre = 10050

thermal_conductivity_watt_per_metre_kelvin = 10057

thermodynamic_temperature_degree_celsius = 10031

thermodynamic_temperature_degree_fahrenheit = 10156

thermodynamic_temperature_kelvin = 9989

time_day = 10082

time_hour = 10081

time_minute = 10080

time_month = 10164

time_second = 9987

time_year = 10163

transfer_rate_milliliter_per_kilogram_per_minute = 10167

velocity_kilometer_per_minute = 10173

velocity_kilometre_per_hour = 10150

velocity_knot = 10117

velocity_metres_per_second = 10002

velocity_mile_per_hour = 10151

volume_cubic_metres = 10001

volume_flow_litre_per_second = 10178

volume_litre = 10087

wavenumber_reciprocal_metre = 10004

class blatann.bt_sig.assigned_numbers.**Appearance**(_, *description=''*)

Bases: *IntEnumWithDescription*

Appearance enumeration for use with advertising data

unknown = 0

phone = 64

computer = 128

watch = 192

sports_watch = 193

clock = 256

display = 320

remote_control = 384

eye_glasses = 448

tag = 512

keyring = 576

media_player = 640

barcode_scanner = 704

thermometer = 768

thermometer_ear = 769

heart_rate_sensor = 832

heart_rate_sensor_heart_rate_belt = 833

blood_pressure = 896

blood_pressure_arm = 897

blood_pressure_wrist = 898

hid = 960

hid_keyboard = 961

hid_mouse = 962

hid_joystick = 963

hid_gamepad = 964

hid_digitizer = 965

hid_card_reader = 966

hid_digital_pen = 967

hid_barcode = 968

glucose_meter = 1024

running_walking_sensor = 1088

running_walking_sensor_in_shoe = 1089

running_walking_sensor_on_shoe = 1090

running_walking_sensor_on_hip = 1091

cycling = 1152

cycling_cycling_computer = 1153

cycling_speed_sensor = 1154

cycling_cadence_sensor = 1155

cycling_power_sensor = 1156

cycling_speed_cadence_sensor = 1157

pulse_oximeter = 3136

pulse_oximeter_fingertip = 3137

pulse_oximeter_wrist_worn = 3138

weight_scale = 3200

outdoor_sports_act = 5184

outdoor_sports_act_loc_disp = 5185

outdoor_sports_act_loc_and_nav_disp = 5186

outdoor_sports_act_loc_pod = 5187

outdoor_sports_act_loc_and_nav_pod = 5188

as_bytes()

blatann.bt_sig.uuids module

Bluetooth SIG defined UUIDs, populated from their website.

See also:

https://btprodspecificationrefs.blob.core.windows.net/assigned-values/16-bit%20UUID%20Numbers%20Document.pdf

Definitions last scraped on 2022/05/02

class blatann.bt_sig.uuids.**DeclarationUuid**

Bases: object

UUIDs used for declarations within the GATT Database

primary_service = 2800

secondary_service = 2801

include = 2802

characteristic = 2803

class blatann.bt_sig.uuids.**DescriptorUuid**

Bases: object

UUIDs that are used for characteristic descriptors

extended_properties = 2900

user_description = 2901

cccd = 2902

sccd = 2903

presentation_format = 2904

aggregate_format = 2905

valid_range = 2906

external_report_reference = 2907

report_reference = 2908

number_of_digitals = 2909

value_trigger_setting = 290a

es_configuration = 290b

es_measurement = 290c

es_trigger_setting = 290d

time_trigger_setting = 290e

complete_br_edr_transport_block_data = 290f

class blatann.bt_sig.uuids.ServiceUuid

Bases: object

Bluetooth SIG defined service UUIDs

alert_notification = 1811

audio_input_control = 1843

audio_stream_control = 184e

automation_io = 1815

basic_audio_announcement = 1851

battery_service = 180f

binary_sensor = 183b

blood_pressure = 1810

body_composition = 181b

bond_management = 181e

broadcast_audio_announcement = 1852

broadcast_audio_scan = 184f

common_audio = 1853

constant_tone_extension = 184a

continuous_glucose_monitoring = 181f

coordinated_set_identification = 1846

current_time = 1805

cycling_power = 1818

cycling_speed_and_cadence = 1816

device_information = 180a

device_time = 1847

emergency_configuration = 183c

environmental_sensing = 181a

fitness_machine = 1826

generic_access = 1800

generic_attribute = 1801

```
generic_media_control = 1849
generic_telephone_bearer = 184c
glucose = 1808
health_thermometer = 1809
hearing_access = 1854
heart_rate = 180d
http_proxy = 1823
human_interface_device = 1812
immediate_alert = 1802
indoor_positioning = 1821
insulin_delivery = 183a
internet_protocol_support = 1820
link_loss = 1803
location_and_navigation = 1819
media_control = 1848
mesh_provisioning = 1827
mesh_proxy = 1828
microphone_control = 184d
next_dst_change = 1807
object_transfer = 1825
phone_alert_status = 180e
physical_activity_monitor = 183e
published_audio_capabilities = 1850
pulse_oximeter = 1822
reconnection_configuration = 1829
reference_time_update = 1806
running_speed_and_cadence = 1814
scan_parameters = 1813
telephone_bearer = 184b
tmas = 1855
transport_discovery = 1824
```

tx_power = 1804

user_data = 181c

volume_control = 1844

volume_offset_control = 1845

weight_scale = 181d

class blatann.bt_sig.uuids.CharacteristicUuid

Bases: object

Bluetooth SIG defined characteristic UUIDs

active_preset_index = 2bdc

activity_current_session = 2b44

activity_goal = 2b4e

adv_constant_tone_interval = 2bb1

adv_constant_tone_min_length = 2bae

adv_constant_tone_min_tx_count = 2baf

adv_constant_tone_phy = 2bb2

adv_constant_tone_tx_duration = 2bb0

aerobic_heart_rate_lower_limit = 2a7e

aerobic_heart_rate_upper_limit = 2a84

aerobic_threshold = 2a7f

age = 2a80

aggregate = 2a5a

alert_category_id = 2a43

alert_category_id_bit_mask = 2a42

alert_level = 2a06

alert_notification_control_point = 2a44

alert_status = 2a3f

altitude = 2ab3

ammonia_concentration = 2bcf

anaerobic_heart_rate_lower_limit = 2a81

anaerobic_heart_rate_upper_limit = 2a82

anaerobic_threshold = 2a83

```
analog = 2a58
analog_output = 2a59
apparent_wind_direction = 2a73
apparent_wind_speed = 2a72
appearance = 2a01
ase_control_point = 2bc6
audio_input_control_point = 2b7b
audio_input_description = 2b7c
audio_input_state = 2b77
audio_input_status = 2b7a
audio_input_type = 2b79
audio_location = 2b81
audio_output_description = 2b83
available_audio_contexts = 2bcd
average_current = 2ae0
average_voltage = 2ae1
barometric_pressure_trend = 2aa3
battery_level = 2a19
battery_level_state = 2a1b
battery_power_state = 2a1a
bearer_list_current_calls = 2bb9
bearer_provider_name = 2bb3
bearer_signal_strength = 2bb7
bearer_signal_strength_reporting_interval = 2bb8
bearer_technology = 2bb5
bearer_uci = 2bb4
bearer_uri_schemes_supported_list = 2bb6
blood_pressure_feature = 2a49
blood_pressure_measurement = 2a35
blood_pressure_record = 2b36
bluetooth_sig_data = 2b39
```

```
body_composition_feature = 2a9b
body_composition_measurement = 2a9c
body_sensor_location = 2a38
bond_management_control_point = 2aa4
bond_management_feature = 2aa5
boolean = 2ae2
boot_keyboard_input_report = 2a22
boot_keyboard_output_report = 2a32
boot_mouse_input_report = 2a33
br_edr_handover_data = 2b38
broadcast_audio_scan_control_point = 2bc7
broadcast_receive_state = 2bc8
bss_control_point = 2b2b
bss_response = 2b2c
call_control_point = 2bbe
call_control_point_optional_opcodes = 2bbf
call_friendly_name = 2bc2
call_state = 2bbd
caloric_intake = 2b50
carbon_monoxide_concentration = 2bd0
cardiorespiratory_activity_instantaneous_data = 2b3e
cardiorespiratory_activity_summary_data = 2b3f
central_address_resolution = 2aa6
cgm_feature = 2aa8
cgm_measurement = 2aa7
cgm_session_run_time = 2aab
cgm_session_start_time = 2aaa
cgm_specific_ops_control_point = 2aac
cgm_status = 2aa9
chromatic_distance_from_planckian = 2ae3
chromaticity_coordinate = 2b1c
```

```
chromaticity_coordinates = 2ae4
chromaticity_in_cct_and_duv_values = 2ae5
chromaticity_tolerance = 2ae6
cie_color_rendering_index = 2ae7
client_supported_features = 2b29
coefficient = 2ae8
constant_tone_extension_enable = 2bad
content_control_id = 2bba
coordinated_set_size = 2b85
correlated_color_temperature = 2ae9
count_16 = 2aea
count_24 = 2aeb
country_code = 2aec
cross_trainer_data = 2ace
csc_feature = 2a5c
csc_measurement = 2a5b
current_group_object_id = 2ba0
current_time = 2a2b
current_track_object_id = 2b9d
current_track_segments_object_id = 2b9c
cycling_power_control_point = 2a66
cycling_power_feature = 2a65
cycling_power_measurement = 2a63
cycling_power_vector = 2a64
database_change_increment = 2a99
database_hash = 2b2a
date_of_birth = 2a85
date_of_threshold_assessment = 2a86
date_time = 2a08
date_utc = 2aed
day_date_time = 2a0a
```

```
day_of_week = 2a09
descriptor_value_changed = 2a7d
device_name = 2a00
device_time = 2b90
device_time_control_point = 2b91
device_time_feature = 2b8e
device_time_parameters = 2b8f
device_wearing_position = 2b4b
dew_point = 2a7b
digital = 2a56
digital_output = 2a57
directory_listing = 2acb
dst_offset = 2a0d
electric_current = 2aee
electric_current_range = 2aef
electric_current_specification = 2af0
electric_current_statistics = 2af1
elevation = 2a6c
email_address = 2a87
emergency_id = 2b2d
emergency_text = 2b2e
energy = 2af2
energy_in_a_period_of_day = 2af3
enhanced_blood_pressure_measurement = 2b34
enhanced_intermediate_cuff_pressure = 2b35
event_statistics = 2af4
exact_time_100 = 2a0b
exact_time_256 = 2a0c
fat_burn_heart_rate_lower_limit = 2a88
fat_burn_heart_rate_upper_limit = 2a89
firmware_revision_string = 2a26
```

```
first_name = 2a8a
fitness_machine_control_point = 2ad9
fitness_machine_feature = 2acc
fitness_machine_status = 2ada
five_zone_heart_rate_limits = 2a8b
fixed_string_16 = 2af5
fixed_string_24 = 2af6
fixed_string_36 = 2af7
fixed_string_8 = 2af8
floor_number = 2ab2
four_zone_heart_rate_limits = 2b4c
gain_settings_attribute = 2b78
gender = 2a8c
general_activity_instantaneous_data = 2b3c
general_activity_summary_data = 2b3d
generic_level = 2af9
global_trade_item_number = 2afa
glucose_feature = 2a51
glucose_measurement = 2a18
glucose_measurement_context = 2a34
group_object_type = 2bac
gust_factor = 2a74
handedness = 2b4a
hardware_revision_string = 2a27
hearing_aid_features = 2bda
hearing_aid_preset_control_point = 2bdb
heart_rate_control_point = 2a39
heart_rate_max = 2a8d
heart_rate_measurement = 2a37
heat_index = 2a7a
height = 2a8e
```

```
hid_control_point = 2a4c
hid_information = 2a4a
high_intensity_exercise_threshold = 2b4d
high_resolution_height = 2b47
hip_circumference = 2a8f
http_control_point = 2aba
http_entity_body = 2ab9
http_headers = 2ab7
http_status_code = 2ab8
https_security = 2abb
humidity = 2a6f
idd_annunciation_status = 2b22
idd_command_control_point = 2b25
idd_command_data = 2b26
idd_features = 2b23
idd_history_data = 2b28
idd_record_access_control_point = 2b27
idd_status = 2b21
idd_status_changed = 2b20
idd_status_reader_control_point = 2b24
ieee11073_20601_regulatory_certification_data_list = 2a2a
illuminance = 2afb
incoming_call = 2bc1
incoming_call_target_bearer_uri = 2bbc
indoor_bike_data = 2ad2
indoor_positioning_configuration = 2aad
intermediate_cuff_pressure = 2a36
intermediate_temperature = 2a1e
irradiance = 2a77
language = 2aa2
last_name = 2a90
```

```
latitude = 2aae
ln_control_point = 2a6b
ln_feature = 2a6a
local_east_coordinate = 2ab1
local_north_coordinate = 2ab0
local_time_information = 2a0f
location_and_speed = 2a67
location_name = 2ab5
longitude = 2aaf
luminous_efficacy = 2afc
luminous_energy = 2afd
luminous_exposure = 2afe
luminous_flux = 2aff
luminous_flux_range = 2b00
luminous_intensity = 2b01
magnetic_declination = 2a2c
magnetic_flux_density_2d = 2aa0
magnetic_flux_density_3d = 2aa1
manufacturer_name_string = 2a29
mass_flow = 2b02
maximum_recommended_heart_rate = 2a91
measurement_interval = 2a21
media_control_point = 2ba4
media_control_point_opcodes_supported = 2ba5
media_player_icon_object_id = 2b94
media_player_icon_object_type = 2ba9
media_player_icon_url = 2b95
media_player_name = 2b93
media_state = 2ba3
mesh_provisioning_data_in = 2adb
mesh_provisioning_data_out = 2adc
```

```
mesh_proxy_data_in = 2add

mesh_proxy_data_out = 2ade

methane_concentration = 2bd1

middle_name = 2b48

model_number_string = 2a24

mute = 2bc3

navigation = 2a68

network_availability = 2a3e

new_alert = 2a46

next_track_object_id = 2b9e

nitrogen_dioxide_concentration = 2bd2

non_methane_volatile_organic_compounds_concentration = 2bd3

object_action_control_point = 2ac5

object_changed = 2ac8

object_first_created = 2ac1

object_id = 2ac3

object_last_modified = 2ac2

object_list_control_point = 2ac6

object_list_filter = 2ac7

object_name = 2abe

object_properties = 2ac4

object_size = 2ac0

object_type = 2abf

ots_feature = 2abd

ozone_concentration = 2bd4

parent_group_object_id = 2b9f

particulate_matter_10_concentration = 2bd7

particulate_matter_1_concentration = 2bd5

particulate_matter_2_5_concentration = 2bd6

perceived_lightness = 2b03

percentage_8 = 2b04
```

```
peripheral_preferred_connection_parameters = 2a04
peripheral_privacy_flag = 2a02
physical_activity_monitor_control_point = 2b43
physical_activity_monitor_features = 2b3b
physical_activity_session_descriptor = 2b45
playback_speed = 2b9a
playing_order = 2ba1
playing_orders_supported = 2ba2
plx_continuous_measurement = 2a5f
plx_features = 2a60
plx_spot_check_measurement = 2a5e
pnp_id = 2a50
pollen_concentration = 2a75
position_2d = 2a2f
position_3d = 2a30
position_quality = 2a69
power = 2b05
power_specification = 2b06
preferred_units = 2b46
pressure = 2a6d
protocol_mode = 2a4e
pulse_oximetry_control_point = 2a62
rainfall = 2a78
rc_feature = 2b1d
rc_settings = 2b1e
reconnection_address = 2a03
reconnection_configuration_control_point = 2b1f
record_access_control_point = 2a52
reference_time_information = 2a14
registered_user_characteristic = 2b37
relative_runtime_current_range = 2b07
```

relative_runtime_generic_level_range = 2b08

relative_value_period_of_day = 2b0b

relative_value_temperature_range = 2b0c

relative_value_voltage_range = 2b09

relative_value_illuminance_range = 2b0a

removable = 2a3a

report = 2a4d

report_map = 2a4b

resolvable_private_address_only = 2ac9

resting_heart_rate = 2a92

ringer_control_point = 2a40

ringer_setting = 2a41

rower_data = 2ad1

rsc_feature = 2a54

rsc_measurement = 2a53

sc_control_point = 2a55

scan_interval_window = 2a4f

scan_refresh = 2a31

scientific_temperature_celsius = 2a3c

secondary_time_zone = 2a10

search_control_point = 2ba7

search_results_object_id = 2ba6

sedentary_interval_notification = 2b4f

seeking_speed = 2b9b

sensor_location = 2a5d

serial_number_string = 2a25

server_supported_features = 2b3a

service_changed = 2a05

service_required = 2a3b

set_identity_resolving_key = 2b84

set_member_lock = 2b86

```
set_member_rank = 2b87
sink_ase = 2bc4
sink_audio_locations = 2bca
sink_pac = 2bc9
sleep_activity_instantaneous_data = 2b41
sleep_activity_summary_data = 2b42
software_revision_string = 2a28
source_ase = 2bc5
source_audio_locations = 2bcc
source_pac = 2bcb
sport_type_for_aerobic_and_anaerobic_thresholds = 2a93
stair_climber_data = 2ad0
status_flags = 2bbb
step_climber_data = 2acf
step_counter_activity_summary_data = 2b40
stride_length = 2b49
string = 2a3d
sulfur_dioxide_concentration = 2bd8
sulfur_hexafluoride_concentration = 2bd9
supported_audio_contexts = 2bce
supported_heart_rate_range = 2ad7
supported_inclination_range = 2ad5
supported_new_alert_category = 2a47
supported_power_range = 2ad8
supported_resistance_level_range = 2ad6
supported_speed_range = 2ad4
supported_unread_alert_category = 2a48
system_id = 2a23
tds_control_point = 2abc
temperature = 2a6e
temperature_celsius = 2a1f
```

temperature_fahrenheit = 2a20

temperature_8 = 2b0d

temperature_8_in_a_period_of_day = 2b0e

temperature_8_statistics = 2b0f

temperature_measurement = 2a1c

temperature_range = 2b10

temperature_statistics = 2b11

temperature_type = 2a1d

termination_reason = 2bc0

three_zone_heart_rate_limits = 2a94

time_accuracy = 2a12

time_broadcast = 2a15

time_change_log_data = 2b92

time_decihour_8 = 2b12

time_exponential_8 = 2b13

time_hour_24 = 2b14

time_millisecond_24 = 2b15

time_second_16 = 2b16

time_second_8 = 2b17

time_source = 2a13

time_update_control_point = 2a16

time_update_state = 2a17

time_with_dst = 2a11

time_zone = 2a0e

tmap_role = 2b51

track_changed = 2b96

track_duration = 2b98

track_object_type = 2bab

track_position = 2b99

track_segments_object_type = 2baa

track_title = 2b97

```
training_status = 2ad3

treadmill_data = 2acd

true_wind_direction = 2a71

true_wind_speed = 2a70

two_zone_heart_rate_limit = 2a95

tx_power_level = 2a07

uncertainty = 2ab4

unread_alert_status = 2a45

unspecified = 2aca

uri = 2ab6

user_control_point = 2a9f

user_index = 2a9a

uv_index = 2a76

vo2_max = 2a96

voltage = 2b18

voltage_specification = 2b19

voltage_statistics = 2b1a

volume_control_point = 2b7e

volume_flags = 2b7f

volume_flow = 2b1b

volume_offset_control_point = 2b82

volume_offset_state = 2b80

volume_state = 2b7d

waist_circumference = 2a97

weight = 2a98

weight_measurement = 2a9d

weight_scale_feature = 2a9e

wind_chill = 2a79
```

blatann.bt_sig.uuids.company_assigned_uuid16s = {fce1: 'Sony Group Corporation', fce2: 'Baracoda Daily Healthtech', fce3: 'Smith & Nephew Medical Limited', fce4: 'Samsara Networks, Inc', fce5: 'Samsara Networks, Inc', fce6: 'Guard RFID Solutions Inc.', fce7: 'TKH Security B.V.', fce8: 'ITT Industries', fce9: 'MindRhythm, Inc.', fcea: 'Chess Wise B.V.', fceb: 'Avi-On', fcec: 'Griffwerk GmbH', fced: 'Workaround Gmbh', fcee: 'Velentium, LLC', fcef: 'Divesoft s.r.o.', fcf0: 'Security Enhancement Systems, LLC', fcf1: 'Google LLC', fcf2: 'Bitwards Oy', fcf3: 'Armatura LLC', fcf4: 'Allegion', fcf5: 'Trident Communication Technology, LLC', fcf6: 'The Linux Foundation', fcf7: 'Honor Device Co., Ltd.', fcf8: 'Honor Device Co., Ltd.', fcf9: 'Leupold & Stevens, Inc.', fcfa: 'Leupold & Stevens, Inc.', fcfb: 'Shenzhen Benwei Media Co., Ltd.', fcfc: 'Barrot Technology Limited', fcfd: 'Barrot Technology Limited', fcfe: 'Sennheiser Consumer Audio GmbH', fcff: '701x', fd00: 'FUTEK Advanced Sensor Technology, Inc.', fd01: 'Sanvita Medical Corporation', fd02: 'LEGO System A/S', fd03: 'Quuppa Oy', fd04: 'Shure Inc.', fd05: 'Qualcomm Technologies, Inc.', fd06: 'RACE-AI LLC', fd07: 'Swedlock AB', fd08: 'Bull Group Incorporated Company', fd09: 'Cousins and Sears LLC', fd0a: 'Luminostics, Inc.', fd0b: 'Luminostics, Inc.', fd0c: 'OSM HK Limited', fd0d: 'Blecon Ltd', fd0e: 'HerdDogg, Inc', fd0f: 'AEON MOTOR CO.,LTD.', fd10: 'AEON MOTOR CO.,LTD.', fd11: 'AEON MOTOR CO.,LTD.', fd12: 'AEON MOTOR CO.,LTD.', fd13: 'BRG Sports, Inc.', fd14: 'BRG Sports, Inc.', fd15: 'Panasonic Corporation', fd16: 'Sensitech, Inc.', fd17: 'LEGIC Identsystems AG', fd18: 'LEGIC Identsystems AG', fd19: 'Smith & Nephew Medical Limited', fd1a: 'CSIRO', fd1b: 'Helios Sports, Inc.', fd1c: 'Brady Worldwide Inc.', fd1d: 'Samsung Electronics Co., Ltd', fd1e: 'Plume Design Inc.', fd1f: '3M', fd20: 'GN Hearing A/S', fd21: 'Huawei Technologies Co., Ltd.', fd22: 'Huawei Technologies Co., Ltd.', fd23: 'DOM Sicherheitstechnik GmbH & Co. KG', fd24: 'GD Midea Air-Conditioning Equipment Co., Ltd.', fd25: 'GD Midea Air-Conditioning Equipment Co., Ltd.', fd26: 'Novo Nordisk A/S', fd27: 'i2Systems', fd28: 'Julius Blum GmbH', fd29: 'Asahi Kasei Corporation', fd2a: 'Sony Corporation', fd2b: 'The Access Technologies', fd2c: 'The Access Technologies', fd2d: 'Xiaomi Inc.', fd2e: 'Bitstrata Systems Inc.', fd2f: 'Bitstrata Systems Inc.', fd30: 'Sesam Solutions BV', fd31: 'LG Electronics Inc.', fd32: 'Gemalto Holding BV', fd33: 'DashLogic, Inc.', fd34: 'Aerosens LLC.', fd35: 'Transsion Holdings Limited', fd36: 'Google LLC', fd37: 'TireCheck GmbH', fd38: 'Danfoss A/S', fd39: 'PREDIKTAS', fd3a: 'Verkada Inc.', fd3b: 'Verkada Inc.', fd3c: 'Redline Communications Inc.', fd3d: 'Woan Technology (Shenzhen) Co., Ltd.', fd3e: 'Pure Watercraft, inc.', fd3f: 'Cognosos, Inc', fd40: 'Beflex Inc.', fd41: 'Amazon Lab126', fd42: 'Globe (Jiangsu) Co.,Ltd', fd43: 'Apple Inc.', fd44: 'Apple Inc.', fd45: 'GB Solution co.,Ltd', fd46: 'Lemco IKE', fd47: 'Liberty Global Inc.', fd48: 'Geberit International AG', fd49: 'Panasonic Corporation', fd4a: 'Sigma Elektro GmbH', fd4b: 'Samsung Electronics Co., Ltd.', fd4c: 'Adolf Wuerth GmbH & Co KG', fd4d: '70mai Co.,Ltd.', fd4e: '70mai Co.,Ltd.', fd4f: 'Forkbeard Technologies AS', fd50: 'Hangzhou Tuya Information Technology Co., Ltd', fd51: 'UTC Fire and Security', fd52: 'UTC Fire and Security', fd53: 'PCI Private Limited', fd54: 'Qingdao Haier Technology Co., Ltd.', fd55: 'Braveheart Wireless, Inc.', fd57: 'Volvo Car Corporation', fd58: 'Volvo Car Corporation', fd59: 'Samsung Electronics Co., Ltd.', fd5a: 'Samsung Electronics Co., Ltd.', fd5b: 'V2SOFT INC.', fd5c: 'React Mobile', fd5d: 'maxon motor ltd.', fd5e: 'Tapkey GmbH', fd5f: 'Oculus VR, LLC', fd60: 'Sercomm Corporation', fd61: 'Arendi AG', fd62: 'Fitbit, Inc.', fd63: 'Fitbit, Inc.', fd64: 'INRIA', fd65: 'Razer Inc.', fd66: 'Zebra Technologies Corporation', fd67: 'Montblanc Simplo GmbH', fd68: 'Ubique Innovation AG', fd69: 'Samsung Electronics Co., Ltd.', fd6a: 'Emerson', fd6b: 'rapitag GmbH', fd6c: 'Samsung Electronics Co., Ltd.', fd6d: 'Sigma Elektro GmbH', fd6e: 'Polidea sp. z o.o.', fd6f: 'Apple, Inc.', fd70: 'GuangDong Oppo Mobile Telecommunications Corp., Ltd.', fd71: 'GN Hearing A/S', fd72: 'Logitech International SA', fd73: 'BRControls Products BV', fd74: 'BRControls Products BV', fd75: 'Insulet Corporation', fd76: 'Insulet Corporation', fd77: 'Withings', fd78: 'Withings', fd79: 'Withings', fd7a: 'Withings', fd7b: 'WYZE LABS, INC.', fd7c: 'Toshiba Information Systems(Japan) Corporation', fd7d: 'Center for Advanced Research Wernher Von Braun', fd7e: 'Samsung Electronics Co., Ltd.', fd7f: 'Husqvarna AB', fd80: 'Phindex Technologies, Inc', fd81: 'CANDY HOUSE, Inc.', fd82: 'Sony Corporation', fd83: 'iNFORM Technology GmbH', fd84: 'Tile, Inc.', fd85: 'Husqvarna AB', fd86: 'Abbott', fd87: 'Google LLC', fd88: 'Urbanminded LTD', fd89: 'Urbanminded LTD', fd8a: 'Signify Netherlands B.V.', fd8b: 'Jigowatts Inc.', fd8c: 'Google LLC', fd8d: 'quip NYC Inc.',

16-bit UUIDs assigned to companies by Bluetooth SIG

blatann.bt_sig.uuids.**t**

alias of *CharacteristicUuid*

blatann.examples package

Submodules

blatann.examples.broadcaster module

This is an example of a broadcaster role BLE device. It advertises as a non-connectable device and emits the device's current time as a part of the advertising data.

blatann.examples.broadcaster.**wait_for_user_stop**(*stop_event*)

blatann.examples.broadcaster.**main**(*serial_port*)

blatann.examples.centeral_uart_service module

This example implements Nordic's custom UART service and demonstrates how to configure the MTU size. It is configured to use an MTU size based on the Data Length Extensions feature of BLE for maximum throughput. This is compatible with the peripheral_uart_service example.

This is a simple example which just echos back any data that the client sends to it.

blatann.examples.centeral_uart_service.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.centeral_uart_service.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.centeral_uart_service.**on_mtu_size_update**(*peer*, *event_args*)

Callback for when the peer's MTU size has been updated/negotiated

Parameters

peer (blatann.peer.Client) – The peer the MTU was updated on

blatann.examples.centeral_uart_service.**on_data_rx**(*service*, *data*)

Called whenever data is received on the RX line of the Nordic UART Service

Parameters

- **service** (nordic_uart.service.NordicUartClient) – the service the data was received from

- **data** (*bytes*) – The data that was received

blatann.examples.centeral_uart_service.**main**(*serial_port*)

blatann.examples.central module

This example demonstrates implementing a central BLE connection in a procedural manner. Each bluetooth operation performed is done sequentially in a linear fashion, and the main context blocks until each operation completes before moving on to the rest of the program

This is designed to work alongside the peripheral example running on a separate nordic chip

blatann.examples.central.**on_counting_char_notification**(*characteristic*, *event_args*)

Callback for when a notification is received from the peripheral's counting characteristic. The peripheral will periodically notify a monotonically increasing, 4-byte integer. This callback unpacks the value and logs it out

Parameters

- **characteristic** (blatann.gatt.gattc.GattcCharacteristic) – The characteristic the notification was on (counting characteristic)
- **event_args** (blatann.event_args.NotificationReceivedEventArgs) – The event arguments

blatann.examples.central.**on_passkey_entry**(*peer*, *passkey_event_args*)

Callback for when the user is requested to enter a passkey to resume the pairing process. Requests the user to enter the passkey and resolves the event with the passkey entered

Parameters

- **peer** – the peer the passkey is for
- **passkey_event_args** (blatann.event_args.PasskeyEntryEventArgs) –

blatann.examples.central.**on_peripheral_security_request**(*peer*, *event_args*)

Handler for peripheral-initiated security requests. This is useful in the case that the application wants to override the default response to peripheral-initiated security requests based on parameters, the peer, etc.

For example, to reject new pairing requests but allow already-bonded devices to enable encryption, one could use the event_args.is_bonded_device flag to accept or reject the request.

This handler is optional. If not provided the SecurityParameters.reject_pairing_requests parameter will determine the action to take.

Parameters

- **peer** (blatann.peer.Peer) – The peer that requested security
- **event_args** (blatann.event_args.PeripheralSecurityRequestEventArgs) – The event arguments

blatann.examples.central.**main**(*serial_port*)

blatann.examples.central_battery_service module

This example demonstrates reading and subscribing to a peripheral's Battery service to get updates on the peripheral's current battery levels. The operations here are programmed in a procedural manner.

This can be used alongside any peripheral which implements the Battery Service and advertises the 16-bit Battery Service UUID. The peripheral_battery_service example can be used with this.

blatann.examples.central_battery_service.**on_battery_level_update**(*battery_service*, *event_args*)

Parameters
battery_service –

blatann.examples.central_battery_service.**main**(*serial_port*)

blatann.examples.central_descriptors module

This example shows how to read descriptors of a peripheral's characteristic.

This can be used with the peripheral_descriptor example running on a separate nordic device.

blatann.examples.central_descriptors.**main**(*serial_port*)

blatann.examples.central_device_info_service module

This example demonstrates reading a peripheral's Device Info Service using blatann's device_info service module. The operations here are programmed in a procedural manner.

This can be used alongside any peripheral which implements the DIS and advertises the 16-bit DIS service UUID. The peripheral_device_info_service example can be used with this.

blatann.examples.central_device_info_service.**main**(*serial_port*)

blatann.examples.central_event_driven module

This example demonstrates programming with blatann in an event-driven, object-oriented manner. The BLE device is set up in the main context, however all logic past that point is done using event callbacks. The main context is blocked by a "GenericWaitable", which is notified when the program completes its intended function.

The program's logic itself is equivalent to the central example, where it connects and pairs to a device, registers a notification callback for the counting characteristic, then tests out the conversion of strings to hex.

One thing to note: when using event-driven callbacks, it is imperative that the callbacks themselves do not ever block on events (i.e. use the .wait() functionality). If this happens, you are essentially blocking the event thread from processing any more events and will wait indefinitely. A good rule of thumb when using blatann is just to not mix blocking and non-blocking calls.

This is designed to work alongside the peripheral example running on a separate nordic chip

class blatann.examples.central_event_driven.**HexConverterTest**(*characteristic*, *waitable*)

Bases: object

Class to perform the hex conversion process. It is passed in the hex conversion characteristic and the waitable to signal when the process completes

start()

Starts a new hex conversion process by writing the data to the peripheral's characteristic

class blatann.examples.central_event_driven.**MyPeripheralConnection**(*peer*, *waitable*)

Bases: object

Class to handle the post-connection database discovery and pairing process

class blatann.examples.central_event_driven.**ConnectionManager**(*ble_device*, *exit_waitable*)

Bases: object

Manages scanning and connecting to the target peripheral

scan_and_connect(*name*, *timeout=4*)

Starts the scanning process and sets up the callback for when scanning completes

Parameters

- **name** – The name of the peripheral to look for
- **timeout** – How long to scan for

blatann.examples.central_event_driven.**main**(*serial_port*)

blatann.examples.constants module

blatann.examples.example_utils module

blatann.examples.example_utils.**find_target_device**(*ble_device*, *name*)

Starts the scanner and searches the advertising report for the desired name. If found, returns the peer's address that can be connected to

Parameters

- **ble_device** (*blatann.BleDevice*) – The ble device to operate on
- **name** – The device's local name that is advertised

Returns

The peer's address if found, or None if not found

blatann.examples.peripheral module

This example exhibits some of the functionality of a peripheral BLE device, such as reading, writing and notifying characteristics.

This peripheral can be used with one of the central examples running on a separate nordic device, or can be run with the nRF Connect app to explore the contents of the service

blatann.examples.peripheral.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral.**on_hex_conversion_characteristic_write**(*characteristic*, *event_args*)

Event callback for when the client writes to the hex conversion characteristic. This takes the data written, converts it to the hex representation, and updates the characteristic with this new value. If the client is subscribed to the characteristic, the client will be notified.

Parameters

- **characteristic** (blatann.gatt.gatts.GattsCharacteristic) – The hex conversion characteristic
- **event_args** (blatann.event_args.WriteEventArgs) – the event arguments

blatann.examples.peripheral.**on_gatts_subscription_state_changed**(*characteristic*, *event_args*)

Event callback for when a client subscribes or unsubscribes from a characteristic. This is the equivalent to when a client writes to a CCCD descriptor on a characteristic.

blatann.examples.peripheral.**on_time_char_read**(*characteristic*, *event_args*)

Event callback for when the client reads our time characteristic. Gets the current time and updates the characteristic. This demonstrates "lazy evaluation" of characteristics–instead of having to constantly update this characteristic, it is only updated when read/observed by an outside actor.

Parameters

- **characteristic** (blatann.gatt.gatts.GattsCharacteristic) – the time characteristic
- **event_args** – None

blatann.examples.peripheral.**on_discovery_complete**(*peer*, *event_args*)

Callback for when the service discovery completes on the client. This will look for the client's Device name characteristic (part of the Generic Access Service) and read the value

Parameters

- **peer** (blatann.peer.Client) – The peer the discovery completed on
- **event_args** (blatann.event_args.DatabaseDiscoveryCompleteEventArgs) – The event arguments (unused)

blatann.examples.peripheral.**on_security_level_changed**(*peer*, *event_args*)

Called when the security level changes, i.e. a bonded device connects and enables encryption or pairing has finished. If security has been enabled (i.e. bonded) and the peer's services have yet to be discovered, discover now.

This code demonstrates that even in a peripheral connection role, the peripheral can still discover the database on the client, if the client has a database.

Parameters

- **peer** (blatann.peer.Client) – The peer that security was changed to
- **event_args** (blatann.event_args.SecurityLevelChangedEventArgs) – the event arguments

blatann.examples.peripheral.**on_client_pairing_complete**(*peer*, *event_args*)

Event callback for when the pairing process completes with the client

Parameters

- **peer** (blatann.peer.Client) – the peer that completed pairing
- **event_args** (blatann.event_args.PairingCompleteEventArgs) – the event arguments

blatann.examples.peripheral.**on_passkey_display**(*peer*, *event_args*)

Event callback that is called when a passkey is required to be displayed to a user for the pairing process.

Parameters

- **peer** (blatann.peer.Client) – The peer the passkey is for
- **event_args** (blatann.event_args.PasskeyDisplayEventArgs) – The event args

blatann.examples.peripheral.**on_passkey_entry**(*peer*, *passkey_event_args*)

Callback for when the user is requested to enter a passkey to resume the pairing process. Requests the user to enter the passkey and resolves the event with the passkey entered

Parameters

- **peer** – the peer the passkey is for
- **passkey_event_args** (blatann.event_args.PasskeyEntryEventArgs) –

class blatann.examples.peripheral.**CountingCharacteristicThread**(*characteristic*)

Bases: object

Thread which updates the counting characteristic and notifies the client each time its updated. This also demonstrates the notification queuing functionality–if a notification/indication is already in progress, future notifications will be queued and sent out when the previous ones complete.

join()

Used to stop and join the thread

run()

blatann.examples.peripheral.**on_conn_params_updated**(*peer*, *event_args*)

blatann.examples.peripheral.**main**(*serial_port*)

blatann.examples.peripheral_battery_service module

This example demonstrates using Bluetooth SIG's Battery service as a peripheral. The peripheral adds the service, then updates the battery level percentage periodically.

This can be used in conjunction with the nRF Connect apps to explore the functionality demonstrated

blatann.examples.peripheral_battery_service.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_battery_service.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_battery_service.**main**(*serial_port*)

blatann.examples.peripheral_current_time_service module

This example demonstrates using Bluetooth SIG's defined Current Time service as a peripheral.

blatann.examples.peripheral_current_time_service.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_current_time_service.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_current_time_service.**on_current_time_write**(*characteristic*, *event_args*)

Callback registered to be triggered whenever the Current Time characteristic is written to

Parameters

- **characteristic** –
- **event_args** (blatann.event_args.DecodedWriteEventArgs) – The write event args

blatann.examples.peripheral_current_time_service.**on_local_time_info_write**(*characteristic*, *event_args*)

Callback registered to be triggered whenever the Local Time Info characteristic is written to

Parameters

- **characteristic** –
- **event_args** (blatann.event_args.DecodedWriteEventArgs) – The write event args

blatann.examples.peripheral_current_time_service.**main**(*serial_port*)

blatann.examples.peripheral_descriptors module

This example shows how to add descriptors to a characteristic in a GATT Database.

This can be used with the central_descriptor example running on a separate nordic device or can be run with the nRF Connect app

blatann.examples.peripheral_descriptors.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_descriptors.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_descriptors.**on_read**(*characteristic*, *event_args*)

On read for the Time characteristic. Updates the characteristic with the current UTC time as a 32-bit number

blatann.examples.peripheral_descriptors.**main**(*serial_port*)

blatann.examples.peripheral_device_info_service module

This example shows how to implement a Device Info Service on a peripheral.

This example can be used alongside the central_device_info_service example running on another nordic device, or using the Nordic nRF Connect app to connect and browse the peripheral's service data

blatann.examples.peripheral_device_info_service.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_device_info_service.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_device_info_service.**main**(*serial_port*)

blatann.examples.peripheral_glucose_service module

This example demonstrates using Bluetooth SIG's defined Glucose service as a peripheral. The peripheral creates a range of fake glucose readings that can be queried from the central.

This can be used in conjunction with the nRF Connect apps to explore the peripheral's functionality

blatann.examples.peripheral_glucose_service.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_glucose_service.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_glucose_service.**on_security_level_changed**(*peer*, *event_args*)

Event callback for when the security level changes on a connection with a peer

Parameters

- **peer** (blatann.peer.Client) – The peer the security level changed on
- **event_args** (blatann.event_args.SecurityLevelChangedEventArgs) – The event args

blatann.examples.peripheral_glucose_service.**display_passkey**(*peer*, *event_args*)

Event callback that is called when a passkey is required to be displayed to a user for the pairing process.

Parameters

- **peer** (blatann.peer.Client) – The peer the passkey is for
- **event_args** (blatann.event_args.PasskeyDisplayEventArgs) – The event args

blatann.examples.peripheral_glucose_service.**add_fake_glucose_readings**(*glucose_database*, *num_records=15*)

Helper method to create some glucose readings and add them to the glucose database

Parameters

- **glucose_database** (*glucose.BasicGlucoseDatabase*) – The database to add readings to
- **num_records** – The number of records to generate

blatann.examples.peripheral_glucose_service.**main**(*serial_port*)

blatann.examples.peripheral_rssi module

This is a simple example which demonstrates enabling RSSI updates for active connections.

blatann.examples.peripheral_rssi.**on_rssi_changed**(*peer*, *rssi*)

Event callback for when the RSSI with the central device changes by the configured dBm threshold

Parameters

- **peer** – The peer object
- **rssi** (int) – The new RSSI for the connection

blatann.examples.peripheral_rssi.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_rssi.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_rssi.**main**(*serial_port*)

blatann.examples.peripheral_uart_service module

This example implements Nordic's custom UART service and demonstrates how to configure the MTU size. It is configured to use an MTU size based on the Data Length Extensions feature of BLE for maximum throughput. This is compatible with the nRF Connect app (Android version tested) and the central_uart_service example.

This is a simple example which just echos back any data that the client sends to it.

blatann.examples.peripheral_uart_service.**on_connect**(*peer*, *event_args*)

Event callback for when a central device connects to us

Parameters

- **peer** (blatann.peer.Client) – The peer that connected to us
- **event_args** – None

blatann.examples.peripheral_uart_service.**on_disconnect**(*peer*, *event_args*)

Event callback for when the client disconnects from us (or when we disconnect from the client)

Parameters

- **peer** (blatann.peer.Client) – The peer that disconnected
- **event_args** (blatann.event_args.DisconnectionEventArgs) – The event args

blatann.examples.peripheral_uart_service.**on_mtu_size_update**(*peer*, *event_args*)

Callback for when the peer's MTU size has been updated/negotiated

Parameters

peer (blatann.peer.Client) – The peer the MTU was updated on

blatann.examples.peripheral_uart_service.**on_data_rx**(*service*, *data*)

Called whenever data is received on the RX line of the Nordic UART Service

Parameters

- **service** (nordic_uart.service.NordicUartServer) – the service the data was received from
- **data** (*bytes*) – The data that was received

blatann.examples.peripheral_uart_service.**on_tx_complete**(*service*, *event_args*)

blatann.examples.peripheral_uart_service.**main**(*serial_port*)

blatann.examples.scanner module

This example simply demonstrates scanning for peripheral devices

blatann.examples.scanner.**main**(*serial_port*)

blatann.gap package

blatann.gap.**HciStatus**

The default link-layer packet size used when a connection is established

blatann.gap.**DLE_SIZE_DEFAULT = 27**

The minimum allowed link-layer packet size

blatann.gap.**DLE_SIZE_MINIMUM = 27**

The maximum allowed link-layer packet size

Submodules

blatann.gap.advertise_data module

class blatann.gap.advertise_data.**AdvertisingFlags**

Bases: object

LIMITED_DISCOVERY_MODE = 1

GENERAL_DISCOVERY_MODE = 2

BR_EDR_NOT_SUPPORTED = 4

BR_EDR_CONTROLLER = 8

BR_EDR_HOST = 16

class blatann.gap.advertise_data.**AdvertisingData**(*flags=None*, *local_name=None*, *local_name_complete=True*, *service_uuid16s=None*, *service_uuid128s=None*, *has_more_uuid16_services=False*, *has_more_uuid128_services=False*, *service_data=None*, *manufacturer_data=None*, ***other_entries*)

Bases: object

Class which represents data that can be advertised

MAX_ENCODED_LENGTH = 31

class **Types**(*value*)

Bases: Enum

An enumeration.

flags = 1

service_16bit_uuid_more_available = 2

service_16bit_uuid_complete = 3

service_32bit_uuid_more_available = 4

service_32bit_uuid_complete = 5

service_128bit_uuid_more_available = 6

service_128bit_uuid_complete = 7

short_local_name = 8

complete_local_name = 9

tx_power_level = 10

class_of_device = 13

simple_pairing_hash_c = 14

simple_pairing_randimizer_r = 15

security_manager_tk_value = 16

security_manager_oob_flags = 17

slave_connection_interval_range = 18

solicited_sevice_uuids_16bit = 20

solicited_sevice_uuids_128bit = 21

service_data = 22

public_target_address = 23

random_target_address = 24

appearance = 25

advertising_interval = 26

le_bluetooth_device_address = 27

le_role = 28

simple_pairng_hash_c256 = 29

simple_pairng_randomizer_r256 = 30

service_data_32bit_uuid = 32

service_data_128bit_uuid = 33

uri = 36

information_3d_data = 61

manufacturer_specific_data = 255

property flags: Optional[int]

The advertising flags in the payload, if set

Getter
Gets the advertising flags in the payload, or None if not set

Setter
Sets the advertising flags in the payload

Delete
Removes the advertising flags from the payload

property service_data: Optional[Union[bytes, List[int]]]

The service data in the payload, if set

Getter
Gets the service data in the payload, or None if not set

Setter
Sets the service data for the payload

Delete
Removes the service data from the payload

property manufacturer_data: Optional[Union[bytes, List[int]]]

The manufacturer data in the payload, if set

Getter
Gets the manufacturer data in the payload, or None if not set

Setter
Sets the manufacturer data for the payload

Delete
Removes the manufacturer data from the payload

property service_uuids: List[*Uuid*]

Gets all of the 16-bit and 128-bit service UUIDs specified in the advertising data

check_encoded_length()

Checks if the encoded length of this advertising data payload meets the maximum allowed length specified by the Bluetooth specification

Return type
Tuple[int, bool]

Returns
a tuple of the encoded length and a bool result of whether or not it meets requirements

to_ble_adv_data()

Converts the advertising data to a BLEAdvData object that can be used by the nRF Driver

Returns
the BLEAdvData object which represents this advertising data

Return type
nrf_types.BLEAdvData

to_bytes()

Converts the advertising data to the encoded bytes that will be advertised over the air. Advertising payloads are encoded in a length-type-value format

Return type
bytes

Returns
The encoded payload

classmethod from_ble_adv_records(*advertise_records*)

Converts a dictionary of AdvertisingData.Type: value keypairs into an object of this class

Parameters
advertise_records (*dict*) – a dictionary mapping the advertise data types to their corresponding values

Returns
the AdvertisingData from the records given

Return type
AdvertisingData

class blatann.gap.advertise_data.**ScanReport**(*adv_report*, *resolved_address*)

Bases: object

Represents a payload and associated metadata that's received during scanning

property device_name: str

Read Only

The name of the device, pulled from the advertising data (if advertised) or uses the Peer's MAC Address if not set

property is_bonded_device: bool

If the scan report is from a BLE device that the local device has a matching bond database entry

property resolved_address: Optional[*PeerAddress*]

If the scan report is from a bonded device, this is the resolved public/static/random BLE address. This may be the same as peer_addr if the device is not advertising as a private resolvable address

update(*adv_report*)

Used internally to merge a new advertising payload that was received into the current scan report

class blatann.gap.advertise_data.**ScanReportCollection**

Bases: object

Collection of all the advertising data and scan reports found in a scanning session

property advertising_peers_found: Iterable[*ScanReport*]

Gets the list of scans which have been combined and condensed into a list where each entry is a unique peer. The scan reports in this list represent aggregated data of each advertising packet received by the advertising device, such that later advertising packets will update/overwrite packet attributes received from earlier packets, if the data has been modified.

Returns
The list of scan reports, with each being a unique peer

property all_scan_reports: Iterable[*ScanReport*]

Gets the list of all of the individual advertising packets received.

Returns
The list of all scan reports

get_report_for_peer(*peer_addr*)

Gets the combined/aggregated scan report for a given Peer's address. If the peer's scan report isn't found, returns None

Parameters
peer_addr – The peer's address to search for

Return type
Optional[*ScanReport*]

Returns
The associated scan report, if found

clear()

Clears out all of the scan reports cached

update(*adv_report*, *resolved_peer_addr=None*)

Used internally to update the collection with a new advertising report received

Return type
ScanReport

Returns
The Scan Report created from the advertising report

blatann.gap.advertising module

class blatann.gap.advertising.**Advertiser**(*ble_device*, *client*, *conn_tag=0*)

Bases: object

Class which manages the advertising state of the BLE Device

ADVERTISE_FOREVER = 0

Special value used to indicate that the BLE device should advertise indefinitely until either a central is connected or stopped manually.

property on_advertising_timeout: *Event*[*Advertiser*, None]

Event generated whenever advertising times out and finishes with no connections made

Note: If auto-restart advertising is enabled, this will trigger on each advertising timeout configured

Returns
an Event which can have handlers registered to and deregistered from

property is_advertising: bool

Read Only

Current state of advertising

property min_interval_ms: float

Read Only

The minimum allowed advertising interval, in millseconds. This is defined by the Bluetooth specification.

property max_interval_ms: float

Read Only

The maximum allowed advertising interval, in milliseconds. This is defined by the Bluetooth specification.

property auto_restart: bool

Enables/disables whether or not the device should automatically restart advertising when an advertising timeout occurs or the client is disconnected.

Note: Auto-restart is disabled automatically when *stop()* is called

Getter
Gets the auto-restart flag

Setter
Sets/clears the auto-restart flag

set_channel_mask(*ch37_enabled=True*, *ch38_enabled=True*, *ch39_enabled=True*)

Enables/disables which channels advertising packets are sent out on. By default, all 3 channels (37, 38, 39) are enabled. At least one of the 3 channels MUST be enabled, otherwise a ValueError exception will be raised.

This mask will take effect the next time advertising is started or restarted due to timeout/disconnect.

Parameters

- **ch37_enabled** – True to enable advertising on channel 37, False to disable
- **ch38_enabled** – True to enable advertising on channel 38, False to disable
- **ch39_enabled** – True to enable advertising on channel 39, False to disable

set_advertise_data(*advertise_data=AdvertisingData()*, *scan_response=AdvertisingData()*)

Sets the advertising and scan response data which will be broadcasted to peers during advertising

---

Note: BLE Restricts advertise and scan response data to an encoded length of 31 bytes each. Use `AdvertisingData.check_encoded_length()` to determine if the payload is too large

---

Parameters

- **advertise_data** (*AdvertisingData*) – The advertising data to use
- **scan_response** (*AdvertisingData*) – The scan response data to use. This data is only sent when a scanning device requests the scan response packet (active scanning)

Raises

InvalidOperationException if one of the payloads is too large

set_default_advertise_params(*advertise_interval_ms*, *timeout_seconds*, *advertise_mode=BLEGapAdvType.connectable_undirected*)

Sets the default advertising parameters so they do not need to be specified on each start

Parameters

- **advertise_interval_ms** (`float`) – The advertising interval, in milliseconds. Should be a multiple of 0.625ms, otherwise it'll be rounded down to the nearest 0.625ms
- **timeout_seconds** (`int`) – How long to advertise for before timing out, in seconds. For no timeout, use ADVERTISE_FOREVER (0)
- **advertise_mode** (*BLEGapAdvType*) – The mode the advertiser should use

start(*adv_interval_ms=None*, *timeout_sec=None*, *auto_restart=None*, *advertise_mode=None*)

Starts advertising with the given parameters. If none given, will use the default set through *set_default_advertise_params()*

Parameters

- **adv_interval_ms** (`Optional[float]`) – The interval at which to send out advertise packets, in milliseconds. Should be a multiple of 0.625ms, otherwise it'll be round down to the nearest 0.625ms
- **timeout_sec** (`Optional[int]`) – The duration which to advertise for. For no timeout, use ADVERTISE_FOREVER (0)
- **auto_restart** (`Optional[bool]`) – Flag indicating that advertising should restart automatically when the timeout expires, or when the client disconnects
- **advertise_mode** (`Optional[`*BLEGapAdvType*`]`) – The mode the advertiser should use

Returns

A waitable that will expire either when the timeout occurs or a client connects. The waitable will return either `None` on timeout or *Client* on successful connection

Return type

ClientConnectionWaitable

stop()

Stops advertising and disables the auto-restart functionality (if enabled)

blatann.gap.bond_db module

class blatann.gap.bond_db.**BondingData**(*own_ltk*, *peer_ltk*, *peer_id*, *peer_sign*)

Bases: object

classmethod from_keyset(*bonding_keyset*)

to_dict()

classmethod from_dict(*data*)

class blatann.gap.bond_db.**BondDbEntry**(*entry_id=0*)

Bases: object

resolved_peer_address()

Return type
PeerAddress

matches_peer(*own_address*, *peer_address*, *peer_is_client*, *master_id=None*)

Return type
bool

peer_address_matches_or_resolves(*peer_address*)

Return type
bool

to_dict()

classmethod from_dict(*data*)

class blatann.gap.bond_db.**BondDatabase**

Bases: object

create()

Return type
BondDbEntry

add(*db_entry*)

update(*db_entry*)

delete(*db_entry*)

delete_all()

find_entry(*own_address*, *peer_address*, *peer_is_client*, *master_id=None*)

class blatann.gap.bond_db.**BondDatabaseLoader**

Bases: object

load()

Return type
BondDatabase

save(*db*)

blatann.gap.default_bond_db module

class blatann.gap.default_bond_db.**DatabaseStrategy**

Bases: object

Abstract base class defining the methods and properties for serializing/deserializing bond databases into different formats

property file_extension: str

The file extension that this strategy can serialize/deserialize

load(*filename*)

Loads/deserializes a database file

Parameters
filename (str) – Name of the file to deserialize

Return type
DefaultBondDatabase

Returns
The loaded bond database

save(*filename*, *db*)

Saves/serializes a database to a file

Parameters

- **filename** (str) – Filename to save the database to
- **db** (*DefaultBondDatabase*) – The database object serialize

class blatann.gap.default_bond_db.**JsonDatabaseStrategy**

Bases: *DatabaseStrategy*

Strategy for serializing/deseriralizing bond databases in JSON format

property file_extension: str

The file extension that this strategy can serialize/deserialize

load(*filename*)

Loads/deserializes a database file

Parameters
filename – Name of the file to deserialize

Return type
DefaultBondDatabase

Returns
The loaded bond database

save(*filename*, *db*)

Saves/serializes a database to a file

Parameters

- **filename** (str) – Filename to save the database to
- **db** (*DefaultBondDatabase*) – The database object serialize

class blatann.gap.default_bond_db.**PickleDatabaseStrategy**

Bases: *DatabaseStrategy*

Strategy for serializing/deserializing bond databases in pickle format

property file_extension: str

The file extension that this strategy can serialize/deserialize

load(*filename*)

Loads/deserializes a database file

Parameters
filename – Name of the file to deserialize

Return type
DefaultBondDatabase

Returns
The loaded bond database

save(*filename*, *db*)

Saves/serializes a database to a file

Parameters

- **filename** – Filename to save the database to
- **db** (*DefaultBondDatabase*) – The database object serialize

blatann.gap.default_bond_db.**database_strategies = [<blatann.gap.default_bond_db.PickleDatabaseStrategy object>, <blatann.gap.default_bond_db.JsonDatabaseStrategy object>]**

List of supported database strategies

blatann.gap.default_bond_db.**database_strategies_by_extension: Dict[str, *DatabaseStrategy*] = {'.json': <blatann.gap.default_bond_db.JsonDatabaseStrategy object>, '.pkl': <blatann.gap.default_bond_db.PickleDatabaseStrategy object>}**

Mapping of database file extensions to their respective strategies

class blatann.gap.default_bond_db.**DefaultBondDatabaseLoader**(*filename='user'*)

Bases: *BondDatabaseLoader*

migrate_to_json(*base_filename*)

load()

Return type
DefaultBondDatabase

save(*db*)

class blatann.gap.default_bond_db.**DefaultBondDatabase**(*records=None*)

Bases: *BondDatabase*

create()

add(*db_entry*)

update(*db_entry*)

delete(*db_entry*)

delete_all()

find_entry(*own_address*, *peer_address*, *peer_is_client*, *master_id=None*)

Attempts to find a bond entry which satisfies the parameters provided

Parameters

- **own_address** (*PeerAddress*) – The local device's BLE address
- **peer_address** (*PeerAddress*) – The peer's BLE address
- **peer_is_client** (bool) – Flag indicating the role of the peer. True if the peer is a client/central, False if the peer is a server/peripheral
- **master_id** (Optional[*BLEGapMasterId*]) – If during a security info request, this is the Master ID provided by the peer to search for

Return type

Optional[*BondDbEntry*]

Returns

The first entry that satisfies the above parameters, or None if no entry was found

blatann.gap.default_bond_db.**migrate_bond_database**(*from_file*, *to_file*)

Migrates a bond database file from one format to another.

For supported extensions/formats, check `database_strategies_by_extension.keys()`

Parameters

- **from_file** (str) – File to migrate from
- **to_file** (str) – File to migrate to

blatann.gap.gap_types module

class blatann.gap.gap_types.**Phy**(*value*)

Bases: IntFlag

The supported PHYs

Note: Coded PHY is currently not supported (hardware limitation)

auto = 0

Automatically select the PHY based on what's supported

one_mbps = 1

1 Mbps PHY

two_mbps = 2

2 Mbps PHY

class blatann.gap.gap_types.**PeerAddress**(*addr_type*, *addr*)

Bases: *BLEGapAddr*

class blatann.gap.gap_types.**ConnectionParameters**(*min_conn_interval_ms*, *max_conn_interval_ms*, *timeout_ms*, *slave_latency=0*)

Bases: *BLEGapConnParams*

Represents the connection parameters that are sent during negotiation. This includes the preferred min/max interval range, timeout, and slave latency

class blatann.gap.gap_types.**ActiveConnectionParameters**(*conn_params*)

Bases: object

Represents the connection parameters that are currently in use with a peer device. This is similar to ConnectionParameters with the sole difference being the connection interval is not a min/max range but a single number

property interval_ms: float

Read Only

The connection interval, in milliseconds

property timeout_ms: float

Read Only

The connection timeout, in milliseconds

property slave_latency: int

Read Only

The slave latency (the number of connection intervals the slave is allowed to skip before being required to respond)

blatann.gap.generic_access_service module

class blatann.gap.generic_access_service.**GenericAccessService**(*ble_driver*, *device_name='nRF5x'*, *appearance=Appearance.unknown*)

Bases: object

Class which represents the Generic Access service within the local database

DEVICE_NAME_MAX_LENGTH = 31

property device_name: str

The device name that is configured in the Generic Access service of the local GATT database

Getter
Gets the current device name

Setter
Sets the current device name. Length (after utf8 encoding) must be <= 31 bytes

property appearance: *Appearance*

The Appearance that is configured in the Generic Access service of the local GATT database

Getter
Gets the device appearance

Setter
Sets the device appearance

property preferred_peripheral_connection_params: Optional[*ConnectionParameters*]
The preferred peripheral connection parameters that are configured in the Generic Access service of the local GATT Database. If not configured, returns None.

Getter
Gets the configured connection parameters or None if not configured

Setter
Sets the configured connection parameters

update()
Not to be called by users

Used internally to configure the generic access in the case that values were set before the driver was opened and configured.

blatann.gap.scanning module

class blatann.gap.scanning.**ScanParameters**(*interval_ms*, *window_ms*, *timeout_s*, *active=True*)
Bases: *BLEGapScanParams*

Class which holds scanning parameters

validate()

update(*window_ms*, *interval_ms*, *timeout_s*, *active*)

class blatann.gap.scanning.**Scanner**(*ble_device*)
Bases: object

property on_scan_received: *Event*[*Scanner*, *ScanReport*]
Event that is raised whenever a scan report is received

property on_scan_timeout: *Event*[*Scanner*, *ScanReportCollection*]
Event that is raised when scanning completes/times out

property is_scanning: bool
Read Only

Current state of scanning

set_default_scan_params(*interval_ms=200*, *window_ms=150*, *timeout_seconds=10*, *active_scanning=True*)
Sets the default scan parameters so they do not have to be specified each time a scan is started. Reference the Bluetooth specification for valid ranges for parameters.

Parameters

- **interval_ms** (float) – The interval which to scan for advertising packets, in milliseconds
- **window_ms** (float) – How long within a single scan interval to be actively listening for advertising packets, in milliseconds
- **timeout_seconds** (int) – How long to advertise for, in seconds
- **active_scanning** (bool) – Whether or not to fetch scan response packets from advertisers

start_scan(*scan_parameters=None*, *clear_scan_reports=True*)

Starts a scan and returns a waitable for when the scan completes

Parameters

- **scan_parameters** (Optional[*ScanParameters*]) – Optional scan parameters. Uses default if not specified
- **clear_scan_reports** – Flag to clear out previous scan reports

Return type
ScanFinishedWaitable

Returns
A Waitable which will trigger once the scan finishes based on the timeout specified. Waitable returns a ScanReportCollection of the advertising packets found

stop()

Stops scanning

blatann.gap.smp module

class blatann.gap.smp.**SecurityLevel**(*value*)

Bases: Enum

Security levels used for defining GATT server characteristics

NO_ACCESS = 0

OPEN = 1

JUST_WORKS = 2

MITM = 3

LESC_MITM = 4

class blatann.gap.smp.**PairingPolicy**(*value*)

Bases: IntFlag

An enumeration.

allow_all = 0

reject_new_pairing_requests = 1

reject_nonbonded_peripheral_requests = 2

reject_bonded_peripheral_requests = 4

reject_bonded_device_repairing_requests = 8

reject_peripheral_requests = 6

reject_all_requests = 15

static combine(**policies*)

class blatann.gap.smp.**SecurityParameters**(*passcode_pairing=False*, *io_capabilities=BLEGapIoCaps.KEYBOARD_DISPLAY*, *bond=False*, *out_of_band=False*, *reject_pairing_requests=False*, *lesc_pairing=False*)

Bases: object

Class representing the desired security parameters for a given connection

class blatann.gap.smp.**SecurityManager**(*ble_device*, *peer*, *security_parameters*)

Bases: object

Handles performing security procedures with a connected peer

property on_pairing_complete: *Event*[*Peer*, *PairingCompleteEventArgs*]

Event that is triggered when pairing completes with the peer

Returns
an Event which can have handlers registered to and deregistered from

property on_security_level_changed: *Event*[*Peer*, *SecurityLevelChangedEventArgs*]

Event that is triggered when the security/encryption level changes. This can be triggered from a pairing sequence or if a bonded client starts the encryption handshaking using the stored LTKs.

Note: This event is triggered before on_pairing_complete

Returns
an Event which can have handlers registered to and deregestestered from

property on_passkey_display_required: *Event*[*Peer*, *PasskeyDisplayEventArgs*]

Event that is triggered when a passkey needs to be displayed to the user and depending on the pairing mode the user must confirm that keys match (PasskeyDisplayEventArgs.match_request == True).

Note: If multiple handlers are registered to this event, the first handler which resolves the match confirmation will set the response. All others will be ignored.

Returns
an Event which can have handlers registered to and deregistered from

Return type
Event

property on_passkey_required: *Event*[*Peer*, *PasskeyEntryEventArgs*]

Event that is triggered when a passkey needs to be entered by the user

---

Note: If multiple handlers are registered to this event, the first handler which resolves the passkey will set the value. All others will be ignored.

Returns
an Event which can have handlers registered to and deregistered from

property on_peripheral_security_request: *Event*[*Peer*, *PeripheralSecurityRequestEventArgs*]

Event that is triggered when the connected peripheral explicitly requests pairing/encryption to be enabled. The event provides the higher levels an opportunity to accept, reject, or force re-pair with the peripheral.

If no handler is registered to this event, pairing requests will be accepted unless the reject_pairing_requests parameter is set.

Note: If a handler is registered to this event, it **must** respond with one of the options (accept/reject/repair).

Note: If multiple handlers are registered to this event, the first handler to respond is the response used. All other inputs will be ignored

Returns
Event that is triggered when the peripheral requests a secure connection

property on_pairing_request_rejected: *Event*[*Peer*, *PairingRejectedEventArgs*]

Event that's emitted when a pairing request is rejected locally, either due to the user event handler or due to the rejection policy set in the security parameters

Returns
Event that is triggered when a pairing request is rejected

property is_previously_bonded: bool

Gets if the peer this security manager is for was bonded in a previous connection

Returns
True if previously bonded, False if not

property pairing_in_process: bool

Gets whether or not pairing/encryption is currently in process

property security_level: *SecurityLevel*

Gets the current security level of the connection

property security_params: *SecurityParameters*

Gets the security parameters structure

set_security_params(*passcode_pairing*, *io_capabilities*, *bond*, *out_of_band*, *reject_pairing_requests=False*, *lesc_pairing=False*)

Sets the security parameters to use with the peer

Parameters

- **passcode_pairing** (bool) – Flag indicating that passcode pairing is required
- **io_capabilities** (*BLEGapIoCaps*) – The input/output capabilities of this device
- **bond** (bool) – Flag indicating that long-term bonding should be performed
- **out_of_band** (bool) – Flag indicating if out-of-band pairing is supported
- **reject_pairing_requests** (Union[bool, *PairingPolicy*]) – Flag indicating that all security requests by the peer should be rejected
- **lesc_pairing** (bool) – Flag indicating that LE Secure Pairing methods are supported

pair(*force_repairing=False*)

Starts the pairing process with the peer with the set security parameters.

If the peer is already bonded, initiates the encryption process unless force_repairing is set to True

If the peer is a central and we are a local device, sends the peripheral security request to the central so they can start the pairing/encryption process

Return type
EventWaitable[*Peer*, *PairingCompleteEventArgs*]

Returns
A waitable that will trigger when pairing is complete

`use_debug_lesc_key()`

Changes the security settings to use the debug public/private key-pair for future LESC pairing interactions. The key is defined in the Core Bluetooth Specification v4.2 Vol.3, Part H, Section 2.3.5.6.

> **Warning:** Using this key allows Bluetooth sniffers to be able to decode the encrypted traffic over the air

`delete_bonding_data()`

Deletes the bonding data for the peer, if any. Cannot be called during pairing, will throw an InvalidOperationException

blatann.gap.smp_crypto module

`blatann.gap.smp_crypto.lesc_pubkey_to_raw(public_key, little_endian=True)`

Converts from a python public key to the raw (x, y) bytes for the nordic

Return type
bytearray

`blatann.gap.smp_crypto.lesc_privkey_to_raw(private_key, little_endian=True)`

Return type
bytearray

`blatann.gap.smp_crypto.lesc_pubkey_from_raw(raw_key, little_endian=True)`

Converts from raw (x, y) bytes to a public key that can be used for the DH request

Return type
EllipticCurvePublicKey

`blatann.gap.smp_crypto.lesc_privkey_from_raw(raw_priv_key, raw_pub_key, little_endian=True)`

Return type
EllipticCurvePrivateKey

`blatann.gap.smp_crypto.lesc_generate_private_key()`

Generates a new private key that can be used for LESC pairing

Return type
EllipticCurvePrivateKey

Returns
The generated private key

`blatann.gap.smp_crypto.lesc_compute_dh_key(private_key, peer_public_key, little_endian=False)`

Computes the DH key for LESC pairing given our private key and the peer's public key

Parameters

- **private_key** (EllipticCurvePrivateKey) – Our private key
- **peer_public_key** (EllipticCurvePublicKey) – The peer's public key
- **little_endian** – whether or not to return the shared secret in little endian

Return type
bytes

Returns
The shared secret

blatann.gap.smp_crypto.**ble_ah**(*key*, *p_rand*)

Function for calculating the ah() hash function described in Bluetooth core specification 4.2 section 3.H.2.2.2.

This is used for resolving private addresses where a private address is prand[3] || aes-128(irk, prand[3]) % 2^24

Parameters

- **key** (bytes) – the IRK to use, in big endian format
- **p_rand** (bytes) – The random component, first 3 bytes of the address

Return type
bytes

Returns
The last 3 bytes of the encrypted hash

blatann.gap.smp_crypto.**private_address_resolves**(*peer_addr*, *irk*)

Checks if the given peer address can be resolved with the IRK

Private Resolvable Peer Addresses are in the format [4x:xx:xx:yy:yy:yy], where 4x:xx:xx is a random number hashed with the IRK to generate yy:yy:yy This function checks if the random number portion hashed with the IRK equals the hashed part of the address

Parameters

- **peer_addr** (*PeerAddress*) – The peer address to check
- **irk** (bytes) – The identity resolve key to try

Return type
bool

Returns
True if it resolves, False if not

blatann.gatt package

blatann.gatt.**logger = <Logger blatann.gatt (INFO)>**

The default MTU size that's used when a connection is established

blatann.gatt.**MTU_SIZE_DEFAULT = 23**

The minimum allowed MTU size

blatann.gatt.**MTU_SIZE_MINIMUM = 23**

The ideal MTU size to use when using the maximum link-layer Data Length Extension setting (251)

blatann.gatt.**DLE_OVERHEAD = 4**

Status codes that can be returned during GATT Operations (reads, writes, etc.)

blatann.gatt.**GattStatusCode**

The two notification types (notification, indication) used when a characteristic is notified from a peripheral

class blatann.gatt.**ServiceType**(*value*)

Bases: IntEnum

An enumeration.

PRIMARY = 1

SECONDARY = 2

class blatann.gatt.**SubscriptionState**(*value*)

Bases: IntEnum

Defines the different subscription states/types for a characteristic

NOT_SUBSCRIBED = 0

NOTIFY = 1

INDICATION = 2

classmethod to_buffer(*value*)

Converts to a little-endian uint16 buffer to be written over BLE

classmethod from_buffer(*buf*)

Converts from a little-endian uint16 buffer received over BLE to the subscription state

class blatann.gatt.**CharacteristicProperties**(*read=True*, *write=False*, *notify=False*, *indicate=False*, *broadcast=False*, *write_no_response=False*, *signed_write=False*)

Bases: object

class blatann.gatt.**Attribute**(*uuid*, *handle*, *value=b''*, *string_encoding='utf8'*)

Bases: object

Represents a single attribute which lives inside a Characteristic (both remote and local)

property uuid: *Uuid*

The attribute's UUID

property handle: int

The attribute's handle

property value: bytes

Gets the current value of the attribute

property string_encoding: str

The default method for encoding strings into bytes when a string is provided as a value

class blatann.gatt.**Characteristic**(*ble_device*, *peer*, *uuid*, *properties*, *attributes=None*, *default_string_encoding='utf8'*)

Bases: object

Abstract class that represents a BLE characteristic (both remote and local).

class blatann.gatt.**Service**(*ble_device*, *peer*, *uuid*, *service_type*, *start_handle=0*, *end_handle=0*)

Bases: object

Abstract class that represents a BLE Service (both remote and local)

class blatann.gatt.**GattDatabase**(*ble_device*, *peer*)

Bases: object

Abstract class that represents a BLE Database (both remote and local)

class blatann.gatt.**PresentationFormat**(*fmt*, *exponent*, *unit*, *namespace=0*, *description=0*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Int8'>, <class 'blatann.services.ble_data_types.Uint16'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint16'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

static try_get_enum(*value*, *enum_type*)

Submodules

blatann.gatt.gattc module

class blatann.gatt.gattc.**GattcCharacteristic**(*ble_device*, *peer*, *uuid*, *properties*, *decl_attr*, *value_attr*, *cccd_attr=None*, *attributes=None*)

Bases: *Characteristic*

Represents a characteristic that lives within a service in the server's GATT database.

This class is normally not instantiated directly and instead created when the database is discovered via *Peer.discover_services()*

property declaration_attribute: *GattcAttribute*

Read Only

Gets the declaration attribute of the characteristic

property value_attribute: *GattcAttribute*

Read Only

Gets the value attribute of the characteristic

property value: bytes

Read Only

The current value of the characteristic. This is updated through read, write, and notify operations

property readable: bool

Read Only

Gets if the characteristic can be read from

property writable: bool

Read Only

Gets if the characteristic can be written to

property writable_without_response: bool

Read Only

Gets if the characteristic accepts write commands that don't require a confirmation response

property subscribable: bool

Read Only

Gets if the characteristic can be subscribed to

property subscribable_indications: bool

Read Only

Gets if the characteristic can be subscribed to using indications

property subscribable_notifications: bool

Read Only

Gets if the characteristic can be subscribed to using notifications

property subscribed: bool

Read Only

Gets if the characteristic is currently subscribed to

property attributes: Iterable[*GattcAttribute*]

Read Only

Returns the list of all attributes/descriptors that reside in the characteristic. This includes the declaration attribute, value attribute, and descriptors (CCCD, Name, etc.)

property string_encoding: str

The default method for encoding strings into bytes when a string is provided as a value

Getter
Gets the current string encoding for the characteristic

Setter
Sets the string encoding for the characteristic

property on_read_complete: *Event*[*GattcCharacteristic*, *ReadCompleteEventArgs*]

Event that is raised when a read operation from the characteristic is completed

property on_write_complete: *Event*[*GattcCharacteristic*, *WriteCompleteEventArgs*]

Event that is raised when a write operation to the characteristic is completed

property on_notification_received: *Event*[*GattcCharacteristic*, *NotificationReceivedEventArgs*]

Event that is raised when an indication or notification is received on the characteristic

subscribe(*on_notification_handler*, *prefer_indications=False*)

Subscribes to the characteristic's indications or notifications, depending on what's available and the prefer_indications setting. Returns a Waitable that triggers when the subscription on the peripheral finishes.

Parameters

- **on_notification_handler** (Callable[[*GattcCharacteristic*, *NotificationReceivedEventArgs*], None]) – The handler to be called when an indication or notification is received from the peripheral. Must take two parameters: (GattcCharacteristic this, NotificationReceivedEventArgs event args)
- **prefer_indications** – If the peripheral supports both indications and notifications, will subscribe to indications instead of notifications

Return type
EventWaitable[*GattcCharacteristic*, *SubscriptionWriteCompleteEventArgs*]

Returns
A Waitable that will trigger when the subscription finishes

Raises
InvalidOperationException if the characteristic cannot be subscribed to (characteristic does not support indications or notifications)

unsubscribe()

Unsubscribes from indications and notifications from the characteristic and clears out all handlers for the characteristic's on_notification event handler. Returns a Waitable that triggers when the unsubscription finishes.

Return type
EventWaitable[*GattcCharacteristic*, *SubscriptionWriteCompleteEventArgs*]

Returns
A Waitable that will trigger when the unsubscription operation finishes

Raises
InvalidOperationException if characteristic cannot be subscribed to (characteristic does not support indications or notifications)

read()

Initiates a read of the characteristic and returns a Waitable that triggers when the read finishes with the data read.

Return type
EventWaitable[*GattcCharacteristic*, *ReadCompleteEventArgs*]

Returns
A waitable that will trigger when the read finishes

Raises
InvalidOperationException if characteristic not readable

write(*data*)

Performs a write request of the data provided to the characteristic and returns a Waitable that triggers when the write completes and the confirmation response is received from the other device.

Parameters
data (*str or bytes or bytearray*) – The data to write. Can be a string, bytes, or anything that can be converted to bytes

Return type
EventWaitable[*GattcCharacteristic*, *WriteCompleteEventArgs*]

Returns
A waitable that returns when the write finishes

Raises
InvalidOperationException if characteristic is not writable

`write_without_response`(*data*)

Performs a write command, which does not require the peripheral to send a confirmation response packet. This is a faster but lossy operation in the case that the packet is dropped/never received by the peer. This returns a waitable that triggers when the write is transmitted to the peripheral device.

Note: Data sent without responses must fit within a single MTU minus 3 bytes for the operation overhead.

Parameters
data (*str or bytes or bytearray*) – The data to write. Can be a string, bytes, or anything that can be converted to bytes

Return type
EventWaitable[*GattcCharacteristic*, *WriteCompleteEventArgs*]

Returns
A waitable that returns when the write finishes

Raises
InvalidOperationException if characteristic is not writable without responses

`find_descriptor`(*uuid*)

Searches for the descriptor/attribute matching the UUID provided and returns the attribute. If not found, returns None. If multiple attributes with the same UUID exist in the characteristic, this returns the first attribute found.

Parameters
uuid (*Uuid*) – The UUID to search for

Return type
Optional[*GattcAttribute*]

Returns
THe descriptor attribute, if found

`class blatann.gatt.gattc.GattcService`(*ble_device*, *peer*, *uuid*, *service_type*, *start_handle=0*, *end_handle=0*)

Bases: *Service*

Represents a service that lives within the server's GATT database.

This class is normally not instantiated directly and instead created when the database is discovered via *Peer.discover_services()*

`property characteristics: List[GattcCharacteristic]`

Gets the list of characteristics within the service

`find_characteristic`(*characteristic_uuid*)

Finds the characteristic matching the given UUID inside the service. If not found, returns None. If multiple characteristics with the same UUID exist within the service, this will return the first one found.

Parameters
characteristic_uuid (*Uuid*) – The UUID of the characteristic to find

Return type
Optional[*GattcCharacteristic*]

Returns
The characteristic if found, otherwise None

class blatann.gatt.gattc.**GattcDatabase**(*ble_device*, *peer*, *write_no_resp_queue_size=1*)

Bases: *GattDatabase*

Represents a remote GATT Database which lives on a connected peripheral. Contains all discovered services, characteristics, and descriptors

property services: List[*GattcService*]

Gets the list of services within the database

find_service(*service_uuid*)

Finds the service matching the given UUID inside the database. If not found, returns None. If multiple services with the same UUID exist in the database, this will return the first service found.

Parameters
service_uuid (*Uuid*) – The UUID of the service to find

Return type
Optional[*GattcService*]

Returns
The service if found, otherwise None

find_characteristic(*characteristic_uuid*)

Finds the characteristic matching the given UUID inside the database. If not found, returns None. If multiple characteristics with the same UUID exist in the database, this will return the first characteristic found.

Parameters
characteristic_uuid (blatann.uuid.Uuid) – The UUID of the characteristic to find

Returns
The characteristic if found, otherwise None

Return type
GattcCharacteristic

iter_characteristics()

Iterates through all the characteristics in the database

Return type
Iterable[*GattcCharacteristic*]

Returns
An iterable of the characterisitcs in the database

blatann.gatt.gattc_attribute module

class blatann.gatt.gattc_attribute.**GattcAttribute**(*uuid*, *handle*, *read_write_manager*, *initial_value=b''*, *string_encoding='utf8'*)

Bases: *Attribute*

Represents a client-side interface to a single attribute which lives inside a Characteristic

property on_read_complete: *Event*[*GattcAttribute*, *ReadCompleteEventArgs*]

Event that is triggered when a read from the attribute is completed

property on_write_complete: *Event*[*GattcAttribute*, *WriteCompleteEventArgs*]

Event that is triggered when a write to the attribute is completed

read()

Performs a read of the attribute and returns a Waitable that executes when the read finishes with the data read.

Return type
IdBasedEventWaitable[*GattcAttribute*, *ReadCompleteEventArgs*]

Returns
A waitable that will trigger when the read finishes

write(*data*, *with_response=True*)

Initiates a write of the data provided to the attribute and returns a Waitable that executes when the write completes and the confirmation response is received from the other device.

Parameters

- **data** (*str or bytes or bytearray*) – The data to write. Can be a string, bytes, or anything that can be converted to bytes
- **with_response** – Used internally for characteristics that support write without responses. Should always be true for any other case (descriptors, etc.).

Return type
IdBasedEventWaitable[*GattcAttribute*, *WriteCompleteEventArgs*]

Returns
A waitable that returns when the write finishes

update(*value*)

Used internally to update the value after data is received from another means, i.e. Indication/notification. Should not be called by the user.

blatann.gatt.gatts module

class blatann.gatt.gatts.**GattsUserDescriptionProperties**(*value*, *write=False*, *security_level=SecurityLevel.OPEN*, *max_length=0*, *variable_length=False*)

Bases: *GattsAttributeProperties*

Properties used to configure the User Description characteristic descriptor.

The most basic, set-once, read-only usage of this is `GattsUserDescriptionProperties("my description")`

class blatann.gatt.gatts.**GattsCharacteristicProperties**(*read=True*, *write=False*, *notify=False*, *indicate=False*, *broadcast=False*, *write_no_response=False*, *signed_write=False*, *security_level=SecurityLevel.OPEN*, *max_length=20*, *variable_length=True*, *sccd=False*, *user_description=None*, *presentation_format=None*, *cccd_write_security_level=SecurityLevel.OPEN*)

Bases: *CharacteristicProperties*

Properties for Gatt Server characeristics

class blatann.gatt.gatts.**GattsCharacteristic**(*ble_device*, *peer*, *uuid*, *properties*, *value_handle*, *cccd_handle*, *sccd_handle*, *user_desc_handle*, *notification_manager*, *value=b''*, *prefer_indications=True*, *string_encoding='utf8'*)

Bases: *Characteristic*

Represents a single characteristic within a service. This class is usually not instantiated directly; it is added to a service through *GattsService.add_characteristic()*

set_value(*value*, *notify_client=False*)

Sets the value of the characteristic.

Parameters

- **value** – The value to set to. Must be an iterable type such as a str, bytes, or list of uint8 values, or a BleDataStream object. Length must be less than or equal to the characteristic's max length. If a string is given, it will be encoded using the string_encoding property of the characteristic.
- **notify_client** – Flag whether or not to notify the client. If indications and notifications are not set up for the characteristic, will raise an InvalidOperationException

Raises
InvalidOperationException if value length is too long, or notify client set and characteristic is not notifiable

Raises
InvalidStateException if the client is not currently subscribed to the characteristic

Return type
Optional[*IdBasedEventWaitable*[*GattsCharacteristic*, *NotificationCompleteEventArgs*]]

Returns
If notify_client is true, this method will return the waitable for when the notification is sent to the client

notify(*data*)

Notifies the client with the data provided without setting the data into the characteristic value. If data is not provided (None), will notify with the currently-set value of the characteristic

Parameters
data – Optional data to notify the client with. If supplied, must be an iterable type such as a str, bytes, or list of uint8 values, or a BleDataStream object. Length must be less than or equal to the characteristic's max length. If a string is given, it will be encoded using the string_encoding property of the characteristic.

Raises
InvalidStateException if the client is not subscribed to the characteristic

Raises
InvalidOperationException if the characteristic is not configured for notifications/indications

Return type
IdBasedEventWaitable[*GattsCharacteristic*, *NotificationCompleteEventArgs*]

Returns
An EventWaitable that will trigger when the notification is successfully sent to the client. The waitable also contains the ID of the sent notification which is used in the on_notify_complete event

add_descriptor(*uuid*, *properties*, *initial_value=b''*, *string_encoding='utf8'*)

Creates and adds a descriptor to the characteristic

Note: Due to limitations of the BLE stack, the CCCD, SCCD, User Description, Extended Properties, and Presentation Format descriptors cannot be added through this method. They must be added through the `GattsCharacteristicProperties` fields when creating the characteristic.

Parameters

- **uuid** (*Uuid*) – The UUID of the descriptor to add, and cannot be the UUIDs of any of the reserved descriptor UUIDs in the note
- **properties** (*GattsAttributeProperties*) – The properties of the descriptor
- **initial_value** – The initial value to set the descriptor to
- **string_encoding** – The string encoding to use, if a string is set

Return type
GattsAttribute

Returns
the descriptor that was created and added to the characteristic

add_constant_value_descriptor(*uuid*, *value*, *security_level=SecurityLevel.OPEN*)

Adds a descriptor to the characteristic which is a constant, read-only value that cannot be updated after this call. This is a simplified parameter set built on top of *add_descriptor()* for this common use-case.

Note: See note on *add_descriptor()* for limitations on descriptors that can be added through this method.

Parameters

- **uuid** (*Uuid*) – The UUID of the descriptor to add
- **value** (bytes) – The value to set the descriptor to
- **security_level** – The security level for the descriptor

Return type
GattsAttribute

Returns
The descriptor that was created and added to the characteristic

property max_length: int

Read Only

The max possible the value the characteristic can be set to

property notifiable: bool

Read Only

Gets if the characteristic is set up to asynchonously notify clients via notifications or indications

property value: bytes

Read Only

Gets the current value of the characteristic. Value is updated using *set_value()*

property client_subscribed: bool

Read Only

Gets if the client is currently subscribed (notify or indicate) to this characteristic

property attributes: Iterable[*GattsAttribute*]

Read Only

Gets all of the attributes and descriptors associated with this characteristic

property user_description: Optional[*GattsAttribute*]

Read Only

Gets the User Description attribute for the characteristic if set in the properties. If the user description was not configured for the characteristic, returns `None`

property sccd: Optional[*GattsAttribute*]

Read Only

Gets the Server Characteristic Configuration Descriptor (SCCD) attribute if set in the properties. If the SCCD was not configured for the characteristic, returns `None`

property presentation_format: Optional[*PresentationFormat*]

Read Only

Gets the presentation format that was set for the characteristic. If the presentation format was not configured for the characteristic, returns `None`

property string_encoding: str

The default method for encoding strings into bytes when a string is provided as a value

Getter
Gets the string encoding in use

Setter
Sets the string encoding to use

property on_write: *Event*[*GattsCharacteristic*, *WriteEventArgs*]

Event generated whenever a client writes to this characteristic.

Returns
an Event which can have handlers registered to and deregistered from

property on_read: *Event*[*GattsCharacteristic*, None]

Event generated whenever a client requests to read from this characteristic. At this point, the application may choose to update the value of the characteristic to a new value using set_value.

A good example of this is a "system time" characteristic which reports the applications system time in seconds. Instead of updating this characteristic every second, it can be "lazily" updated only when read from.

NOTE: if there are multiple handlers subscribed to this and each set the value differently, it may cause undefined behavior.

Returns
an Event which can have handlers registered to and deregistered from

property on_subscription_change: *Event*[*GattsCharacteristic*, *SubscriptionStateChangeEventArgs*]

Event that is generated whenever a client changes its subscription state of the characteristic (notify, indicate, none).

Returns
an Event which can have handlers registered to and deregistered from

property on_notify_complete: *Event*[*GattsCharacteristic*, *NotificationCompleteEventArgs*]

Event that is generated when a notification or indication sent to the client successfully

Returns
an event which can have handlers registered to and deregistered from

class blatann.gatt.gatts.**GattsService**(*ble_device*, *peer*, *uuid*, *service_type*, *notification_manager*, *start_handle=0*, *end_handle=0*)

Bases: *Service*

Represents a registered GATT service that lives locally on the device.

This class is usually not instantiated directly and is instead created through *GattsDatabase.add_service()*.

property characteristics: List[*GattsCharacteristic*]

Read Only

Gets the list of characteristics in this service.

Characteristics are added through *add_characteristic()*

add_characteristic(*uuid*, *properties*, *initial_value=b''*, *prefer_indications=True*, *string_encoding='utf8'*)

Adds a new characteristic to the service

Parameters

- **uuid** (*Uuid*) – The UUID of the characteristic to add
- **properties** (*GattsCharacteristicProperties*) – The characteristic's properties
- **initial_value** (*str or list or bytearray*) – The initial value of the characteristic. May be a string, bytearray, or list of ints
- **prefer_indications** – Flag for choosing indication/notification if a characteristic has both indications and notifications available
- **string_encoding** – The encoding method to use when a string value is provided (utf8, ascii, etc.)

Returns
The characteristic just added to the service

Return type
GattsCharacteristic

class blatann.gatt.gatts.**GattsDatabase**(*ble_device*, *peer*, *notification_hardware_queue_size=1*)

Bases: *GattDatabase*

Represents the entire GATT server that lives locally on the device which clients read from and write to

property **services**: List[*GattsService*]

Read Only

The list of services registered in the database

iter_services()

Iterates through all of the registered services in the database

Return type
Iterable[*GattsService*]

Returns
Generator of the database's services

add_service(*uuid*, *service_type=ServiceType.PRIMARY*)

Adds a service to the local database

Parameters

- **uuid** (*Uuid*) – The UUID for the service
- **service_type** – The type of service (primary or secondary)

Return type
GattsService

Returns
The added and newly created service

clear_pending_notifications()

Clears all pending notifications that are queued to be sent to the client

blatann.gatt.gatts_attribute module

class blatann.gatt.gatts_attribute.**GattsAttributeProperties**(*read=True*, *write=False*, *security_level=SecurityLevel.OPEN*, *max_length=20*, *variable_length=True*, *read_auth=False*, *write_auth=False*)

Bases: object

class blatann.gatt.gatts_attribute.**GattsAttribute**(*ble_device*, *peer*, *parent*, *uuid*, *handle*, *properties*, *initial_value=b''*, *string_encoding='utf8'*)

Bases: *Attribute*

Represents the server-side interface of a single attribute which lives inside a Characteristic.

property parent: *GattsCharacteristic*

Read Only

Gets the parent characteristic which owns this attribute

property max_length: int

Read Only

The max possible length data the attribute can be set to

property read_in_process: bool

Read Only

Gets whether or not the client is in the process of reading out this attribute

set_value(*value*)

Sets the value of the attribute.

Parameters

value – The value to set to. Must be an iterable type such as a str, bytes, or list of uint8 values, or a BleDataStream object. Length must be less than the attribute's max length. If a str is given, it will be encoded using the string_encoding property.

Raises

InvalidOperationException if value length is too long

get_value()

Fetches the attribute's value from hardware and updates the local copy. This isn't often necessary and should instead use the value property to avoid unnecessary reads from the hardware.

Return type

bytes

property on_write: *Event*[*GattsAttribute*, *WriteEventArgs*]

Event generated whenever a client writes to this attribute.

Returns

an Event which can have handlers registered to and deregistered from

property on_read: *Event*[*GattsAttribute*, None]

Event generated whenever a client requests to read from this attribute. At this point, the application may choose to update the value of the attribute to a new value using set_value.

Note: This will only be triggered if the attribute was configured with the read_auth property

---

A good example of using this is a "system time" characteristic which reports the application's current system time in seconds. Instead of updating this characteristic every second, it can be "lazily" updated only when read.

NOTE: if there are multiple handlers subscribed to this and each set the value differently, it may cause undefined behavior.

Returns

an Event which can have handlers registered to and deregistered from

blatann.gatt.managers module

class blatann.gatt.managers.**GattcOperationManager**(*ble_device*, *peer*, *reader*, *writer*, *write_no_response_queue_size=1*)

Bases: object

read(*handle*, *callback*)

write(*handle*, *value*, *callback*, *with_response=True*)

clear_all()

class blatann.gatt.managers.**GattsOperationManager**(*ble_device*, *peer*, *notification_queue_size=1*)

Bases: object

notify(*characteristic*, *handle*, *event_on_complete*, *data=None*)

clear_all()

blatann.gatt.reader module

class blatann.gatt.reader.**GattcReadCompleteEventArgs**(*handle*, *status*, *data*)

Bases: *EventArgs*

class blatann.gatt.reader.**GattcReader**(*ble_device*, *peer*)

Bases: object

Class which implements the state machine for completely reading a peripheral's attribute

property on_read_complete

Event that is emitted when a read completes on an attribute handle.

Handler args: (int attribute_handle, gatt.GattStatusCode, bytes data_read)

Returns
an Event which can have handlers registered to and deregistered from

Return type
Event

read(*handle*)

Reads the attribute value from the handle provided. Can only read from a single attribute at a time. If a read is in progress, raises an InvalidStateException

Parameters
handle – the attribute handle to read

Returns
A waitable that will fire when the read finishes. See on_read_complete for the values returned from the waitable

Return type
EventWaitable

blatann.gatt.service_discovery module

class blatann.gatt.service_discovery.**DatabaseDiscoverer**(*ble_device*, *peer*)

Bases: object

property on_discovery_complete

Return type
Event[*blatann.peer.Peripheral*, *DatabaseDiscoveryCompleteEventArgs*]

start()

blatann.gatt.writer module

class blatann.gatt.writer.**GattcWriteCompleteEventArgs**(*handle*, *status*, *data*)

Bases: *EventArgs*

class blatann.gatt.writer.**GattcWriter**(*ble_device*, *peer*)

Bases: object

Class which implements the state machine for writing a value to a peripheral's attribute

property on_write_complete

Event that is emitted when a write completes on an attribute handler

Handler args: (int attribute_handle, gatt.GattStatusCode, bytearray data_written)

Returns
an Event which can have handlers registered to and deregistered from

Return type
Event

write(*handle*, *data*)

Writes data to the attribute at the handle provided. Can only write to a single attribute at a time. If a write is in progress, raises an InvalidStateException

Parameters

- **handle** – The attribute handle to write
- **data** – The data to write

Returns
A Waitable that will fire when the write finishes. see on_write_complete for the values returned from the waitable

Return type
EventWaitable

blatann.nrf package

Subpackages

blatann.nrf.nrf_events package

blatann.nrf.nrf_events.**event_decode**(*ble_event*)

Submodules

blatann.nrf.nrf_events.gap_events module

class blatann.nrf.nrf_events.gap_events.**GapEvt**(*conn_handle*)

Bases: *BLEEvent*

class blatann.nrf.nrf_events.gap_events.**GapEvtRssiChanged**(*conn_handle*, *rssi*)

Bases: *GapEvt*

evt_id = 28

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtAdvReport**(*conn_handle*, *peer_addr*, *rssi*, *adv_type*, *adv_data*)

Bases: *GapEvt*

evt_id = 29

get_device_name()

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtTimeout**(*conn_handle*, *src*)

Bases: *GapEvt*

evt_id = 27

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtConnParamUpdateRequest**(*conn_handle*, *conn_params*)

Bases: *GapEvt*

evt_id = 31

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtConnParamUpdate**(*conn_handle*, *conn_params*)

Bases: *GapEvt*

evt_id = 18

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtConnected**(*conn_handle*, *peer_addr*, *role*, *conn_params*)

Bases: *GapEvt*

evt_id = 16

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtDisconnected**(*conn_handle*, *reason*)

Bases: *GapEvt*

evt_id = 17

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtDataLengthUpdate**(*conn_handle*, *max_tx_octets*, *max_rx_octets*, *max_tx_time_us*, *max_rx_time_us*)

Bases: *GapEvt*

evt_id = 36

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtDataLengthUpdateRequest**(*conn_handle*, *max_tx_octets*, *max_rx_octets*, *max_tx_time_us*, *max_rx_time_us*)

Bases: *GapEvt*

evt_id = 35

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtPhyUpdate**(*conn_handle*, *status*, *tx_phy*, *rx_phy*)

Bases: *GapEvt*

evt_id = 34

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gap_events.**GapEvtPhyUpdateRequest**(*conn_handle*, *tx_phy*, *rx_phy*)

Bases: *GapEvt*

evt_id = 33

classmethod from_c(*event*)

blatann.nrf.nrf_events.gatt_events module

class blatann.nrf.nrf_events.gatt_events.**GattEvt**(*conn_handle*)

Bases: *BLEEvent*

class blatann.nrf.nrf_events.gatt_events.**GattcEvt**(*conn_handle*)

Bases: *GattEvt*

class blatann.nrf.nrf_events.gatt_events.**GattsEvt**(*conn_handle*)

Bases: *GattEvt*

class blatann.nrf.nrf_events.gatt_events.**GattcEvtReadResponse**(*conn_handle*, *status*, *error_handle*, *attr_handle*, *offset*, *data*)

Bases: *GattcEvt*

evt_id = 54

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtHvx**(*conn_handle*, *status*, *error_handle*, *attr_handle*, *hvx_type*, *data*)

Bases: *GattcEvt*

evt_id = 57

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtWriteCmdTxComplete**(*conn_handle*, *count*)

Bases: *GattcEvt*

evt_id = 60

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtWriteResponse**(*conn_handle*, *status*, *error_handle*, *attr_handle*, *write_op*, *offset*, *data*)

Bases: *GattcEvt*

evt_id = 56

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtPrimaryServiceDiscoveryResponse**(*conn_handle*, *status*, *services*)

Bases: *GattcEvt*

evt_id = 48

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtCharacteristicDiscoveryResponse**(*conn_handle*, *status*, *characteristics*)

Bases: *GattcEvt*

evt_id = 50

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtDescriptorDiscoveryResponse**(*conn_handle*, *status*, *descriptions*)

Bases: *GattcEvt*

evt_id = 51

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtAttrInfoDiscoveryResponse**(*conn_handle*, *status*, *attr_info16=None*, *attr_info128=None*)

Bases: *GattcEvt*

evt_id = 52

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtMtuExchangeResponse**(*conn_handle*, *server_mtu*)

Bases: *GattcEvt*

evt_id = 58

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattcEvtTimeout**(*conn_handle*, *source*)

Bases: *GattcEvt*

evt_id = 59

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtSysAttrMissing**(*conn_handle*, *hint*)

Bases: *GattsEvt*

evt_id = 82

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtWrite**(*conn_handle*, *attr_handle*, *uuid*, *write_operand*, *auth_required*, *offset*, *data*)

Bases: *GattsEvt*

evt_id = 80

classmethod from_c(*event*)

classmethod from_auth_request(*conn_handle*, *write_event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtRead**(*conn_handle*, *attr_handle*, *uuid*, *offset*)

Bases: *GattsEvt*

classmethod from_auth_request(*conn_handle*, *read_event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtReadWriteAuthorizeRequest**(*conn_handle*, *read=None*, *write=None*)

Bases: *GattsEvt*

evt_id = 81

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtHandleValueConfirm**(*conn_handle*, *attr_handle*)

Bases: *GattsEvt*

evt_id = 83

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtNotificationTxComplete**(*conn_handle*, *tx_count*)

Bases: *GattsEvt*

evt_id = 87

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtExchangeMtuRequest**(*conn_handle*, *client_mtu*)

Bases: *GattsEvt*

evt_id = 85

classmethod from_c(*event*)

class blatann.nrf.nrf_events.gatt_events.**GattsEvtTimeout**(*conn_handle*, *source*)

Bases: *GattsEvt*

evt_id = 86

classmethod from_c(*event*)

blatann.nrf.nrf_events.generic_events module

class blatann.nrf.nrf_events.generic_events.**BLEEvent**(*conn_handle*)

Bases: object

evt_id = None

class blatann.nrf.nrf_events.generic_events.**EvtUserMemoryRequest**(*conn_handle*, *request_type*)

Bases: *BLEEvent*

evt_id = 1

classmethod from_c(*event*)

blatann.nrf.nrf_events.smp_events module

class blatann.nrf.nrf_events.smp_events.**GapEvtSec**(*conn_handle*)

Bases: *GapEvt*

class blatann.nrf.nrf_events.smp_events.**GapEvtConnSecUpdate**(*conn_handle*, *sec_mode*, *sec_level*, *encr_key_size*)

Bases: *GapEvtSec*

evt_id = 26

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtSecInfoRequest**(*conn_handle*, *peer_addr*, *master_id*, *enc_info*, *id_info*, *sign_info*)

Bases: *GapEvtSec*

evt_id = 20

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtSecRequest**(*conn_handle*, *bond*, *mitm*, *lesc*, *keypress*)

Bases: *GapEvtSec*

evt_id = 30

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtSecParamsRequest**(*conn_handle*, *sec_params*)

Bases: *GapEvtSec*

evt_id = 19

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtAuthKeyRequest**(*conn_handle*, *key_type*)

Bases: *GapEvtSec*

evt_id = 23

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtAuthStatus**(*conn_handle*, *auth_status*, *error_src*, *bonded*, *sm1_levels*, *sm2_levels*, *kdist_own*, *kdist_peer*)

Bases: *GapEvtSec*

evt_id = 25

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtPasskeyDisplay**(*conn_handle*, *passkey*, *match_request*)

Bases: *GapEvtSec*

evt_id = 21

classmethod from_c(*event*)

class blatann.nrf.nrf_events.smp_events.**GapEvtLescDhKeyRequest**(*conn_handle*, *remote_public_key*, *oob_required*)

Bases: *GapEvtSec*

evt_id = 24

classmethod from_c(*event*)

blatann.nrf.nrf_types package

Submodules

blatann.nrf.nrf_types.config module

class blatann.nrf.nrf_types.config.**BleOptionFlag**(*value*)

Bases: IntEnum

An enumeration.

pa_lna = 1

conn_event_extension = 2

gap_channel_map = 32

gap_local_conn_latency = 33

gap_passkey = 34

gap_scan_req_report = 35

gap_compat_mode_1 = 36

gap_auth_payload_timeout = 37

gap_slave_latency_disable = 38

class blatann.nrf.nrf_types.config.**BleOption**

Bases: object

option_flag = None

path = ''

to_c()

class blatann.nrf.nrf_types.config.**BleEnableOpt**(*enabled=False*)

Bases: *BleOption*

to_c()

class blatann.nrf.nrf_types.config.**BleOptConnEventExtenion**(*enabled=False*)

Bases: *BleEnableOpt*

option_flag = 2

path = 'common_opt.conn_evt_ext'

class blatann.nrf.nrf_types.config.**BlePaLnaConfig**(*enabled=False*, *active_high=True*, *pin=0*)

Bases: object

to_c()

class blatann.nrf.nrf_types.config.**BleOptPaLna**(*pa_config=None*, *lna_cfg=None*, *ppi_channel_set=0*, *ppi_channel_clear=0*, *gpiote_channel=0*)

Bases: *BleOption*

option_flag = 1

path = 'common_opt.pa_lna'

to_c()

class blatann.nrf.nrf_types.config.**BleOptGapChannelMap**(*enabled_channels=None*, *conn_handle=0*)

Bases: *BleOption*

option_flag = 32

path = 'gap_opt.ch_map'

to_c()

class blatann.nrf.nrf_types.config.**BleOptGapLocalConnLatency**(*conn_handle=0*, *requested_latency=0*)

Bases: *BleOption*

option_flag = 33

path = 'gap_opt.local_conn_latency'

to_c()

class blatann.nrf.nrf_types.config.**BleOptGapPasskey**(*passkey='000000'*)

Bases: *BleOption*

option_flag = 34

path = 'gap_opt.passkey'

to_c()

class blatann.nrf.nrf_types.config.**BleOptGapScanRequestReport**(*enabled=False*)

Bases: *BleEnableOpt*

option_flag = 35

path = 'gap_opt.scan_req_report'

class blatann.nrf.nrf_types.config.**BleOptGapCompatMode1**(*enabled=False*)

Bases: *BleEnableOpt*

option_flag = 36

path = 'gap_opt.compat_mode_q'

class blatann.nrf.nrf_types.config.**BleOptGapAuthPayloadTimeout**(*conn_handle*, *timeout_ms=48000*)

Bases: *BleOption*

option_flag = 37

path = 'gap_opt.auth_payload_timeout'

to_c()

class blatann.nrf.nrf_types.config.BleOptGapSlaveLatencyDisable(*conn_handle*, *disabled=False*)

Bases: *BleOption*

option_flag = 38

path = 'gap_opt.slave_latency_disable'

to_c()

class blatann.nrf.nrf_types.config.BleEnableConfig(*vs_uuid_count=10*, *periph_role_count=1*, *central_role_count=3*, *central_sec_count=3*, *service_changed_char=1*, *attr_table_size=1408*)

Bases: object

get_vs_uuid_cfg()

get_role_count_cfg()

get_device_name_cfg()

get_service_changed_cfg()

get_attr_tab_size_cfg()

get_configs()

class blatann.nrf.nrf_types.config.BleConnConfig(*conn_tag=1*, *conn_count=1*, *event_length=3*, *write_cmd_tx_queue_size=1*, *hvn_tx_queue_size=1*, *max_att_mtu=23*)

Bases: object

DEFAULT_CONN_TAG = 1

get_gap_config()

get_gatt_config()

get_gattc_config()

get_gatts_config()

get_configs()

blatann.nrf.nrf_types.enums module

class blatann.nrf.nrf_types.enums.BLEHci(*value*)

Bases: Enum

An enumeration.

success = 0

unknown_btle_command = 1

unknown_connection_identifier = 2

authentication_failure = 5

pin_or_key_missing = 6

memory_capacity_exceeded = 7

connection_timeout = 8

command_disallowed = 12

invalid_btle_command_parameters = 18

remote_user_terminated_connection = 19

remote_dev_termination_due_to_low_resources = 20

remote_dev_termination_due_to_power_off = 21

local_host_terminated_connection = 22

unsupported_remote_feature = 26

invalid_lmp_parameters = 30

unspecified_error = 31

lmp_response_timeout = 34

lmp_transaction_collision = 35

lmp_pdu_not_allowed = 36

instant_passed = 40

pairintg_with_unit_key_unsupported = 41

differen_transaction_collision = 42

controller_busy = 58

conn_interval_unacceptable = 59

parameter_out_of_mandatory_range = 48

directed_advertiser_timeout = 60

conn_terminated_due_to_mic_failure = 61

conn_failed_to_be_established = 62

class blatann.nrf.nrf_types.enums.**NrfError**(*value*)

Bases: Enum

An enumeration.

success = 0

```
svc_handler_missing = 1

softdevice_not_enabled = 2

internal = 3

no_mem = 4

not_found = 5

not_supported = 6

invalid_param = 7

invalid_state = 8

invalid_length = 9

invalid_flags = 10

invalid_data = 11

data_size = 12

timeout = 13

null = 14

forbidden = 15

invalid_addr = 16

busy = 17

conn_count = 18

resources = 19

ble_not_enabled = 12289

ble_invalid_conn_handle = 12290

ble_invalid_attr_handle = 12291

ble_invalid_adv_handle = 12292

ble_invalid_role = 12293

ble_blocked_by_other_links = 12294

ble_gap_uuid_list_mismatch = 12800

ble_gap_discoverable_with_whitelist = 12801

ble_gap_invalid_ble_addr = 12802

ble_gap_whitelist_in_use = 12803

ble_gap_device_identities_in_use = 12804

ble_gap_device_identities_duplicate = 12805
```

ble_gattc_proc_not_permitted = 13056

ble_gatts_invalid_attr_type = 13312

ble_gatts_sys_attr_missing = 13313

rpc_encode = 32769

rpc_decode = 32770

rpc_send = 32771

rpc_invalid_argument = 32772

rpc_no_response = 32773

rpc_invalid_state = 32774

rpc_serialization_transport = 32788

rpc_serialization_transport_invalid_state = 32789

rpc_serialization_transport_no_response = 32790

rpc_serialization_transport_already_open = 32791

rpc_serialization_transport_already_closed = 32792

rpc_h5_transport = 32808

rpc_h5_transport_state = 32809

rpc_h5_transport_no_response = 32810

rpc_h5_transport_slip_payload_size = 32811

rpc_h5_transport_slip_calculated_payload_size = 32812

rpc_h5_transport_slip_decoding = 32813

rpc_h5_transport_header_checksum = 32814

rpc_h5_transport_packet_checksum = 32815

rpc_h5_transport_already_open = 32816

rpc_h5_transport_already_closed = 32817

rpc_h5_transport_internal_error = 32818

rpc_serial_port = 32828

rpc_serial_port_state = 32829

rpc_serial_port_already_open = 32830

rpc_serial_port_already_closed = 32831

rpc_serial_port_internal_error = 32832

class blatann.nrf.nrf_types.enums.**BLEGapAdvType**(*value*)

Bases: IntEnum

An enumeration.

connectable_undirected = 0

connectable_directed = 1

scanable_undirected = 2

non_connectable_undirected = 3

scan_response = 4

class blatann.nrf.nrf_types.enums.**BLEGapRoles**(*value*)

Bases: IntEnum

An enumeration.

invalid = 0

periph = 1

central = 2

class blatann.nrf.nrf_types.enums.**BLEGapTimeoutSrc**(*value*)

Bases: IntEnum

An enumeration.

advertising = 0

scan = 1

conn = 2

class blatann.nrf.nrf_types.enums.**BLEGapPhy**(*value*)

Bases: IntFlag

An enumeration.

auto = 0

one_mbps = 1

two_mbps = 2

coded = 4

class blatann.nrf.nrf_types.enums.**BLEGapIoCaps**(*value*)

Bases: IntEnum

An enumeration.

DISPLAY_ONLY = 0

DISPLAY_YESNO = 1

KEYBOARD_ONLY = 2

NONE = 3

KEYBOARD_DISPLAY = 4

class blatann.nrf.nrf_types.enums.BLEGapAuthKeyType(*value*)

Bases: IntEnum

An enumeration.

NONE = 0

OOB = 2

PASSKEY = 1

class blatann.nrf.nrf_types.enums.BLEGapSecStatus(*value*)

Bases: IntEnum

An enumeration.

success = 0

timeout = 1

pdu_invalid = 2

passkey_entry_failed = 129

oob_not_available = 130

auth_req = 131

confirm_value = 132

pairing_not_supp = 133

enc_key_size = 134

smp_cmd_unsupported = 135

unspecified = 136

repeated_attempts = 137

invalid_params = 138

dhkey_failure = 139

num_comp_failure = 140

br_edr_in_prog = 141

x_trans_key_disallowed = 142

class blatann.nrf.nrf_types.enums.BLEGattWriteOperation(*value*)

Bases: Enum

An enumeration.

invalid = 0

write_req = 1

write_cmd = 2

signed_write_cmd = 3

prepare_write_req = 4

execute_write_req = 5

class blatann.nrf.nrf_types.enums.BLEGattHVXType(*value*)

Bases: Enum

An enumeration.

invalid = 0

notification = 1

indication = 2

class blatann.nrf.nrf_types.enums.BLEGattStatusCode(*value*)

Bases: Enum

An enumeration.

success = 0

unknown = 1

invalid = 256

invalid_handle = 257

read_not_permitted = 258

write_not_permitted = 259

invalid_pdu = 260

insuf_authentication = 261

request_not_supported = 262

invalid_offset = 263

insuf_authorization = 264

prepare_queue_full = 265

attribute_not_found = 266

attribute_not_long = 267

insuf_enc_key_size = 268

invalid_att_val_length = 269

unlikely_error = 270

insuf_encryption = 271

unsupported_group_type = 272

insuf_resources = 273

rfu_range1_begin = 274

rfu_range1_end = 383

app_begin = 384

app_end = 415

rfu_range2_begin = 416

rfu_range2_end = 479

rfu_range3_begin = 480

rfu_range3_end = 508

cps_cccd_config_error = 509

cps_proc_alr_in_prog = 510

cps_out_of_range = 511

class blatann.nrf.nrf_types.enums.BLEGattExecWriteFlag(*value*)

Bases: Enum

An enumeration.

prepared_cancel = 0

prepared_write = 1

unused = 0

class blatann.nrf.nrf_types.enums.BLEGattsWriteOperation(*value*)

Bases: Enum

An enumeration.

invalid = 0

write_req = 1

write_cmd = 2

sign_write_cmd = 3

prep_write_req = 4

exec_write_req_cancel = 5

exec_write_req_now = 6

blatann.nrf.nrf_types.gap module

class blatann.nrf.nrf_types.gap.**TimeRange**(*name*, *val_min*, *val_max*, *unit_ms_conversion*, *divisor=1.0*, *units='ms'*)

Bases: object

property name: str

property min: float

property max: float

property units: str

is_in_range(*value*)

validate(*value*)

class blatann.nrf.nrf_types.gap.**BLEGapAdvParams**(*interval_ms*, *timeout_s*, *advertising_type=BLEGapAdvType.connectable_undirected*, *channel_mask=None*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gap.**BLEGapScanParams**(*interval_ms*, *window_ms*, *timeout_s*, *active=True*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gap.**BLEGapConnParams**(*min_conn_interval_ms*, *max_conn_interval_ms*, *conn_sup_timeout_ms*, *slave_latency*)

Bases: object

validate()

classmethod from_c(*conn_params*)

to_c()

class blatann.nrf.nrf_types.gap.**BLEGapAddrTypes**(*value*)

Bases: IntEnum

An enumeration.

public = 0

random_static = 1

random_private_resolvable = 2

random_private_non_resolvable = 3

anonymous = 127

class blatann.nrf.nrf_types.gap.**BLEGapAddr**(*addr_type*, *addr*)

Bases: object

classmethod from_c(*addr*)

classmethod **from_string**(*addr_string*)

to_c()

get_addr_type_str()

get_addr_flag()

class blatann.nrf.nrf_types.gap.**BLEAdvData**(***kwargs*)

Bases: object

class **Types**(*value*)

Bases: Enum

An enumeration.

flags = 1

service_16bit_uuid_more_available = 2

service_16bit_uuid_complete = 3

service_32bit_uuid_more_available = 4

service_32bit_uuid_complete = 5

service_128bit_uuid_more_available = 6

service_128bit_uuid_complete = 7

short_local_name = 8

complete_local_name = 9

tx_power_level = 10

class_of_device = 13

simple_pairing_hash_c = 14

simple_pairing_randimizer_r = 15

security_manager_tk_value = 16

security_manager_oob_flags = 17

slave_connection_interval_range = 18

solicited_sevice_uuids_16bit = 20

solicited_sevice_uuids_128bit = 21

service_data = 22

public_target_address = 23

random_target_address = 24

appearance = 25

advertising_interval = 26

le_bluetooth_device_address = 27

le_role = 28

simple_pairng_hash_c256 = 29

simple_pairng_randomizer_r256 = 30

service_data_32bit_uuid = 32

service_data_128bit_uuid = 33

uri = 36

information_3d_data = 61

manufacturer_specific_data = 255

to_list()

to_c()

classmethod from_c(*adv_report_evt*)

class blatann.nrf.nrf_types.gap.**BLEGapDataLengthParams**(*max_tx_octets=0*, *max_rx_octets=0*, *max_tx_time_us=0*, *max_rx_time_us=0*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gap.**BLEGapPhys**(*tx_phys=BLEGapPhy.auto*, *rx_phys=BLEGapPhy.auto*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gap.**BLEGapPrivacyParams**(*enabled=False*, *resolvable_addr=False*, *addr_update_rate_s=900*)

Bases: object

DEFAULT_PRIVATE_ADDR_CYCLE_INTERVAL_S = 900

to_c()

classmethod from_c(*privacy*)

blatann.nrf.nrf_types.gatt module

blatann.nrf.nrf_types.gatt.**BLE_GATT_HANDLE_INVALID = 0**

GATT Classes

class blatann.nrf.nrf_types.gatt.**BleGattEnableParams**(*max_att_mtu=0*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattCharacteristicProperties**(*broadcast=False*, *read=False*, *write_wo_resp=False*, *write=False*, *notify=False*, *indicate=False*, *auth_signed_wr=False*)

Bases: object

classmethod from_c(*gattc_char_props*)

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattExtendedCharacteristicProperties**(*reliable_write=False*, *writable_aux=False*)

Bases: object

to_c()

classmethod from_c(*params*)

class blatann.nrf.nrf_types.gatt.**BLEGattService**(*uuid*, *start_handle*, *end_handle*)

Bases: object

srvc_uuid = 0x2800 (Standard.service_primary)

classmethod from_c(*gattc_service*)

char_add(*char*)

class blatann.nrf.nrf_types.gatt.**BLEGattCharacteristic**(*uuid*, *handle_decl*, *handle_value*, *data_decl=None*, *data_value=None*, *char_props=None*)

Bases: object

char_uuid = 0x2803 (Standard.characteristic)

discovered_handles()

missing_handles()

classmethod from_c(*gattc_char*)

class blatann.nrf.nrf_types.gatt.**BleGattHandle**(*handle=0*)

Bases: object

class blatann.nrf.nrf_types.gatt.**BLEGattcWriteParams**(*write_op*, *flags*, *handle*, *data*, *offset*)

Bases: object

classmethod from_c(*gattc_write_params*)

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattcDescriptor**(*uuid*, *handle*, *data=None*)

Bases: object

classmethod from_c(*gattc_desc*)

class blatann.nrf.nrf_types.gatt.**BLEGattcAttrInfo16**(*handle*, *uuid*)

Bases: object

classmethod from_c(*attr_info16*)

class blatann.nrf.nrf_types.gatt.**BLEGattcAttrInfo128**(*attr_handle*, *uuid*)

Bases: object

classmethod from_c(*attr_info128*)

class blatann.nrf.nrf_types.gatt.**BleGattsEnableParams**(*service_changed*, *attribute_table_size*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattsCharHandles**(*value_handle=0*, *user_desc_handle=0*, *cccd_handle=0*, *sccd_handle=0*)

Bases: object

to_c()

classmethod from_c(*handle_params*)

class blatann.nrf.nrf_types.gatt.**BLEGattsAttribute**(*uuid*, *attr_metadata*, *max_len*, *value=b''*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattsPresentationFormat**(*fmt*, *exponent*, *unit*, *namespace*, *description*)

Bases: object

to_c()

classmethod from_c(*params*)

class blatann.nrf.nrf_types.gatt.**BLEGattsAttrMetadata**(*read_permissions=<blatann.nrf.nrf_types.smp.BLEGapSecMod object>*, *write_permissions=<blatann.nrf.nrf_types.smp.BLEGapSecMo object>*, *variable_length=False*, *read_auth=False*, *write_auth=False*)

Bases: object

to_c()

classmethod from_c(*params*)

class blatann.nrf.nrf_types.gatt.**BLEGattsCharMetadata**(*char_props*, *user_description=''*, *user_description_max_size=0*, *user_desc_metadata=None*, *cccd_metadata=None*, *sccd_metadata=None*, *presentation_format=None*)

Bases: object

to_c()

classmethod from_c(*params*)

class blatann.nrf.nrf_types.gatt.**BLEGattsAuthorizeParams**(*gatt_status*, *update*, *offset=0*, *data=''*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattsRwAuthorizeReplyParams**(*read=None*, *write=None*)

Bases: object

to_c()

class blatann.nrf.nrf_types.gatt.**BLEGattsValue**(*value*, *offset=0*)

Bases: object

to_c()

classmethod from_c(*params*)

class blatann.nrf.nrf_types.gatt.**BLEGattsHvx**(*char_handle*, *hvx_type*, *data*, *offset=0*)

Bases: object

to_c()

blatann.nrf.nrf_types.generic module

class blatann.nrf.nrf_types.generic.**BLEUUIDBase**(*vs_uuid_base=None*, *uuid_type=None*)

Bases: object

BLE_UUID_TYPE_BLE = 1

classmethod from_c(*uuid*)

classmethod from_uuid128_array(*uuid128_array*)

to_c()

class blatann.nrf.nrf_types.generic.**BLEUUID**(*value*, *base=<blatann.nrf.nrf_types.generic.BLEUUIDBase object>*)

Bases: object

class Standard(*value*)

Bases: Enum

An enumeration.

unknown = 0

service_primary = 10240

service_secondary = 10241

characteristic = 10243

cccd = 10498

battery_level = 10777

heart_rate = 10807

get_value()

as_array()

classmethod from_c(*uuid*)

classmethod from_uuid128(*uuid128*)

to_c()

classmethod from_array(*uuid_array_lt*)

blatann.nrf.nrf_types.smp module

class blatann.nrf.nrf_types.smp.**BLEGapSecMode**(*sec_mode*, *level*)

Bases: object

to_c()

classmethod from_c(*params*)

class blatann.nrf.nrf_types.smp.**BLEGapSecModeType**

Bases: object

NO_ACCESS = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

OPEN = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

ENCRYPTION = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

MITM = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

LESC_MITM = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

SIGN_OR_ENCRYPT = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

SIGN_OR_ENCRYPT_MITM = <blatann.nrf.nrf_types.smp.BLEGapSecMode object>

class blatann.nrf.nrf_types.smp.**BLEGapSecLevels**(*lv1*, *lv2*, *lv3*, *lv4*)

Bases: object

classmethod from_c(*sec_level*)

to_c()

class blatann.nrf.nrf_types.smp.**BLEGapSecKeyDist**(*enc_key=False*, *id_key=False*, *sign_key=False*, *link_key=False*)

Bases: object

classmethod from_c(*kdist*)

to_c()

class blatann.nrf.nrf_types.smp.**BLEGapSecParams**(*bond*, *mitm*, *le_sec_pairing*, *keypress_noti*, *io_caps*, *oob*, *min_key_size*, *max_key_size*, *kdist_own*, *kdist_peer*)

Bases: object

classmethod from_c(*sec_params*)

to_c()

class blatann.nrf.nrf_types.smp.**BLEGapMasterId**(*ediv=0*, *rand=b''*)

Bases: object

RAND_LEN = 8

RAND_INVALID = b'\x00\x00\x00\x00\x00\x00\x00\x00'

to_c()

property is_valid: bool

classmethod from_c(*master_id*)

to_dict()

classmethod from_dict(*data*)

class blatann.nrf.nrf_types.smp.**BLEGapEncryptInfo**(*ltk=b''*, *lesc=False*, *auth=False*)

Bases: object

KEY_LENGTH = 16

to_c()

classmethod from_c(*info*)

to_dict()

classmethod from_dict(*data*)

class blatann.nrf.nrf_types.smp.**BLEGapEncryptKey**(*enc_info=None*, *master_id=None*)

Bases: object

to_c()

classmethod from_c(*key*)

to_dict()

classmethod from_dict(*data*)

class blatann.nrf.nrf_types.smp.**BLEGapIdKey**(*irk=b''*, *peer_addr=None*)

Bases: object

KEY_LENGTH = 16

to_c()

classmethod from_c(*id_key*)

to_dict()

classmethod from_dict(*data*)

class blatann.nrf.nrf_types.smp.**BLEGapPublicKey**(*key=b''*)

Bases: object

KEY_LENGTH = 64

to_c()

classmethod from_c(*key*)

class blatann.nrf.nrf_types.smp.**BLEGapDhKey**(*key=b''*)

Bases: object

KEY_LENGTH = 32

to_c()

classmethod from_c(*key*)

class blatann.nrf.nrf_types.smp.**BLEGapSignKey**(*key=b''*)

Bases: object

KEY_LENGTH = 16

to_c()

classmethod from_c(*key*)

to_dict()

classmethod from_dict(*data*)

class blatann.nrf.nrf_types.smp.**BLEGapSecKeys**(*enc_key=None*, *id_key=None*, *sign_key=None*, *public_key=None*)

Bases: object

to_c()

classmethod from_c(*keys*)

class blatann.nrf.nrf_types.smp.**BLEGapSecKeyset**(*own_keys=None*, *peer_keys=None*)

Bases: object

to_c()

reload()

classmethod from_c(*keyset*)

Submodules

blatann.nrf.nrf_dll_load module

blatann.nrf.nrf_driver module

blatann.nrf.nrf_driver.**NordicSemiErrorCheck**(*wrapped=None*, *expected=0*)

class blatann.nrf.nrf_driver.**NrfDriverObserver**

Bases: object

on_driver_event(*nrf_driver*, *event*)

class blatann.nrf.nrf_driver.**NrfDriver**(*serial_port*, *baud_rate=None*, *log_driver_comms=False*)

Bases: object

default_baud_rate = 1000000

ATT_MTU_DEFAULT = 23

property serial_port

open()

property is_open

close()

event_subscribe(*handler*, **event_types*)

event_unsubscribe(*handler*, **event_types*)

event_unsubscribe_all(*handler*)

observer_register(*observer*)

observer_unregister(*observer*)

ble_enable_params_setup()

adv_params_setup()

scan_params_setup()

conn_params_setup()

security_params_setup()

ble_conn_configure(*conn_params*)

ble_enable(*ble_enable_params=None*)

ble_opt_set(*ble_opt*)

ble_user_mem_reply(*conn_handle*)

ble_vs_uuid_add(*uuid_base*)

ble_gap_addr_get()

ble_gap_addr_set(*address*)

ble_gap_device_name_set(*name*)

ble_gap_appearance_set(*value*)

ble_gap_ppcp_set(*conn_params*)

ble_gap_tx_power_set(*tx_power*)

ble_gap_privacy_set(*privacy*)

ble_gap_adv_start(*adv_params=None*, *conn_cfg_tag=0*)

ble_gap_conn_param_update(*conn_handle*, *conn_params*)

ble_gap_adv_stop()

ble_gap_scan_start(*scan_params=None*)

ble_gap_scan_stop()

ble_gap_rssi_start(*conn_handle*, *threshold_dbm*, *skip_count*)

ble_gap_rssi_stop(*conn_handle*)

ble_gap_rssi_get(*conn_handle*)

ble_gap_connect(*address*, *scan_params=None*, *conn_params=None*, *conn_cfg_tag=0*)

ble_gap_disconnect(*conn_handle*, *hci_status_code=BLEHci.remote_user_terminated_connection*)

ble_gap_adv_data_set(*adv_data={}*, *scan_data={}*)

ble_gap_data_length_update(*conn_handle*, *params=None*)

ble_gap_phy_update(*conn_handle*, *tx_phy=BLEGapPhy.auto*, *rx_phy=BLEGapPhy.auto*)

ble_gap_authenticate(*conn_handle*, *sec_params*)

ble_gap_sec_params_reply(*conn_handle*, *sec_status*, *sec_params*, *sec_keyset*)

ble_gap_auth_key_reply(*conn_handle*, *key_type*, *key*)

ble_gap_sec_info_reply(*conn_handle*, *enc_info=None*, *irk=None*, *sign_info=None*)

ble_gap_encrypt(*conn_handle*, *master_id*, *enc_info*)

ble_gap_lesc_dhkey_reply(*conn_handle*, *dh_key*)

ble_gatts_service_add(*service_type*, *uuid*, *service_handle*)

ble_gatts_characteristic_add(*service_handle*, *char_md*, *attr_char_value*, *char_handle*)

ble_gatts_descriptor_add(*char_handle*, *attr*)

ble_gatts_rw_authorize_reply(*conn_handle*, *authorize_reply_params*)

ble_gatts_value_get(*conn_handle*, *attribute_handle*, *gatts_value*, *max_bytes_read=512*)

ble_gatts_value_set(*conn_handle*, *attribute_handle*, *gatts_value*)

ble_gatts_hvx(*conn_handle*, *hvx_params*)

ble_gatts_service_changed(*conn_handle*, *start_handle*, *end_handle*)

ble_gatts_exchange_mtu_reply(*conn_handle*, *server_mtu*)

ble_gatts_sys_attr_set(*conn_handle*, *sys_attr_data*, *flags=0*)

ble_gattc_write(*conn_handle*, *write_params*)

ble_gattc_prim_srvc_disc(*conn_handle*, *srvc_uuid*, *start_handle*)

ble_gattc_char_disc(*conn_handle*, *start_handle*, *end_handle*)

ble_gattc_desc_disc(*conn_handle*, *start_handle*, *end_handle*)

ble_gattc_attr_info_disc(*conn_handle*, *start_handle*, *end_handle*)

ble_gattc_read(*conn_handle*, *read_handle*, *offset=0*)

ble_gattc_exchange_mtu_req(*conn_handle*, *att_mtu_size*)

ble_gattc_hv_confirm(*conn_handle*, *attr_handle*)

ble_evt_handler(*adapter*, *ble_event*)

blatann.nrf.nrf_driver_types module

blatann.nrf.nrf_driver_types.**msec_to_units**(*time_ms*, *resolution*)
Convert milliseconds to BLE specific time units.

blatann.nrf.nrf_driver_types.**units_to_msec**(*units*, *resolution*)
Convert BLE specific units to milliseconds.

blatann.nrf.nrf_driver_types.**char_array_to_list**(*array_pointer*, *length*)
Convert char_array to python list.

blatann.nrf.nrf_driver_types.**uint8_array_to_list**(*array_pointer*, *length*)
Convert uint8_array to python list.

blatann.nrf.nrf_driver_types.**uint16_array_to_list**(*array_pointer*, *length*)
Convert uint16_array to python list.

blatann.nrf.nrf_driver_types.**service_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_service_array to python list.

blatann.nrf.nrf_driver_types.**include_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_include_array to python list.

blatann.nrf.nrf_driver_types.**ble_gattc_char_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_char_array to python list.

blatann.nrf.nrf_driver_types.**desc_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_desc_array to python list.

blatann.nrf.nrf_driver_types.**ble_gattc_attr_info16_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_attr_info16_array to python list

blatann.nrf.nrf_driver_types.**ble_gattc_attr_info128_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_attr_info128_array to python list

blatann.nrf.nrf_driver_types.**handle_value_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_handle_value_array to python list.

blatann.nrf.nrf_driver_types.**attr_info_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_attr_info_array to python list.

blatann.nrf.nrf_driver_types.**attr_info16_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_attr_info16_array to python list.

blatann.nrf.nrf_driver_types.**attr_info128_array_to_list**(*array_pointer*, *length*)
Convert ble_gattc_attr_info128_array to python list.

blatann.nrf.nrf_driver_types.**serial_port_desc_array_to_list**(*array_pointer*, *length*)
Convert sd_rpc_serial_port_desc_array to python list.

blatann.nrf.nrf_driver_types.**list_to_char_array**(*data_list*)
Convert python list to char_array.

blatann.nrf.nrf_driver_types.**list_to_uint8_array**(*data_list*)
Convert python list to uint8_array.

blatann.nrf.nrf_driver_types.**list_to_uint16_array**(*data_list*)
Convert python list to uint16_array.

blatann.nrf.nrf_driver_types.**list_to_service_array**(*data_list*)
Convert python list to ble_gattc_service_array.

blatann.nrf.nrf_driver_types.**list_to_include_array**(*data_list*)
Convert python list to ble_gattc_include_array.

blatann.nrf.nrf_driver_types.**list_to_ble_gattc_char_array**(*data_list*)
Convert python list to ble_gattc_char_array.

blatann.nrf.nrf_driver_types.**list_to_desc_array**(*data_list*)
Convert python list to ble_gattc_desc_array.

blatann.nrf.nrf_driver_types.**list_to_handle_value_array**(*data_list*)
Convert python list to ble_gattc_handle_value_array.

blatann.nrf.nrf_driver_types.**list_to_serial_port_desc_array**(*data_list*)
Convert python list to sd_rpc_serial_port_desc_array.

blatann.services package

Subpackages

blatann.services.battery package

blatann.services.battery.**add_battery_service**(*gatts_database*, *enable_notifications=False*, *security_level=SecurityLevel.OPEN*)

Adds a battery service to the given GATT Server database

Parameters

- **gatts_database** (blatann.gatt.gatts.GattsDatabase) – The database to add the service to
- **enable_notifications** – Whether or not the Battery Level characteristic allows notifications
- **security_level** – The security level to use for the service

Returns
The Battery service

Return type
_BatteryServer

blatann.services.battery.**find_battery_service**(*gattc_database*)

Finds a battery service in the given GATT client database

Parameters
gattc_database (blatann.gatt.gattc.GattcDatabase) – the GATT client database to search

Returns
The Battery service if found, None if not found

Return type
_BatteryClient

Submodules

blatann.services.battery.constants module

blatann.services.battery.data_types module

class blatann.services.battery.data_types.**BatteryLevel**

Bases: *Uint8*

blatann.services.battery.service module

class blatann.services.battery.service.**BatteryServer**(*service*, *enable_notifications=False*, *security_level=SecurityLevel.OPEN*)

Bases: object

set_battery_level(*battery_percent*, *notify_client=True*)

Sets the new battery level in the service

Parameters

- **battery_percent** (int) – The new battery percent
- **notify_client** – Whether or not to notify the connected client with the updated value

classmethod add_to_database(*gatts_database*, *enable_notifications=False*, *security_level=SecurityLevel.OPEN*)

class blatann.services.battery.service.**BatteryClient**(*gattc_service*)

Bases: object

read()

Reads the Battery level characteristic.

Return type
EventWaitable[*BatteryClient*, *DecodedReadCompleteEventArgs*[int]]

Returns
A waitable for when the read completes, which waits for the on_battery_level_update_event to be emitted

property on_battery_level_updated: *Event*[*BatteryClient*, *DecodedReadCompleteEventArgs*[int]]

Event that is generated whenever the battery level on the peripheral is updated, whether it is by notification or from reading the characteristic itself.

The DecodedReadCompleteEventArgs value given is the integer battery percent received. If the read failed or failed to decode, the value will be equal to the raw bytes received.

property can_enable_notifications: bool

Checks if the battery level characteristic allows notifications to be subscribed to

Returns
True if notifications can be enabled, False if not

enable_notifications()

Enables notifications for the battery level characteristic. Note: this function will raise an exception if notifications aren't possible

Returns
a Waitable which waits for the write to finish

disable_notifications()

Disables notifications for the battery level characteristic. Note: this function will raise an exception if notifications aren't possible

Returns
a Waitable which waits for the write to finish

classmethod find_in_database(*gattc_database*)

Return type
BatteryClient

blatann.services.current_time package

blatann.services.current_time.**add_current_time_service**(*gatts_database*, *enable_writes=False*, *enable_local_time_info=False*, *enable_reference_info=False*)

Adds a Current Time service to the given GATT server database

Parameters

- **gatts_database** (blatann.gatt.gatts.GattsDatabase) – The database to add the service to
- **enable_writes** – Makes the Current time and Local Time info characteristics writable so clients/centrals can update the server's time
- **enable_local_time_info** – Enables the Local Time characteristic in the service
- **enable_reference_info** – Enables the Reference Info characteristic in the service

Returns
The Current Time service

Return type
_CurrentTimeServer

Submodules

blatann.services.current_time.constants module

blatann.services.current_time.data_types module

class blatann.services.current_time.data_types.**DaylightSavingsTimeOffset**(*value*)

Bases: IntEnum

An enumeration.

standard_time = 0

half_hour_dst = 2

full_hour_dst = 4

two_hour_dst = 8

unknown = 255

static from_seconds(*seconds*)

Converts the DST offset in seconds to one of the above enums. Values which do not map directly to an above enum will be mapped to unknown. Valid values are essentially 0, 1800 (1/2 hr), 3600 (1 hr), and 7200 (2 hr)

Parameters
seconds – DST offset in seconds

Returns
The corresponding enum value

class blatann.services.current_time.data_types.**AdjustmentReasonType**(*value*)

Bases: IntEnum

An enumeration.

manual_time_update = 0

external_time_reference_update = 1

time_zone_change = 2

dst_change = 3

class blatann.services.current_time.data_types.**TimeSource**(*value*)

Bases: IntEnum

An enumeration.

unknown = 0

network_time_protocol = 1

gps = 2

radio_time_signal = 3

manual = 4

atomic_clock = 5

cellular_network = 6

class blatann.services.current_time.data_types.TimeAccuracy(*value*)

Bases: IntEnum

An enumeration.

out_of_range = 254

unknown = 255

class blatann.services.current_time.data_types.AdjustmentReason(**adjustment_reasons*)

Bases: *Bitfield*

bitfield_width = 8

bitfield_enum

alias of *AdjustmentReasonType*

class blatann.services.current_time.data_types.ExactTime256(*date*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.DayDateTime'>, <class 'blatann.services.ble_data_types.Uint8'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.current_time.data_types.CurrentTime(*date*, *adjustment_reason=None*)

Bases: *BleCompoundDataType*

Class used to report the current time and reason for adjustment

data_stream_types = [<class 'blatann.services.current_time.data_types.ExactTime256'>, <class 'blatann.services.current_time.data_types.AdjustmentReason'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.current_time.data_types.**LocalTimeInfo**(*timezone_offset_hrs=0.0*, *dst_offset=DaylightSavingsTimeOffset.standard_time*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.Int8'>, <class 'blatann.services.ble_data_types.Uint8'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.current_time.data_types.**ReferenceTimeInfo**(*source=TimeSource.unknown*, *accuracy_seconds=TimeAccuracy.unknown*, *hours_since_update=None*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

blatann.services.current_time.service module

class blatann.services.current_time.service.**CurrentTimeServer**(*service*, *is_writable=False*, *enable_local_time_info_char=False*, *enable_ref_time_info_char=False*)

Bases: object

property is_writable: bool

Gets whether or not the service was configured to allow writes to the Current Time and Local Time Info characteristics

property has_local_time_info: bool

Gets whether or not the service was configured to show the Local Time Info characteristic

property has_reference_time_info: bool

Gets whether or not the service was configured to show the Reference Time Info characteristic

property on_current_time_write: *Event*[*CurrentTimeServer*, *DecodedWriteEventArgs*[*CurrentTime*]]

Event that is triggered when a client writes to the Current Time Characteristic. Event emits a DecodedWriteEventArgs argument where the value is of type current_time.CurrentTime

property on_local_time_info_write: *Event*[*CurrentTimeServer*, *DecodedWriteEventArgs*[*LocalTimeInfo*]]

Event that is triggered when a client writes to the Local Time Info Characteristic (if present). Event emits a DecodedWriteEventArgs argument where the value is of type current_time.LocalTimeInfo

configure_automatic()

Configures the current time and local time info (if present) to use the system time

set_time(*date=None*, *adjustment_reason=None*, *characteristic_read_callback=None*)

Manually sets the time to report to the client.

If characteristic_read_callback is supplied, the function is called for future reads on that characteristic to fetch the current time If characteristic_read_callback is None, future reads will be based off of the base datetime given and the time passed

Parameters

- **date** (*datetime.datetime*) – The new base date to set the characteristic to. Future characteristic reads will base its time off of this value if characteristic_read_callback is not supplied. If the date is not supplied, will use the current system time (same as configure_automatic but doesn't configure local time info)
- **adjustment_reason** (AdjustmentReason) – Optional reason to give for the adjustment
- **characteristic_read_callback** – Optional callback to fetch subsequent time values. Function signature should take no parameters and return a datetime object

set_local_time_info(*timezone_hrs=0.0*, *dst_offset=DaylightSavingsTimeOffset.standard_time*)

Sets the local time info characteristic data. Only valid if has_local_time_info is True

Parameters

- **timezone_hrs** – The timezone to report, in hours
- **dst_offset** (DaylightSavingsTimeOffset) – The daylight savings time offset

Raises

InvalidOperationException if the service was not configured with the local time info

set_reference_info(*time_source=TimeSource.unknown*, *accuracy=TimeAccuracy.unknown*, *hours_since_update=None*)

Sets the time reference info characteristic data. Only valid if has_reference_time_info is True

Parameters

- **time_source** (TimeSource) – The time source to use
- **accuracy** (TimeAccuracy) – The accuracy to report
- **hours_since_update** – The number of hours since time reference has been updated

Raises

InvalidOperationException if the service was not configured with the reference info

classmethod add_to_database(*gatts_database*, *is_writable=False*, *enable_local_time_info_char=False*, *enable_ref_time_info_char=False*)

class blatann.services.current_time.service.**CurrentTimeClient**(*gattc_service*)

Bases: object

property on_current_time_updated: *Event*[*CurrentTimeClient*, *DecodedReadCompleteEventArgs*[*CurrentTime*]]

Event triggered when the server has updated its current time

property on_local_time_info_updated: *Event*[*CurrentTimeClient*, *DecodedReadCompleteEventArgs*[*LocalTimeInfo*]]

Event triggered when the server has updated its local time info

property on_reference_info_updated: *Event*[*CurrentTimeClient*, *DecodedReadCompleteEventArgs*[*ReferenceTimeInfo*]]

Event triggered when the server has updated its reference time info

property has_local_time_info: bool

property has_reference_info: bool

property can_enable_notifications: bool

property can_set_current_time: bool

property can_set_local_time_info: bool

read_time()

Reads the time from the server

Return type
EventWaitable[*CurrentTimeClient*, *DecodedReadCompleteEventArgs*[*CurrentTime*]]

set_time(*date*, *adjustment_reason=None*)

Sets the time on the server to the datetime provided

blatann.services.device_info package

blatann.services.device_info.**add_device_info_service**(*gatts_database*)

Return type
_DisServer

blatann.services.device_info.**find_device_info_service**(*gattc_database*)

Return type
_DisClient

Submodules

blatann.services.device_info.constants module

blatann.services.device_info.data_types module

class blatann.services.device_info.data_types.**PnpVendorSource**(*value*)

Bases: IntEnum

An enumeration.

bluetooth_sig = 1

usb_vendor = 2

class blatann.services.device_info.data_types.**PnpId**(*vendor_id_source*, *vendor_id*, *product_id*, *product_revision*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint16'>, <class 'blatann.services.ble_data_types.Uint16'>, <class 'blatann.services.ble_data_types.Uint16'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.device_info.data_types.**SystemId**(*manufacturer_id*, *organizationally_unique_id*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.Uint40'>, <class 'blatann.services.ble_data_types.Uint24'>]

encode()

Return type
ble_data_types.BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

blatann.services.device_info.service module

class blatann.services.device_info.service.**DisClient**(*gattc_service*)

Bases: _DeviceInfoService

get(*characteristic*)

get_system_id()

get_model_number()

get_serial_number()

get_firmware_revision()

get_hardware_revision()

get_software_revision()

get_manufacturer_name()

get_regulatory_certifications()

get_pnp_id()

classmethod find_in_database(*gattc_database*)

Return type
DisClient

class blatann.services.device_info.service.**DisServer**(*service*)

Bases: _DeviceInfoService

set(*characteristic*, *value*, *max_len=None*)

set_system_id(*system_id*)

set_model_number(*model_number*, *max_len=None*)

set_serial_number(*serial_number*, *max_len=None*)

set_firmware_revision(*firmware_revision*, *max_len=None*)

set_hardware_revision(*hardware_revision*, *max_len=None*)

set_software_revision(*software_revision*, *max_len=None*)

set_manufacturer_name(*manufacturer_name*, *max_len=None*)

set_regulatory_certifications(*certs*)

set_pnp_id(*pnp_id*)

classmethod add_to_database(*gatts_database*)

blatann.services.glucose package

blatann.services.glucose.**add_glucose_service**(*gatts_database*, *glucose_database*, *security_level=SecurityLevel.OPEN*, *include_context_characteristic=True*)

Adds the Glucose bluetooth service to the Gatt Server database

Parameters

- **gatts_database** (blatann.gatt.gatts.GattsDatabase) – The GATT database to add the service to
- **glucose_database** (IGlucoseDatabase) – The database which holds the glucose measurements
- **security_level** (SecurityLevel) – The security level for the record-access control point of the service
- **include_context_characteristic** – flag whether or not to include the optional context characteristic in the service. If this is False, any context stored with glucose measurements will not be reported.

Submodules

blatann.services.glucose.constants module

blatann.services.glucose.data_types module

class blatann.services.glucose.data_types.**GlucoseConcentrationUnits**(*value*)

Bases: IntEnum

The concentration units available for reporting glucose levels

kg_per_liter = 0

mol_per_liter = 1

class blatann.services.glucose.data_types.**GlucoseType**(*value*)

Bases: IntEnum

The glucose types available

capillary_whole_blood = 1

capillary_plasma = 2

venous_whole_blood = 3

venous_plasma = 4

arterial_whole_blood = 5

arterial_plasma = 6

undetermined_whole_blood = 7

undetermined_plasma = 8

interstitial_fluid = 9

control_solution = 10

class blatann.services.glucose.data_types.**SampleLocation**(*value*)

Bases: IntEnum

Location which the blood sample was taken

finger = 1

alternate_test_site = 2

earlobe = 3

control_solution = 4

unknown = 15

class blatann.services.glucose.data_types.**MedicationUnits**(*value*)

Bases: IntEnum

Available units to report medication values in

milligrams = 0

milliliters = 1

class blatann.services.glucose.data_types.**CarbohydrateType**(*value*)

Bases: IntEnum

The type of carbohydrate consumed by the user

breakfast = 1

lunch = 2

dinner = 3

snack = 4

drink = 5

supper = 6

brunch = 7

class blatann.services.glucose.data_types.**MealType**(*value*)

Bases: IntEnum

The type of meal consumed

preprandial = 1

postprandial = 2

fasting = 3

casual = 4

bedtime = 5

class blatann.services.glucose.data_types.**TesterType**(*value*)

Bases: IntEnum

Information about who tested the glucose levels

self = 1

health_care_professional = 2

lab_test = 3

not_available = 15

class blatann.services.glucose.data_types.**HealthStatus**(*value*)

Bases: IntEnum

Current health status of the user

minor_issues = 1

major_issues = 2

during_menses = 3

under_stress = 4

normal = 5

not_available = 15

class blatann.services.glucose.data_types.**MedicationType**(*value*)

Bases: IntEnum

Medication type consumed

rapid_acting_insulin = 1

short_acting_insulin = 2

intermediate_acting_insulin = 3

long_acting_insulin = 4

premixed_insulin = 5

class blatann.services.glucose.data_types.**SensorStatusType**(*value*)

Bases: IntEnum

The types of sensor statuses that can be communicated

battery_low = 0

sensor_malfunction = 1

sample_size_insufficient = 2

strip_insertion_error = 3

incorrect_strip_type = 4

result_above_range = 5

result_below_range = 6

sensor_temp_high = 7

sensor_temp_low = 8

sensor_read_interrupted = 9

general_device_fault = 10

time_fault = 11

class blatann.services.glucose.data_types.**SensorStatus**(**sensor_statuses*)

Bases: *Bitfield*

Class which holds the current sensor status information

bitfield_width = 16

bitfield_enum

alias of *SensorStatusType*

class blatann.services.glucose.data_types.**GlucoseFeatureType**(*value*)

Bases: IntEnum

Enumeration of the supported feature types to be reported using the Feature characteristic

low_battery_detection = 0

sensor_malfunction_detection = 1

sensor_sample_size = 2

strip_insertion_error_detection = 3

strip_type_error_detection = 4

sensor_result_high_low_detection = 5

sensor_temp_high_low_detection = 6

sensor_read_interrupt_detection = 7

general_device_fault = 8

time_fault = 9

multiple_bond = 10

class blatann.services.glucose.data_types.**GlucoseFeatures**(**supported_features*)

Bases: *Bitfield*

Class which holds the features of the glucose sensor and is reported to over bluetooth. This is the class used for the Feature characteristic.

bitfield_width = 16

bitfield_enum

alias of *GlucoseFeatureType*

class blatann.services.glucose.data_types.**GlucoseSample**(*glucose_type*, *sample_location*, *value*, *units=GlucoseConcentrationUnits.kg_per_liter*)

Bases: *BleCompoundDataType*

Holds the info about a glucose sample to be reported through the Glucose Measurement characteristic

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.glucose.data_types.**GlucoseMeasurement**(*sequence_number*, *measurement_time*, *time_offset_minutes=None*, *sample=None*, *sensor_status=None*, *context=None*)

Bases: *BleCompoundDataType*

Represents a single measurement taken and can be reported over BLE

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.glucose.data_types.**CarbsInfo**(*carbs_grams*, *carb_type*)

Bases: *BleCompoundDataType*

Holds information about the carbs consumed

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.glucose.data_types.**ExerciseInfo**(*duration_seconds*, *intensity_percent*)

Bases: *BleCompoundDataType*

Holds information about the exercise performed with the glucose sample

EXERCISE_DURATION_OVERRUN = 65535

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.glucose.data_types.**MedicationInfo**(*med_type*, *med_value*, *med_units=MedicationUnits.milligrams*)

Bases: *BleCompoundDataType*

Holds information about the medication administered

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.glucose.data_types.**GlucoseContext**(*sequence_number*, *carbs=None*, *meal_type=None*, *tester=None*, *health_status=None*, *exercise=None*, *medication=None*, *hba1c_percent=None*, *extra_flags=None*)

Bases: *BleCompoundDataType*

Class which holds the extra glucose context data associated with the glucose measurement

encode()

Return type
BleDataStream

classmethod decode(*stream*)

blatann.services.glucose.database module

class blatann.services.glucose.database.**IGlucoseDatabase**

Bases: object

Defines the interface required for the Glucose Service to fetch records and record info

first_record()

Gets the first (oldest) record in the database

Returns
The first record in the database, or None if no records in the database

Return type
GlucoseMeasurement

last_record()

Gets the last (newest) record in the database

Returns
The last record in the database, or None if no records in the database

Return type
GlucoseMeasurement

record_count(*min_seq_num=None*, *max_seq_num=None*)

Gets the number of records between the minimum and maximum sequence numbers provided. The min/max limits are inclusive.

Parameters

- **min_seq_num** – The minimum sequence number to get. If None, no minimum is requested
- **max_seq_num** – The maximum sequence number to get. If None, no maximum is requested

Returns
The number of records that fit the parameters specified

Return type
int

get_records(*min_seq_num=None*, *max_seq_num=None*)

Gets a list of the records between the minimum sequence and maximum sequence numbers provided. The min/max limits are inclusive.

Parameters

- **min_seq_num** – The minimum sequence number to get. If None, no minimum is requested
- **max_seq_num** – The maximum sequence number to get. If None, no maximum is requested

Returns
The list of glucose measurement records that fit the parameters

Return type
list[*GlucoseMeasurement*]

delete_records(*min_seq_num=None*, *max_seq_num=None*)

Deletes the records between the minimum sequence and maximum sequence numbers provided. The min/max limits are inclusive.

Parameters

- **min_seq_num** – The minimum sequence number to get. If None, no minimum is requested
- **max_seq_num** – The maximum sequence number to get. If None, no maximum is requested

Returns
The response code to send back for the operation

Return type
RacpResponseCode

class blatann.services.glucose.database.**BasicGlucoseDatabase**(*init_records=None*)

Bases: *IGlucoseDatabase*

Basic glucose database which simply stores the records in a sorted list, and provides a method for adding new records to the database.

delete_records(*min_seq_num=None*, *max_seq_num=None*)

See IGlucoseDatabase

record_count(*min_seq_num=None*, *max_seq_num=None*)

See IGlucoseDatabase

get_records(*min_seq_num=None*, *max_seq_num=None*)

See IGlucoseDatabase

first_record()

See IGlucoseDatabase

last_record()

See IGlucoseDatabase

add_record(*glucose_measurement*)

Adds a record to the database. NOTE: the measurement's sequence number must be unique within the database

Parameters
glucose_measurement (GlucoseMeasurement) – The measurement to add

blatann.services.glucose.racp module

class blatann.services.glucose.racp.**RacpOpcode**(*value*)

Bases: IntEnum

An enumeration.

report_stored_records = 1

delete_stored_records = 2

abort_operation = 3

report_number_of_records = 4

number_of_records_response = 5

response_code = 6

class blatann.services.glucose.racp.**RacpOperator**(*value*)

Bases: IntEnum

An enumeration.

null = 0

all_records = 1

less_than_or_equal_to = 2

greater_than_or_equal_to = 3

within_range_inclusive = 4

first_record = 5

last_record = 6

class blatann.services.glucose.racp.**FilterType**(*value*)

Bases: IntEnum

An enumeration.

sequence_number = 1

user_facing_time = 2

class blatann.services.glucose.racp.**RacpResponseCode**(*value*)

Bases: IntEnum

An enumeration.

success = 1

not_supported = 2

invalid_operator = 3

operator_not_supported = 4

invalid_operand = 5

no_records_found = 6

abort_not_successful = 7

procedure_not_completed = 8

operand_not_supported = 9

class blatann.services.glucose.racp.**RacpCommand**(*opcode*, *operator*, *filter_type=None*, *filter_params=None*)

Bases: *BleCompoundDataType*

get_filter_min_max()

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.glucose.racp.**RacpResponse**(*request_opcode=None*, *response_code=None*, *record_count=None*)

Bases: *BleCompoundDataType*

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

blatann.services.glucose.service module

class blatann.services.glucose.service.**GlucoseServer**(*service*, *glucose_database*, *security_level=SecurityLevel.OPEN*, *include_context_characteristic=True*)

Bases: object

set_features(*features*)

Sets the features for the Glucose Feature characteristic

Parameters
features (GlucoseFeatures) – The supported features of the sensor

classmethod add_to_database(*gatts_database*, *glucose_database*, *security_level=SecurityLevel.OPEN*, *include_context_characteristic=True*)

blatann.services.nordic_uart package

blatann.services.nordic_uart.**add_nordic_uart_service**(*gatts_database*, *max_characteristic_size=None*)

Adds a Nordic UART service to the database

Parameters

- **gatts_database** (blatann.gatt.gatts.GattsDatabase) – The database to add the service to
- **max_characteristic_size** – The size of the characteristic which determines the read/write chunk size. This should be tuned to the MTU size of the connection

Returns
The Nordic Uart Service

Return type
_Server

blatann.services.nordic_uart.**find_nordic_uart_service**(*gattc_database*)

Finds a Nordic UART service in the given GATT client database

Parameters
gattc_database (blatann.gatt.gattc.GattcDatabase) – the GATT client database to search

Returns
The UART service if found, None if not found

Return type
_Client

Submodules

blatann.services.nordic_uart.constants module

blatann.services.nordic_uart.service module

class blatann.services.nordic_uart.service.**NordicUartServer**(*service*, *max_characteristic_size=None*)

Bases: object

property on_data_received

Return type
Event

property on_write_complete

Return type
Event

property max_write_length

write(*data*)

classmethod add_to_database(*gatts_database*, *max_characteristic_size=None*)

class blatann.services.nordic_uart.service.**NordicUartClient**(*service*)

Bases: object

property on_data_received

Return type
Event

property on_write_complete

Return type
Event

property max_write_length

property is_initialized

initialize()

write(*data*)

classmethod find_in_database(*gattc_database*)

Return type
NordicUartClient

Submodules

blatann.services.ble_data_types module

class blatann.services.ble_data_types.**BleDataStream**(*value=b''*)

Bases: object

encode(*ble_type*, **values*)

encode_multiple(**ble_type_value_pairs*)

encode_if(*conditional*, *ble_type*, **values*)

encode_if_multiple(*conditional*, **ble_type_value_pairs*)

decode(*ble_type*)

decode_if(*conditional*, *ble_type*)

decode_multiple(**ble_types*)

decode_if_multiple(*conditional*, **ble_types*)

take(*num_bytes*)

take_all()

class blatann.services.ble_data_types.**BleCompoundDataType**

Bases: object

data_stream_types = []

encode_values(**values*)

Return type
BleDataStream

encode()

Return type
BleDataStream

classmethod **decode**(*stream*)

Returns
The values decoded from the stream

Return type
tuple

classmethod **encoded_size**()

class blatann.services.ble_data_types.**BleDataType**

Bases: object

classmethod **encode**(*value*)

classmethod **decode**(*stream*)

classmethod **encoded_size**()

class blatann.services.ble_data_types.**DoubleNibble**

Bases: *BleDataType*

classmethod **encode**(*value*)

classmethod **decode**(*stream*)

classmethod **encoded_size**()

class blatann.services.ble_data_types.**UnsignedIntegerBase**

Bases: *BleDataType*

signed = False

byte_count = 1

classmethod **encode**(*value*)

classmethod **decode**(*stream*)

classmethod **encoded_size**()

class blatann.services.ble_data_types.**SignedIntegerBase**

Bases: *UnsignedIntegerBase*

signed = True

class blatann.services.ble_data_types.**Int8**

Bases: *SignedIntegerBase*

byte_count = 1

class blatann.services.ble_data_types.**Uint8**

Bases: *UnsignedIntegerBase*

byte_count = 1

class blatann.services.ble_data_types.**Int16**

Bases: *SignedIntegerBase*

byte_count = 2

class blatann.services.ble_data_types.**Uint16**

Bases: *UnsignedIntegerBase*

byte_count = 2

class blatann.services.ble_data_types.**Uint24**

Bases: *UnsignedIntegerBase*

byte_count = 3

class blatann.services.ble_data_types.**Uint32**

Bases: *UnsignedIntegerBase*

byte_count = 4

class blatann.services.ble_data_types.**Int32**

Bases: *SignedIntegerBase*

byte_count = 4

class blatann.services.ble_data_types.**Uint40**

Bases: *UnsignedIntegerBase*

byte_count = 5

class blatann.services.ble_data_types.**Uint48**

Bases: *UnsignedIntegerBase*

byte_count = 6

class blatann.services.ble_data_types.**Uint56**

Bases: *UnsignedIntegerBase*

byte_count = 7

class blatann.services.ble_data_types.**Uint64**

Bases: *UnsignedIntegerBase*

byte_count = 8

class blatann.services.ble_data_types.**Int64**

Bases: *SignedIntegerBase*

byte_count = 8

class blatann.services.ble_data_types.**String**

Bases: *BleDataType*

classmethod encode(*value*)

classmethod decode(*stream*)

class blatann.services.ble_data_types.**SFloat**

Bases: *BleDataType*

class **ReservedMantissaValues**

Bases: object

POS_INFINITY = 2046

NEG_INFINITY = 2050

NAN = 2047

NRES = 2048

RESERVED = 2049

ALL_NAN = [2047, 2048, 2049]

classmethod **encode**(*value*)

classmethod **decode**(*stream*)

classmethod **encoded_size**()

class blatann.services.ble_data_types.**DateTime**(*dt*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.Uint16'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>, <class 'blatann.services.ble_data_types.Uint8'>]

encode()

Return type
BleDataStream

classmethod **decode**(*stream*)

Returns
The values decoded from the stream

Return type
tuple

class blatann.services.ble_data_types.**DayOfWeek**(*value*)

Bases: IntEnum

An enumeration.

unknown = 0

monday = 1

tuesday = 2

wednesday = 3

thursday = 4

friday = 5

saturday = 6

sunday = 7

class blatann.services.ble_data_types.**DayDateTime**(*dt*)

Bases: *BleCompoundDataType*

data_stream_types = [<class 'blatann.services.ble_data_types.DateTime'>, <class 'blatann.services.ble_data_types.Uint8'>]

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Return type
datetime.datetime

class blatann.services.ble_data_types.**Bitfield**

Bases: *BleCompoundDataType*

bitfield_width = 8

bitfield_enum = None

encode()

Return type
BleDataStream

classmethod decode(*stream*)

Returns
The values decoded from the stream

Return type
tuple

classmethod from_integer_value(*value*)

classmethod byte_count()

classmethod encoded_size()

blatann.services.decoded_event_dispatcher module

class blatann.services.decoded_event_dispatcher.**DecodedReadWriteEventDispatcher**(*owner*, *ble_type*, *event_to_raise*, *logger=None*)

Bases: object

decode(*data*)

blatann.utils package

blatann.utils.**setup_logger**(*name=None*, *level='DEBUG'*)

blatann.utils.**repr_format**(*obj*, **args*, ***kwargs*)

Helper function to format objects into strings in the format of: ClassName(param1=value1, param2=value2, ...)

Parameters

- **obj** – Object to get the class name from
- **args** – Optional tuples of (param_name, value) which will ensure ordering during format
- **kwargs** – Other keyword args to populate with

Returns

String which represents the object

class blatann.utils.**Stopwatch**

Bases: object

start()

stop()

mark()

property is_running

property start_time

property stop_time

property elapsed

class blatann.utils.**SynchronousMonotonicCounter**(*start_value=0*)

Bases: object

Utility class which implements a thread-safe monotonic counter

next()

blatann.utils.**snake_case_to_capitalized_words**(*string*)

class blatann.utils.**IntEnumWithDescription**(*_*, *description=''*)

Bases: int, Enum

An enumeration.

property description

Submodules

blatann.utils.queued_tasks_manager module

class blatann.utils.queued_tasks_manager.**QueuedTasksManagerBase**(*max_processing_items_at_once=1*)

Bases: Generic[T]

Handles queuing of tasks that can only be done one at a time

class **TaskFailure**(*reason=None*, *ignore_stack_trace=False*, *clear_all=False*)

Bases: object

clear_all()

blatann.waitables package

Submodules

blatann.waitables.connection_waitable module

class blatann.waitables.connection_waitable.**ConnectionWaitable**(*ble_device*, *current_peer*, *role=BLEGapRoles.periph*)

Bases: *Waitable*

wait(*timeout=None*, *exception_on_timeout=True*)

Return type
blatann.peer.Peer

class blatann.waitables.connection_waitable.**ClientConnectionWaitable**(*ble_device*, *peer*)

Bases: *ConnectionWaitable*

wait(*timeout=None*, *exception_on_timeout=True*)

Return type
blatann.peer.Client

class blatann.waitables.connection_waitable.**PeripheralConnectionWaitable**(*ble_device*, *peer*)

Bases: *ConnectionWaitable*

wait(*timeout=None*, *exception_on_timeout=True*)

Return type
blatann.peer.Peripheral

class blatann.waitables.connection_waitable.**DisconnectionWaitable**(*connected_peer*)

Bases: *Waitable*

blatann.waitables.event_waitable module

class blatann.waitables.event_waitable.**EventWaitable**(*event*)

Bases: *Waitable*, Generic[TSender, TEvent]

Waitable implementation which waits on an *Event*.

wait(*timeout=None*, *exception_on_timeout=True*)

Waits for the asynchronous operation to complete

> **Warning:** If this call times out, it cannot be (successfully) called again as it will clean up all event handlers for the waitable. This is done to remove lingering references to the waitable object through event subscriptions

Parameters

- **timeout** – How long to wait, or None to wait indefinitely
- **exception_on_timeout** – Flag to either throw an exception on timeout, or instead return None object(s)

Return type
Tuple[TypeVar(TSender), TypeVar(TEvent)]

Returns
The result of the asynchronous operation

Raises
TimeoutError

then(*callback*)

Registers a function callback that will be called when the asynchronous operation completes

> **Note:** Only a single callback is supported– subsequent calls to this method will overwrite previous callbacks

Parameters
callback (Callable[[TypeVar(TSender), TypeVar(TEvent)], None]) – The function to call when the async operation completes

Returns
This waitable object

class blatann.waitables.event_waitable.**IdBasedEventWaitable**(*event*, *event_id*)

Bases: *EventWaitable*, Generic[TSender, TEvent]

Extension of *EventWaitable* for high-churn events which require IDs to ensure the correct operation is waited upon, such as characteristic read, write and notify operations

blatann.waitables.scan_waitable module

class blatann.waitables.scan_waitable.**ScanFinishedWaitable**(*ble_device*)

Bases: *Waitable*

Waitable that triggers when a scan operation completes. It also provides a mechanism to acquire the received scan reports in real-time

property scan_reports: Iterable[*ScanReport*]

Iterable which yields the scan reports in real-time as they're received. The iterable will block until scanning has timed out/finished

wait(*timeout=None*, *exception_on_timeout=True*)

Waits for the scanning operation to complete then returns the scan report collection

Parameters

- **timeout** (Optional[float]) – How long to wait for, in seconds
- **exception_on_timeout** (bool) – Flag whether or not to throw an exception if the operation timed out. If false and a timeout occurs, will return None

Return type
ScanReportCollection

Returns
The scan report collection

blatann.waitables.waitable module

class blatann.waitables.waitable.**Waitable**(*n_args=1*)

Bases: object

Base class for an object which can be waited on for an operation to complete. This is a similar concept to concurrent.futures.Future where asynchronous operations can block the current thread, or register a handler to be called when it completes.

wait(*timeout=None*, *exception_on_timeout=True*)

Waits for the asynchronous operation to complete

> **Warning:** If this call times out, it cannot be (successfully) called again as it will clean up all event handlers for the waitable. This is done to remove lingering references to the waitable object through event subscriptions

Parameters

- **timeout** (Optional[float]) – How long to wait, or None to wait indefinitely
- **exception_on_timeout** – Flag to either throw an exception on timeout, or instead return None object(s)

Returns
The result of the asynchronous operation

Raises
TimeoutError

then(*callback*)

Registers a function callback that will be called when the asynchronous operation completes

Note: Only a single callback is supported– subsequent calls to this method will overwrite previous callbacks

Parameters
callback (Callable) – The function to call when the async operation completes

Returns
This waitable object

class blatann.waitables.waitable.**GenericWaitable**(*n_args=1*)

Bases: *Waitable*

Simple wrapper of a Waitable object which exposes a `notify` method so external objects can signal/trigger the waitable's response

notify(**results*)

class blatann.waitables.waitable.**EmptyWaitable**(**args*)

Bases: *Waitable*

Waitable class which will immediately return the args provided when waited on or when a callback function is registered

wait(*timeout=None*, *exception_on_timeout=True*)

Waits for the asynchronous operation to complete

Warning: If this call times out, it cannot be (successfully) called again as it will clean up all event handlers for the waitable. This is done to remove lingering references to the waitable object through event subscriptions

Parameters

- **timeout** – How long to wait, or `None` to wait indefinitely
- **exception_on_timeout** – Flag to either throw an exception on timeout, or instead return `None` object(s)

Returns
The result of the asynchronous operation

Raises
TimeoutError

then(*callback*)

Registers a function callback that will be called when the asynchronous operation completes

Note: Only a single callback is supported– subsequent calls to this method will overwrite previous callbacks

Parameters
callback – The function to call when the async operation completes

Returns
This waitable object

Submodules

blatann.device module

class blatann.device.**BleDevice**(*comport='COM1'*, *baud=1000000*, *log_driver_comms=False*, *notification_hw_queue_size=16*, *write_command_hw_queue_size=16*, *bond_db_filename='user'*)

Bases: *NrfDriverObserver*

Represents the Bluetooth device itself. Provides the high-level bluetooth APIs (Advertising, Scanning, Connections), configuration, and bond database.

Parameters

- **comport** – The port the nRF52 device lives on, e.g. `"COM3"`, `"/dev/ttyS0"`
- **baud** – The baud rate to use. By default, the connectivity firmware images for v0.3+ use 1M baud.
- **log_driver_comms** – debug flag which will enable extra-verbose logging of all communications to the nRF52 hardware
- **notification_hw_queue_size** – Hardware-based queue size to use for notifications. This queue lives within the nRF52 hardware itself and has memory usage implications based on MTU size, etc. This probably won't need to be changed, and from current testing the queue isn't fully exercised
- **write_command_hw_queue_size** – Hardware-based queue size to use for write commands (write w/o response). Same comments about notification queue apply here.
- **bond_db_filename** – Optional filename to use for loading/saving the bonding database. The supported file formats/extensions are: ".pkl" (legacy) and ".json". json is preferred.

  Two special values also exist:

  - `"user"` [default] - saves the database within the user's home directory (`~/.blatann/bonding_db.json`). This is useful for cases where you may not have write access to the python install location, want to persist the bonding database across virtualenvs, or limit the access to just the logged-in user
  - `"system"` - saves the database within this library's directory structure, wherever it is installed or imported from. Useful if you want the bonding database to be constrained to just that python/virtualenv installation

configure(*vendor_specific_uuid_count=10*, *service_changed=False*, *max_connected_peripherals=1*, *max_connected_clients=1*, *max_secured_peripherals=1*, *attribute_table_size=1408*, *att_mtu_max_size=247*, *event_length=6*)

Configures the BLE Device with the given settings.

Note: Configuration must be set before opening the device

Parameters

- **vendor_specific_uuid_count** – The Nordic hardware limits number of 128-bit Base UUIDs that the device can know about. This normally equals the number of custom services that are to be supported, since characteristic UUIDs are usually derived from the service base UUID.
- **service_changed** – Whether the Service Changed characteristic is exposed in the GAP service
- **max_connected_peripherals** – The maximum number of concurrent connections with peripheral devices
- **max_connected_clients** – The maximum number of concurrent connections with client devices (NOTE: blatann currently only supports 1)
- **max_secured_peripherals** – The maximum number of concurrent peripheral connections that will need security (bonding/pairing) enabled
- **attribute_table_size** – The maximum size of the attribute table. Increase this number if there's a lot of services/characteristics in your GATT database.
- **att_mtu_max_size** – The maximum ATT MTU size supported. The default supports an MTU which will fit into a single transmission if Data Length Extensions is set to its max (251)
- **event_length** – The number of 1.25ms event cycles to dedicate for each connection. The default value (6, =7.5ms) will support the max DLE length of 251 bytes. Minimum value is 2, typical values are 3-8 depending on desired throughput and number of concurrent connections

open(*clear_bonding_data=False*)

Opens the connection to the BLE device. Must be called prior to performing any BLE operations

Parameters

clear_bonding_data – Flag that the bonding data should be cleared prior to opening the device.

close()

Closes the connection to the BLE device. The connection to the device must be opened again to perform BLE operations.

clear_bonding_data()

Clears out all bonding data from the bond database. Any subsequent connections will require re-pairing.

property address: *BLEGapAddr*

The MAC Address of the BLE device

Getter

Gets the MAC address of the BLE device

Setter

Sets the MAC address for the device to use

Note: The MAC address cannot be changed while the device is advertising, scanning, or initiating a connection

---

property database: *GattsDatabase*

Read Only

The local database instance that is accessed by connected clients

property generic_access_service: *GenericAccessService*

Read Only

The Generic Access service in the local database

property max_mtu_size: int

Read Only

The maximum allowed ATT MTU size that was configured for the device

---

Note: The Max MTU size is set through *configure()*

---

set_tx_power(*tx_power*)

Sets the radio transmit power. This is used for all connections, advertising, active scanning, etc. Method can be called at any time

Valid transmit power values are -40, -20, -16, -12, -8, -4, 0, 3, and 4 dBm

Parameters
tx_power – The transmit power to use, in dBm

connect(*peer_address*, *connection_params=None*)

Initiates a connection to a peripheral peer with the specified connection parameters, or uses the default connection parameters if not specified. The connection will not be complete until the returned waitable either times out or reports the newly connected peer

Parameters

- **peer_address** (blatann.gap.gap_types.PeerAddress) – The address of the peer to connect to
- **connection_params** (blatann.gap.gap_types.ConnectionParameters) – Optional connection parameters to use. If not specified, uses the set default

Return type
PeripheralConnectionWaitable

Returns
A Waitable which can be used to wait until the connection is successful or times out. Waitable returns a peer.Peripheral object

set_default_peripheral_connection_params(*min_interval_ms*, *max_interval_ms*, *timeout_ms*, *slave_latency=0*)

Sets the default connection parameters for all subsequent connection attempts to peripherals. Refer to the Bluetooth specifications for the valid ranges

Parameters

- **min_interval_ms** (float) – The minimum desired connection interval, in milliseconds
- **max_interval_ms** (float) – The maximum desired connection interval, in milliseconds
- **timeout_ms** (int) – The connection timeout period, in milliseconds
- **slave_latency** (int) – The connection slave latency

set_default_security_params(*passcode_pairing*, *io_capabilities*, *bond*, *out_of_band*, *reject_pairing_requests=False*, *lesc_pairing=False*)

Sets the default security parameters for all subsequent connections to peripherals.

Parameters

- **passcode_pairing** (bool) – Flag indicating that passcode pairing is required
- **io_capabilities** (*BLEGapIoCaps*) – The input/output capabilities of this device
- **bond** (bool) – Flag indicating that long-term bonding should be performed
- **out_of_band** (bool) – Flag indicating if out-of-band pairing is supported
- **reject_pairing_requests** (Union[bool, *PairingPolicy*]) – Flag indicating that all security requests by the peer should be rejected
- **lesc_pairing** (bool) – Flag indicating that LE Secure Pairing methods are supported

set_privacy_settings(*enabled*, *resolvable_address=True*, *update_rate_seconds=900*)

Sets the privacy parameters for advertising and connections to the device. When enabled, a random private address will be advertised and updated at the provided interval.

Parameters

- **enabled** (bool) – True to enable device privacy. Note that only device privacy is supported, network privacy is not
- **resolvable_address** (bool) – True to use a private random resolvable address. If the address is resolvable, bonded peers can use the device's IRK to determine the device's actual public/random address.
- **update_rate_seconds** (int) – How often the address should be changed/updated, in seconds. Default is 900 (15min)

blatann.event_args module

class blatann.event_args.**GattOperationCompleteReason**(*value*)

Bases: Enum

The reason why a GATT operation completed

SUCCESS = 0

QUEUE_CLEARED = 1

CLIENT_DISCONNECTED = 2

SERVER_DISCONNECTED = 3

CLIENT_UNSUBSCRIBED = 4

FAILED = 5

TIMED_OUT = 6

class blatann.event_args.**EventArgs**

Bases: object

Base Event Arguments class

class blatann.event_args.**DisconnectionEventArgs**(*reason*)

Bases: *EventArgs*

Event arguments sent when a peer disconnects

class blatann.event_args.**MtuSizeUpdatedEventArgs**(*previous_mtu_size*, *current_mtu_size*)

Bases: *EventArgs*

Event arguments for when the effective MTU size on a connection is updated

class blatann.event_args.**DataLengthUpdatedEventArgs**(*tx_bytes*, *rx_bytes*, *tx_time_us*, *rx_time_us*)

Bases: *EventArgs*

Event arguments for when the Data Length of the link layer has been changed

class blatann.event_args.**PhyUpdatedEventArgs**(*status*, *phy_channel*)

Bases: *EventArgs*

Event arguments for when the phy channel is updated

class blatann.event_args.**ConnectionParametersUpdatedEventArgs**(*active_connection_params*)

Bases: *EventArgs*

Event arguments for when connection parameters between peers are updated

class blatann.event_args.**SecurityProcess**(*value*)

Bases: Enum

An enumeration.

ENCRYPTION = 0

PAIRING = 1

BONDING = 1

class blatann.event_args.**PairingCompleteEventArgs**(*status*, *security_level*, *security_process*)

Bases: *EventArgs*

Event arguments when pairing completes, whether it failed or was successful

class blatann.event_args.**SecurityLevelChangedEventArgs**(*security_level*)

Bases: *EventArgs*

class blatann.event_args.**PasskeyEntryEventArgs**(*key_type*, *resolve*)

Bases: *EventArgs*

Event arguments when a passkey needs to be entered by the user

resolve(*passkey=None*)

Submits the passkey entered by the user to the peer

Parameters

passkey (Union[str, int, None]) – The passkey entered by the user. If the key type is passcode, should be a 6-digit string or integer. Use None or an empty string to cancel.

class blatann.event_args.**PasskeyDisplayEventArgs**(*passkey*, *match_request*, *match_confirm_callback*)

Bases: *EventArgs*

Event arguments when a passkey needs to be displayed to the user. If match_request is set, the user must confirm that the passkeys match on both devices then send back the confirmation

match_confirm(*keys_match*)

If key matching was requested, this function responds with whether or not the keys matched correctly :type keys_match: :param keys_match: True if the keys matched, False if not

class blatann.event_args.**PeripheralSecurityRequestEventArgs**(*bond*, *mitm*, *lesc*, *keypress*, *is_bonded_device*, *resolver*)

Bases: *EventArgs*

Event arguments for when a peripheral requests security to be enabled on the connection. The application must choose how to handle the request: accept, reject, or force re-pairing (if device is bonded).

class Response(*value*)

Bases: Enum

An enumeration.

accept = 1

reject = 2

force_repair = 3

accept()

Accepts the security request. If device is already bonded will initiate encryption, otherwise will start the pairing process

reject()

Rejects the security request

force_repair()

Accepts the security request and initiates the pairing process, even if the device is already bonded

class blatann.event_args.**PairingRejectedReason**(*value*)

Bases: Enum

Reason why pairing was rejected

non_bonded_central_request = 1

non_bonded_peripheral_request = 2

bonded_peripheral_request = 3

bonded_device_repairing = 4

user_rejected = 5

class blatann.event_args.**PairingRejectedEventArgs**(*reason*)

Bases: *EventArgs*

Event arguments for when a pairing request was rejected locally

class blatann.event_args.**WriteEventArgs**(*value*)

Bases: *EventArgs*

Event arguments for when a client has written to a characteristic on the local database

class blatann.event_args.**DecodedWriteEventArgs**(*value*, *raw_value*)

Bases: *EventArgs*, Generic[TDecodedValue]

Event arguments for when a client has written to a characteristic on the local database and the value has been decoded into a data type

class blatann.event_args.**SubscriptionStateChangeEventArgs**(*subscription_state*)

Bases: *EventArgs*

Event arguments for when a client's subscription state has changed

class blatann.event_args.**NotificationCompleteEventArgs**(*notification_id*, *data*, *reason*)

Bases: *EventArgs*

Event arguments for when a notification has been sent to the client from the notification queue

Reason

alias of *GattOperationCompleteReason*

class blatann.event_args.**ReadCompleteEventArgs**(*read_id*, *value*, *status*, *reason*)

Bases: *EventArgs*

Event arguments for when a read has completed of a peripheral's characteristic

class blatann.event_args.**WriteCompleteEventArgs**(*write_id*, *value*, *status*, *reason*)

Bases: *EventArgs*

Event arguments for when a write has completed on a peripheral's characteristic

class blatann.event_args.**SubscriptionWriteCompleteEventArgs**(*write_id*, *value*, *status*, *reason*)

Bases: *EventArgs*

Event arguments for when changing the subscription state of a characteristic completes

class blatann.event_args.**NotificationReceivedEventArgs**(*value*, *is_indication*)

Bases: *EventArgs*

Event Arguments for when a notification or indication is received from the peripheral

class blatann.event_args.**DatabaseDiscoveryCompleteEventArgs**(*status*)

Bases: *EventArgs*

Event Arguments for when database discovery completes

class blatann.event_args.**DecodedReadCompleteEventArgs**(*read_id*, *value*, *status*, *reason*, *decoded_stream=None*)

Bases: *ReadCompleteEventArgs*, Generic[TDecodedValue]

Event Arguments for when a read on a peripheral's characteristic completes and the data stream returned is decoded. If unable to decode the value, the bytes read are still returned

static from_notification_complete_event_args(*noti_complete_event_args*, *decoded_stream=None*)

static from_read_complete_event_args(*read_complete_event_args*, *decoded_stream=None*)

blatann.event_type module

class blatann.event_type.**Event**(*name*)

Bases: Generic[TSender, TEvent]

Represents an event that can have handlers registered and deregistered. All handlers registered to an event should take in two parameters: the event sender and the event arguments.

Those familiar with the C#/.NET event architecture, this should look very similar, though registration is done using the `register()` method instead of `+= event_handler`

register(*handler*)

Registers a handler to be called whenever the event is emitted. If the given handler is already registered, function does not register the handler a second time.

This function can be used in a *with* context block which will automatically deregister the handler when the context is exited.

Example

```
>>> with device.client.on_connected.register(my_connected_handler):
>>>     # Do something, my_connected_handler will be deregistered upon leaving␣
↪this context
```

Parameters
handler (Callable[[TypeVar(TSender), TypeVar(TEvent)], None]) – The handler to register

Return type
EventSubscriptionContext[TypeVar(TSender), TypeVar(TEvent)]

Returns
a context block that can be used to automatically unsubscribe the handler

deregister(*handler*)

Deregisters a previously-registered handler so it no longer receives the event. If the given handler is not registered, function does nothing

Parameters
handler (Callable[[TypeVar(TSender), TypeVar(TEvent)], None]) – The handler to deregister

class blatann.event_type.**EventSource**(*name*, *logger=None*)

Bases: *Event*

Represents an Event object along with the controls to emit the events and notify handlers. This is done to "hide" the notify method from subscribers.

property has_handlers: bool

Gets if the event has any handlers subscribed to the event

clear_handlers()

Clears all handlers from the event

notify(*sender*, *event_args=None*)

Notifies all subscribers with the given sender and event arguments

class blatann.event_type.**EventSubscriptionContext**(*event*, *subscriber*)

Bases: Generic[TSender, TEvent]

blatann.exceptions module

exception blatann.exceptions.**BlatannException**

Bases: Exception

exception blatann.exceptions.**InvalidStateException**

Bases: *BlatannException*

exception blatann.exceptions.**InvalidOperationException**

Bases: *BlatannException*

exception blatann.exceptions.**TimeoutError**

Bases: *BlatannException*

exception blatann.exceptions.**DecodeError**

Bases: *BlatannException*

blatann.peer module

class blatann.peer.**PeerState**(*value*)

Bases: Enum

Connection state of the peer

DISCONNECTED = 0

Peer is disconnected

CONNECTING = 1

Peer is in the process of connecting

CONNECTED = 2

Peer is connected

class blatann.peer.**Peer**(*ble_device*, *role*, *connection_params=ConnectionParams([15-30] ms, timeout: 4000 ms, latency: 0*, *security_params=SecurityParameters(passcode_pairing=False, io=<BLEGapIoCaps.KEYBOARD_DISPLAY: 4>, bond=False, oob=False, lesc=False)*, *name=''*, *write_no_resp_queue_size=1*)

Bases: object

Object that represents a BLE-connected (or disconnected) peer

BLE_CONN_HANDLE_INVALID = 65535

Number of bytes that are header/overhead per MTU when sending a notification or indication

NOTIFICATION_INDICATION_OVERHEAD_BYTES = 3

property name: str

The name of the peer, if known. This property is for the user's benefit to name certain connections. The name is also saved in the case that the peer is subsequently bonded to and can be looked up that way in the bond database

Note: For central peers this name is unknown unless set by the setter. For peripheral peers the name is defaulted to the one found in the advertising payload, if any.

Getter
Gets the name of the peer

Setter
Sets the name of the peer

property connected: bool

Read Only

Gets if this peer is currently connected

property rssi: Optional[int]

Read Only

Gets the RSSI from the latest connection interval, or None if RSSI reporting is not enabled.

Note: In order for RSSI information to be available, *start_rssi_reporting()* must be called first.

---

property bytes_per_notification: int

Read Only

The maximum number of bytes that can be sent in a single notification/indication

property is_peripheral: bool

Read Only

Gets if this peer is a peripheral (the local device acting as a central/client)

property is_client: bool

Read Only

Gets if this peer is a Client (the local device acting as a peripheral/server)

property is_previously_bonded: bool

Read Only

Gets if the peer has bonding information stored in the bond database (the peer was bonded to in a previous connection)

property preferred_connection_params: *ConnectionParameters*

Read Only

The connection parameters that were negotiated for this peer

property active_connection_params: *ActiveConnectionParameters*

Read Only

The active connection parameters in use with the peer. If the peer is disconnected, this will return the connection parameters last used

property mtu_size: int

Read Only

The current size of the MTU for the connection to the peer

property max_mtu_size: int

Read Only

The maximum allowed MTU size. This is set when initially configuring the BLE Device

property preferred_mtu_size: *int*

The user-set preferred MTU size. Defaults to the Bluetooth default MTU size (23). This is the value that will be negotiated during an MTU Exchange but is not guaranteed in the case that the peer has a smaller MTU

Getter
Gets the preferred MTU size that was configured

Setter
Sets the preferred MTU size to use for MTU exchanges

property preferred_phy: *Phy*

The PHY that is preferred for this connection. This value is used for Peer-initiated PHY update procedures and as the default for *update_phy()*.

Default value is `Phy.auto`

Getter
Gets the preferred PHY

Setter
Sets the preferred PHY

property phy_channel: *Phy*

Read Only

The current PHY in use for the connection

property database: *GattcDatabase*

Read Only

The GATT database of the peer.

Note: This is not useful until services are discovered first

---

property on_connect: *Event*[*Peer*, None]

Event generated when the peer connects to the local device

property on_disconnect: *Event*[*Peer*, *DisconnectionEventArgs*]

Event generated when the peer disconnects from the local device

property on_rssi_changed: *Event*[*Peer*, int]

Event generated when the RSSI has changed for the connection

property on_mtu_exchange_complete: *Event*[*Peer*, *MtuSizeUpdatedEventArgs*]

Event generated when an MTU exchange completes with the peer

property on_mtu_size_updated: *Event*[*Peer*, *MtuSizeUpdatedEventArgs*]

Event generated when the effective MTU size has been updated on the connection

property on_connection_parameters_updated: *Event*[*Peer*, *ConnectionParametersUpdatedEventArgs*]

Event generated when the connection parameters with this peer is updated

property on_data_length_updated: *Event*[*Peer*, *DataLengthUpdatedEventArgs*]

Event generated when the link layer data length has been updated

property on_phy_updated: *Event*[*Peer*, *PhyUpdatedEventArgs*]

Event generated when the PHY in use for this peer has been updated

property on_database_discovery_complete: *Event*[*Peripheral*, *DatabaseDiscoveryCompleteEventArgs*]

Event that is triggered when database discovery has completed

disconnect(*status_code=BLEHci.remote_user_terminated_connection*)

Disconnects from the peer, giving the optional status code. Returns a waitable that will trigger when the disconnection is complete. If the peer is already disconnected, the waitable will trigger immediately

Parameters
status_code – The HCI Status code to send back to the peer

Return type
DisconnectionWaitable

Returns
A waitable that will trigger when the peer is disconnected

set_connection_parameters(*min_connection_interval_ms*, *max_connection_interval_ms*, *connection_timeout_ms*, *slave_latency=0*)

Sets the connection parameters for the peer and starts the connection parameter update process (if connected)

Note: Connection interval values should be a multiple of 1.25ms since that is the granularity allowed in the Bluetooth specification. Any non-multiples will be rounded down to the nearest 1.25ms. Additionally, the connection timeout has a granularity of 10 milliseconds and will also be rounded as such.

Parameters

- **min_connection_interval_ms** (float) – The minimum acceptable connection interval, in milliseconds
- **max_connection_interval_ms** (float) – The maximum acceptable connection interval, in milliseconds
- **connection_timeout_ms** (int) – The connection timeout, in milliseconds
- **slave_latency** – The slave latency allowed, which regulates how many connection intervals the peripheral is allowed to skip before responding

Return type
Optional[*EventWaitable*[*Peer*, *ConnectionParametersUpdatedEventArgs*]]

Returns
If the peer is connected, this will return a waitable that will trigger when the update completes with the new connection parameters. If disconnected, returns None

update_connection_parameters()

Starts the process to re-negotiate the connection parameters using the configured preferred connection parameters

Return type
EventWaitable[*Peer*, *ConnectionParametersUpdatedEventArgs*]

Returns
A waitable that will trigger when the connection parameters are updated

exchange_mtu(*mtu_size=None*)

Initiates the MTU Exchange sequence with the peer device.

If the MTU size is not provided *preferred_mtu_size* value will be used. If an MTU size is provided preferred_mtu_size will be updated to the given value.

Parameters
mtu_size – Optional MTU size to use. If provided, it will also updated the preferred MTU size

Return type
EventWaitable[*Peer*, *MtuSizeUpdatedEventArgs*]

Returns
A waitable that will trigger when the MTU exchange completes

update_data_length(*data_length=None*)

Starts the process which updates the link layer data length to the optimal value given the MTU. For best results call this method after the MTU is set to the desired size.

Parameters
data_length (Optional[int]) – Optional value to override the data length to. If not provided, uses the optimal value based on the current MTU

Return type
EventWaitable[*Peripheral*, *DataLengthUpdatedEventArgs*]

Returns
A waitable that will trigger when the process finishes

update_phy(*phy=None*)

Performs the PHY update procedure, negotiating a new PHY (1Mbps, 2Mbps, or coded PHY) to use for the connection. Performing this procedure does not guarantee that the PHY will change based on what the peer supports.

Parameters
phy (Optional[*Phy*]) – Optional PHY to use. If None, uses the *preferred_phy* attribute. If not None, the preferred PHY is updated to this value.

Return type
EventWaitable[*Peer*, *PhyUpdatedEventArgs*]

Returns
An event waitable that triggers when the phy process completes

discover_services()

Starts the database discovery process of the peer. This will discover all services, characteristics, and descriptors on the peer's database.

Return type
EventWaitable[*Peer*, *DatabaseDiscoveryCompleteEventArgs*]

Returns
a Waitable that will trigger when service discovery is complete

start_rssi_reporting(*threshold_dbm=None*, *skip_count=1*)

Starts collecting RSSI readings for the connection

Parameters

- **threshold_dbm** (Optional[int]) – Minimum change in dBm before triggering an RSSI changed event. The default value None disables the RSSI event (RSSI polled via the *rssi* property)
- **skip_count** – Number of RSSI samples with a change of threshold_dbm or more before sending a new RSSI update event. Parameter ignored if threshold_dbm is None

Return type
EventWaitable[*Peer*, int]

Returns
a Waitable that triggers once the first RSSI value is received, if threshold_dbm is not None

stop_rssi_reporting()

Stops collecting RSSI readings. Once stopped, *rssi* will return None

class blatann.peer.**Peripheral**(*ble_device*, *peer_address*, *connection_params=ConnectionParams([15-30] ms, timeout: 4000 ms, latency: 0, security_params=SecurityParameters(passcode_pairing=False, io=<BLEGapIoCaps.KEYBOARD_DISPLAY: 4>, bond=False, oob=False, lesc=False), name='', write_no_resp_queue_size=1*)

Bases: *Peer*

Object which represents a BLE-connected device that is acting as a peripheral/server (local device is client/central)

set_conn_param_request_handler(*handler*)

Configures a function callback to handle when a connection parameter request is received from the peripheral and allows the user to decide how to handle the peripheral's requested connection parameters.

The callback is passed in 2 positional parameters: this Peripheral object and the desired ``ConnectionParameter``s received in the request. The callback should return the desired connection parameters to use, or None to reject the request altogether.

Parameters
handler (Callable[[*Peripheral*, *ConnectionParameters*], Optional[*ConnectionParameters*]]) – The callback to determine which connection parameters to negotiate when an update request is received from the peripheral

accept_all_conn_param_requests()

Sets the connection parameter request handler to a callback that accepts any connection parameter update requests received from the peripheral. This is the same as calling set_conn_param_request_handler with a callback that simply returns the connection parameters passed in.

This is the default functionality.

reject_conn_param_requests()

Sets the connection parameter request handler to a callback that rejects all connection parameter update requests received from the peripheral. This is same as calling set_conn_param_request_handler with a callback that simply returns None

class blatann.peer.**Client**(*ble_device*, *connection_params=ConnectionParams([15-30] ms, timeout: 4000 ms, latency: 0, security_params=SecurityParameters(passcode_pairing=False, io=<BLEGapIoCaps.KEYBOARD_DISPLAY: 4>, bond=False, oob=False, lesc=False), name='', write_no_resp_queue_size=1*)

Bases: *Peer*

Object which represents a BLE-connected device that is acting as a client/central (local device is peripheral/server)

blatann.uuid module

class blatann.uuid.**Uuid**(*nrf_uuid=None*, *description=''*)

Bases: object

Base class for UUIDs

property descriptive_string: str

class blatann.uuid.**Uuid128**(*uuid*, *description=''*)

Bases: *Uuid*

Represents a 128-bit UUID

property uuid_base: List[int]

Read Only

The base of the 128-bit UUID which can be used to create other UUIDs with the same base

property uuid16: int

Read Only

The 16-bit representation of the 128-bit UUID

new_uuid_from_base(*uuid16*)

Creates a new 128-bit UUID with the same base as this UUID, replacing out the 16-bit individual identifier with the value provided

Parameters
uuid16 (Union[str, int, *Uuid16*]) – The new 16-bit UUID to append to the base. Should either be a 16-bit integer, a hex string of the value (without the leading '0x'), or a Uuid16 object

Return type
Uuid128

Returns
The newly created UUID

classmethod combine_with_base(*uuid16*, *uuid128_base*)

Combines a 16-bit UUID with a 128-bit UUID base and returns the new UUID

Parameters

- **uuid16** (Union[str, int, *Uuid16*]) – The 16-bit UUID to use. See new_uuid_from_base for format.
- **uuid128_base** (Union[str, bytes, List[int]]) – The 128-bit base UUID to use. See __init__ for format.

Return type
Uuid128

Returns
The created UUID

class blatann.uuid.**Uuid16**(*uuid*, *description=''*)

Bases: *Uuid*

Represents a 16-bit "short form" UUID

blatann.uuid.**generate_random_uuid16**()

Generates a random 16-bit UUID

Return type
Uuid16

Returns
The generated 16-bit UUID

blatann.uuid.**generate_random_uuid128**()

Generates a random 128-bit UUID

Return type
Uuid128

Returns
The generated 128-bit UUID

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

b

INDEX

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X