Python POD API

Release v1.4.0

Thresa Kelly

Aug 07, 2023

CONTENTS:

PYTHON POD API PROJECT

1.1 PodApi package

1.1.1 Subpackages

1.1.1.1 PodApi.Commands package

Submodules

PodApi.Commands.PodCommands module

class PodApi.Commands.PodCommands.**CommandSet**

> Bases: `object`
>
> POD_Commands manages a dictionary containing available commands for a POD device.
>
> **__commands**
>
> > Dictionary containing the available commands for a POD device. Each entry is formatted as { key(command number) : value([command name, number of argument ASCII bytes, number of return bytes, binary flag) }.
> >
> > **Type**
> > > dict[int,list[str|tuple[int]|bool]]
>
> **AddCommand**(*commandNumber: int*, *commandName: str*, *argumentBytes: tuple[int]*, *returnBytes: tuple[int]*, *isBinary: bool*, *description: str*) → bool
>
> > Adds a command entry to the current commands dictionary (__commands) if the command does not exist.
> >
> > **Parameters**
> >
> > - **commandNumber** (`int`) – Integer of the command number.
> >
> > - **commandName** (`str`) – String of the command's name.
> >
> > - **argumentBytes** (`tuple[int]`) – Integer of the number of bytes in the argument.
> >
> > - **returnBytes** (`tuple[int]`) – Integer of the number of bytes in the return.
> >
> > - **isBinary** (`bool`) – Boolean flag to mark if the command is binary (True) or standard (False).
> >
> > - **description** (`str`) – String description of the command.

> **Returns**
> True if the command was successfully added, False if the command could not be added because it already exists.
>
> **Return type**
> bool

ArgumentHexChar(*cmd: int | str*) → tuple[int] | None

Gets the tuple for the number of hex characters in the argument for a given command.

> **Parameters**
> **cmd** (`int | str`) – Integer command number or string command name.
>
> **Returns**
> Tuple representing the number of bytes in the argument for cmd. If the command could not be found, return None.
>
> **Return type**
> tuple[int]|None

CommandNumberFromName(*name: str*) → int | None

Gets the command number from the command dictionary using the command's name.

> **Parameters**
> **name** (`str`) – String of the command's name.
>
> **Returns**
> Integer representing the command number. If the command could not be found, return None.
>
> **Return type**
> int|None

Description(*cmd: int | str*) → str | None

Gets the description for a given command.

> **Parameters**
> **cmd** (`int | str`) – Integer command number or string command name.
>
> **Returns**
> String description for the command. If the command could not be found, return None.
>
> **Return type**
> str|None

DoesCommandExist(*cmd: int | str*) → bool

Checks if a command exists in the __commands dictionary.

> **Parameters**
> **cmd** (`int | str`) – Integer command number or string command name.
>
> **Returns**
> True if the command exists, false otherwise.
>
> **Return type**
> bool

static GetBasicCommands() → dict[int, list[str | tuple[int] | bool]]

Creates a dictionary containing the basic POD command set (0,1,2,3,4,5,6,7,8,9,10,11,12).

> **Returns**
> Dictionary containing the available commands for this POD device. Each entry is formatted as

> { key(command number) : value([command name, number of argument ASCII bytes, number of return bytes, binary flag, description) }.

> **Return type**
>> dict[int,list[str|tuple[int]|bool|str]]

GetCommands() → dict[int, list[str | tuple[int] | bool]]

> Gets the contents of the current command dictionary (__commands).

>> **Returns**
>>> Dictionary containing the available commands for a POD device. Each entry is formatted as { key(command number) : value([command name, number of argument ASCII bytes, number of return bytes, binary flag, description) }.

>> **Return type**
>>> dict[int, list[str|tuple[int]|bool|str]]

IsCommandBinary(*cmd: int | str*) → bool | None

> Gets the binary flag for a given command.

>> **Parameters**
>>> **cmd** (`int | str`) – Integer command number or string command name.

>> **Returns**
>>> Boolean flag that is True if the command is binary and False if standard. If the command could not be found, return None.

>> **Return type**
>>> bool|None

static NoValue() → int

> Gets value of __NOVALUE.

>> **Returns**
>>> Value of __NOVALUE.

>> **Return type**
>>> int

RemoveCommand(*cmd: int | str*) → bool

> Removes the entry for a given command in __commands dictionary.

>> **Parameters**
>>> **cmd** (`int | str`) – Integer command number or string command name.

>> **Returns**
>>> True if the command was successfully removed, False if the command does not exist.

>> **Return type**
>>> bool

RestoreBasicCommands() → None

> Sets the current commands (__commands) to the basic POD command set.

ReturnHexChar(*cmd: int | str*) → tuple[int] | None

> Gets the tuple for the number of hex characters in the return for a given command.

>> **Parameters**
>>> **cmd** (`int | str`) – Integer command number or string command name.

> **Returns**
>> Tuple representing the number of hex characters in the return for cmd. If the command could not be found, return None.
>
> **Return type**
>> tuple[int]|None

Search(*cmd: int | str*, *idx: int | None = None*) → str | tuple[int] | bool | None

> Searches the __commands dictionary for the command.
>
> **Parameters**
>> - **cmd** (`int | str`) – Integer command number or string command name.
>> - **idx** (`int, optional`) – Index for the desired value in the command information list. Defaults to None.
>
> **Returns**
>> If an idx was given, this returns the idx value of the command information list if the command was found (None otherwise). If no idx is given, this returns true if the command is found (False otherwise).
>
> **Return type**
>> str|tuple[int]|bool|str|None

static U16() → int

> Gets value of __U16.
>
> **Returns**
>> Value of __U16.
>
> **Return type**
>> int

static U32() → int

> Gets value of __U32.
>
> **Returns**
>> Value of __U32.
>
> **Return type**
>> int

static U8() → int

> Gets value of __U8.
>
> **Returns**
>> Value of __U8.
>
> **Return type**
>> int

__ARGUMENTS: int = 1

> Class-level integer representing the index key for the number of bytes in an argument for __commands list values.

__BINARY: int = 3

> Class-level integer representing the index key for the binary flag for __commands list values.

__DESCRIPTION: int = 4

> Class-level integer representing the index key for the description for __commands list values.

__NAME: int = 0

   Class-level integer representing the index key for the command name for __commands list values.

__NOVALUE: int = -1

   Class-level integer used to mark when a list item in __commands means 'no value' or is undefined.

__RETURNS: int = 2

   Class-level integer representing the index key for the number of bytes in the return for __commands list values.

__U16: int = 4

   Class-level integer representing the number of hexadecimal characters for an unsigned 16-bit value.

__U32: int = 8

   Class-level integer representing the number of hexadecimal characters for an unsigned 32-bit value.

__U8: int = 2

   Class-level integer representing the number of hexadecimal characters for an unsigned 8-bit value.

Module contents

1.1.1.2 PodApi.Devices package

Subpackages

PodApi.Devices.SerialPorts package

Submodules

PodApi.Devices.SerialPorts.PortAccess module

class PodApi.Devices.SerialPorts.PortAccess.**FindPorts**

   Bases: object

   Contains methods for the user to view and select a serial port.

   static ChoosePort(*forbidden: list[str] = []*) → str

   Systems checks user's Operating System, and chooses ports accordingly.

   > **Parameters**
   > **forbidden** (`list[str], optional`) – List of port names that the user should not use. This
   > may be because these ports are already in use or that the port is not a POD device. Defaults
   > to [].
   >
   > **Returns**
   > String name of the port.
   >
   > **Return type**
   > str

   static GetAllPortNames() → list[str]

   Finds all the available COM ports on the user's computer and appends them to an accessible list.

   > **Returns**
   > List containing the names of available COM ports.

> **Return type**
>> list[str]

static GetSelectPortNames(*forbidden: list[str] = []*) → list[str]

> Gets the names of all available ports.
>
>> **Parameters**
>>> **forbidden** (`list[str], optional`) – List of port names that the user should not use. This may be because these ports are already in use or that the port is not a POD device. Defaults to [].
>>
>> **Returns**
>>> List of port names.
>>
>> **Return type**
>>> list[str]

static _ChoosePortLinux(*forbidden: list[str] = []*) → str

> User picks Serial port in Linux.
>
>> **Parameters**
>>> **forbidden** (`list[str], optional`) – List of port names that the user should not use. This may be because these ports are already in use or that the port is not a POD device. Defaults to [].
>>
>> **Returns**
>>> String name of the port.
>>
>> **Return type**
>>> str

static _ChoosePortWindows(*forbidden: list[str] = []*) → str

> User picks COM port in Windows.
>
>> **Parameters**
>>> **forbidden** (`list[str], optional`) – List of port names that the user should not use. This may be because these ports are already in use or that the port is not a POD device. Defaults to [].
>>
>> **Returns**
>>> String name of the port.
>>
>> **Return type**
>>> str

PodApi.Devices.SerialPorts.SerialComm module

class PodApi.Devices.SerialPorts.SerialComm.PortIO(*port: str | int, baudrate: int = 9600*)

> Bases: `object`
>
> COM_io handles serial communication (read/write) using COM ports.
>
> **__serialInst**
>> Instance-level serial COM port.
>>
>>> **Type**
>>>> Serial

CloseSerialPort() → None

Closes the instance serial port if it is open.

Flush() → bool

Reset the input and output serial buffer.

> **Returns**
>> True of the buffers are flushed, False otherwise.
>
> **Return type**
>> bool

GetPortName() → str | None

Gets the name of the open port.

> **Returns**
>> If the serial port is open, it will return a string of the port's name. If the port is closed, it will return None.
>
> **Return type**
>> str|None

IsSerialClosed() → bool

Returns False if the serial instance port is open, True otherwise.

> **Returns**
>> True if the COM port is closed, False otherwise.
>
> **Return type**
>> bool

IsSerialOpen() → bool

Returns True if the serial instance port is open, false otherwise.

> **Returns**
>> True if the COM port is open, False otherwise.
>
> **Return type**
>> bool

OpenSerialPort(*port: str | int*, *baudrate: int = 9600*) → None

First, it closes the serial port if it is open. Then, it opens a serial port with a set baud rate.

> **Parameters**
>> - **port** (`str | int`) – String of the serial port to be opened.
>> - **baudrate** (`int, optional`) – Integer baud rate of the opened serial port. Defaults to 9600.
>
> **Raises**
>> **Exception** – Port does not exist.

Read(*numBytes: int*, *timeout_sec: int | float = 5*) → bytes | None

Reads a specified number of bytes from the open serial port.

> **Parameters**
>> - **numBytes** (`int`) – Integer number of bytes to read.
>> - **timeout_sec** (`int|float, optional`) – Time in seconds to wait for serial data. Defaults to 5.

> **Raises**
> **Exception** – Timeout for serial read.

> **Returns**
> If the serial port is open, it will return a set number of read bytes. If it is closed, it will return None.

> **Return type**
> bytes|None

ReadLine() → bytes | None

Reads until a new line is read from the open serial port.

> **Returns**
> If the serial port is open, it will return a complete read line. If closed, it will return None.

> **Return type**
> bytes|None

ReadUntil(*eol: bytes*) → bytes | None

Reads until a set character from the open serial port.

> **Parameters**
> **eol** (*bytes*) – end-of-line character.

> **Returns**
> If the serial port is open, it will return a read line ending in eol. If closed, it will return None.

> **Return type**
> bytes|None

SetBaudrate(*baudrate: int*) → bool

Sets the baud rate of the serial port

> **Parameters**
> **baudrate** (*int*) – Baud rate, or signals per second.

> **Returns**
> True if the baudrate was set, False otherwise.

> **Return type**
> bool

Write(*message: bytes*) → None

Write a set message to the open serial port.

> **Parameters**
> **message** (*bytes*) – byte string containing the message to write.

__BuildPortName(*port: str | int*) → str

Converts the port parameter into the "COM"+<number> format for Windows or "/dev/tty…"+<number> for Linux.

> **Parameters**
> **port** (*str | int*) – Name of a COM port. Can be an integer or string.

> **Returns**
> Name of the COM port.

> **Return type**
> str

Module contents

Submodules

PodApi.Devices.BasicPodProtocol module

class PodApi.Devices.BasicPodProtocol.**Pod**(*port: str | int*, *baudrate: int = 9600*)

> Bases: `object`
>
> POD_Basics handles basic communication with a generic POD device, including reading and writing packets and packet interpretation.
>
> **_port**
>
> > Instance-level COM_io object, which handles the COM port
> >
> > **Type**
> > > COM_io
>
> **_commands**
>
> > Instance-level POD_Commands object, which stores information about the commands available to this POD device.
> >
> > **Type**
> > > POD_Commands
>
> **static ChoosePort**(*forbidden: list[str] = []*) → str
>
> > Systems checks user's Operating System, and chooses ports accordingly.
> >
> > **Parameters**
> > > **forbidden** (`list[str], optional`) – List of port names that are already used. Defaults to [].
> >
> > **Returns**
> > > String name of the port.
> >
> > **Return type**
> > > str
>
> **FlushPort**() → bool
>
> > Reset the input and output serial port buffer.
> >
> > **Returns**
> > > True of the buffers are flushed, False otherwise.
> >
> > **Return type**
> > > bool
>
> **GetDeviceCommands**() → dict[int, list[str | tuple[int] | bool]]
>
> > Gets the dictionary containing the class instance's available POD commands.
> >
> > **Returns**
> > > Dictionary containing the available commands and their information.Formatted as key(command number) : value([command name, number of argument ASCII bytes, number of return bytes, binary flag])
> >
> > **Return type**
> > > dict[int, list[str|tuple[int]|bool]]

GetPODpacket(*cmd: str | int*, *payload: int | bytes | tuple[int | bytes] | None = None*) → bytes

    Builds a POD packet and writes it to a POD device via COM port. If an integer payload is give, the method will convert it into a bytes string of the length expected by the command. If a bytes payload is given, it must be the correct length.

        Parameters

- **cmd** (`str | int`) – Command number.

- **payload** (`int | bytes | tuple[int | bytes]`, `optional`) – None when there is no payload. If there is a payload, set to an integer value, bytes string, or tuple. Defaults to None.

        Raises

- **Exception** – POD command does not exist.

- **Exception** – POD command requires a payload.

        Returns

        Bytes string of the POD packet.

        Return type

        bytes

static GetU(*u: int*) → int

    number of hexadecimal characters for an unsigned u-bit value.

        Parameters

        u (`int`) – 8, 16, or 32 bits. Enter any other number for NOVALUE.

        Returns

        number of hexadecimal characters for an unsigned u-bit value.

        Return type

        int

ReadPODpacket(*validateChecksum: bool = True*, *timeout_sec: int | float = 5*) → *Packet*

    Reads a complete POD packet, either in standard or binary format, beginning with STX and ending with ETX. Reads first STX and then starts recursion.

        Parameters

- **validateChecksum** (`bool`, `optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.

- **timeout_sec** (`int|float`, `optional`) – Time in seconds to wait for serial data. Defaults to 5.

        Returns

        POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).

        Return type

        Packet

SetBaudrateOfDevice(*baudrate: int*) → bool

    If the port is open, it will change the baud rate to the parameter's value.

        Parameters

        baudrate (`int`) – Baud rate to set for the open serial port.

        Returns

        True if successful at setting the baud rate, false otherwise.

> **Return type**
> bool

WritePacket(*cmd: str | int*, *payload: int | bytes | tuple[int | bytes] | None = None*) → *PacketStandard*

> Builds a POD packet and writes it to the POD device.
>
> > **Parameters**
> >
> > - **cmd** (`str | int`) – Command number.
> >
> > - **payload** (`int | bytes | tuple[int | bytes]`, `optional`) – None when there is no payload. If there is a payload, set to an integer value, bytes string, or tuple. Defaults to None.
> >
> > **Returns**
> > Packet that was written to the POD device.
> >
> > **Return type**
> > Packet_Standard

WriteRead(*cmd: str | int*, *payload: int | bytes | tuple[int | bytes] | None = None*, *validateChecksum: bool = True*) → bytes

> Writes a command with optional payload to POD device, then reads (once) the device response.
>
> > **Parameters**
> >
> > - **cmd** (`str | int`) – Command number.
> >
> > - **payload** (`int | bytes | tuple[int|bytes]`, `optional`) – None when there is no payload. If there is a payload, set to an integer value or a bytes string. Defaults to None.
> >
> > - **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
> >
> > **Returns**
> > Bytes string containing a POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).
> >
> > **Return type**
> > bytes

_ReadPODpacket_Recursive(*validateChecksum: bool = True*) → *Packet*

> Reads the command number. If the command number ends in ETX, the packet is returned. Next, it checks if the command is allowed. Then, it checks if the command is standard or binary and reads accordingly, then returns the packet.
>
> > **Parameters**
> > **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
> >
> > **Raises**
> > **Exception** – Cannot read an invalid command.
> >
> > **Returns**
> > POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).
> >
> > **Return type**
> > *Packet*|Packet_Standard|Packet_BinaryStandard

_Read_Binary(*prePacket: bytes*, *validateChecksum: bool = True*) → *PacketBinary*

Reads the remaining part of the variable-length binary packet. It first reads the standard packet (prePacket+payload+checksum+ETX). Then it determines how long the binary packet is from the payload of the standard POD packet and reads that many bytes. It then reads to ETX to get the checksum+ETX.

> **Parameters**
>
> - **prePacket** (*bytes*) – Bytes string containing the beginning of a POD packet: STX (1 byte) + command number (4 bytes)
>
> - **validateChecksum** (*bool, optional*) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
>
> **Raises**
> **Exception** – An exception is raised if the checksum is invalid (only if validateChecksum=True).
>
> **Returns**
> Variable-length binary POD packet.
>
> **Return type**
> Packet_BinaryStandard

_Read_GetCommand(*validateChecksum: bool = True*) → bytes

Reads one byte at a time up to 4 bytes to get the ASCII-encoded bytes command number. For each byte read, it can (1) start the recursion over if an STX is found, (2) returns if ETX is found, or (3) continue building the command number.

> **Parameters**
> **validateChecksum** (*bool, optional*) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
>
> **Returns**
> 4 byte long string containing the ASCII-encoded command number.
>
> **Return type**
> bytes

_Read_Standard(*prePacket: bytes*, *validateChecksum: bool = True*) → *PacketStandard*

Reads the payload, checksum, and ETX. Then it builds the complete standard POD packet in bytes.

> **Parameters**
>
> - **prePacket** (*bytes*) – Bytes string containing the beginning of a POD packet: STX (1 byte) + command number (4 bytes).
>
> - **validateChecksum** (*bool, optional*) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
>
> **Raises**
> **Exception** – An exception is raised if the checksum is invalid (only if validateChecksum=True).
>
> **Returns**
> Complete standard POD packet.
>
> **Return type**
> Packet_Standard

_Read_ToETX(*validateChecksum: bool = True*) → bytes

Reads one byte at a time until an ETX is found. It will restart the recursive read if an STX is found anywhere.

Parameters
> **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.

Returns
> Bytes string ending with ETX.

Return type
> bytes

static _ValidateChecksum(*msg: bytes*) → bool

Validates the checksum of a given POD packet. The checksum is valid if the calculated checksum from the data matches the checksum written in the packet.

Parameters
> **msg** (`bytes`) – Bytes message containing a POD packet: STX (1 bytes) + data (? bytes) + checksum (2 bytes) + ETX (1 byte).

Returns
> True if the checksum is correct, false otherwise.

Return type
> bool

Raises
> **Exception** – msg does not begin with STX or end with ETX.

PodApi.Devices.PodDevice_8206HR module

class PodApi.Devices.PodDevice_8206HR.**Pod8206HR**(*port: str | int*, *preampGain: int*, *baudrate: int = 9600*)

Bases: *[Pod](#)*

POD_8206HR handles communication using an 8206HR POD device.

_preampGain

Instance-level integer (10 or 100) preamplifier gain.

Type
> int

ReadPODpacket(*validateChecksum: bool = True*, *timeout_sec: int | float = 5*) → *[Packet](#)*

Reads a complete POD packet, either in standard or binary format, beginning with STX and ending with ETX. Reads first STX and then starts recursion.

Parameters

- **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.

- **timeout_sec** (`int|float, optional`) – Time in seconds to wait for serial data. Defaults to 5.

Returns
> POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).

Return type
> *[Packet](#)*

_Read_Binary(*prePacket: bytes*, *validateChecksum: bool = True*) → *PacketBinary4*

> After receiving the prePacket, it reads the 8 bytes(TTL+channels) and then reads to ETX (checksum+ETX).

> > **Parameters**
> >
> > - **prePacket** (`bytes`) – Bytes string containing the beginning of a POD packet: STX (1 byte) + command number (4 bytes).
> >
> > - **validateChecksum** (`bool,` `optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
> >
> > **Raises**
> > **Exception** – Bad checksum for binary POD packet read.
> >
> > **Returns**
> > Binary4 POD packet.
> >
> > **Return type**
> > Packet_Binary4

static _TranslateTTLbyte_ASCII(*ttlByte: bytes*) → dict[str, int]

> Separates the bits of each TTL (0-3) from a ASCII encoded byte.

> > **Parameters**
> > **ttlByte** (`bytes`) – One byte string for the TTL (ASCII encoded).
> >
> > **Returns**
> > Dictionary of the TTLs. Values are 1 when input, 0 when output.
> >
> > **Return type**
> > dict[str,int]

PodApi.Devices.PodDevice_8229 module

class PodApi.Devices.PodDevice_8229.**Pod8229**(*port: str | int*, *baudrate: int = 19200*)

> Bases: *Pod*

> POD_8229 handles communication using an 8229 POD device.

> **static BuildSetDayScheduleArgument**(*day: str | int*, *hours: list | tuple[bool | int]*, *speed: int | list | tuple[int]*) → tuple[int]

> > Appends the day of the week code to the front of the encoded hourly schedule. this tuple is formatted to be used as the #141 'SET DAY SCHEDULE' argument.

> > > **Parameters**
> > >
> > > - **day** (`str` `|` `int`) – Day of the week. Can be either the name of the day (i.e. Sunday, Monday, etc.) or the 0-6 day code (0 for Sunday increacing to 6 for Saturday).
> > >
> > > - **hours** (`list` `|` `tuple[bool` `|` `int]`) – Array of 24 items. The value is 1 for motor on and 0 for motor off.
> > >
> > > - **speed** (`int` `|` `list` `|` `tuple[int]`) – Speed of the motor (0-100). This is an integer of all hours are the same speed. If there are multiple speeds, this should be an array of 24 items.
> > >
> > > **Returns**
> > > _description_
> > >
> > > **Return type**
> > > tuple[int]

static CodeDayOfWeek(*day: str*) → int

Converts the day of the week to an integer code understandable by the POD device. The day is determined by the first 1-2 characters of the string, which supports multiple abbreviations for days of the week.

> **Parameters**
>> **day** (`str`) – Day of the week.
>
> **Raises**
>> **Exception** – Invalid day of the week.
>
> **Returns**
>> Code for the day of the week. Values are 0-6, with 0 for Saturday, 1 for Monday, …, and 6 for Saturday.
>
> **Return type**
>> int

static CodeDaySchedule(*hours: list | tuple[bool | int]*, *speed: int | list | tuple[int]*) → list[int]

Bitmasks the day schedule to encode the motor on/off status and the motor speed. Use this for getting the command #141 'SET DAY SCHEDULE' U8x24 argument component.

> **Parameters**
>> - **hours** (`list | tuple[bool | int]`) – Array of 24 items. The value is 1 for motor on and 0 for motor off.
>>
>> - **speed** (`int | list | tuple[int]`) – Speed of the motor (0-100). This is an integer of all hours are the same speed. If there are multiple speeds, this should be an array of 24 items.
>
> **Returns**
>> List of 24 integer items. The msb is the motor on/off flag and the remaining 7 bits are the speed.
>
> **Return type**
>> list[int]

static DecodeDayAndSchedule(*dayschedule: bytes*)

static DecodeDayOfWeek(*day: int*) → str

Converts the integer code for a day of the week to a human-readable string.

> **Parameters**
>> **day** (`int`) – Day of the week code must be 0-6.
>
> **Returns**
>> Day of the week ('Sunday', 'Monday', etc.).
>
> **Return type**
>> str

static DecodeDaySchedule(*schedule: bytes*) → dict[str, int | tuple[int]]

Interprets the return bytes from the command #142 'GET DAY SCHEDULE'.

> **Parameters**
>> **schedule** (`bytes`) – 24 byte long bitstring with one U8 per hour in a day.
>
> **Returns**
>> Dictionary with 'Hour' as a tuple of 24 0/1 values (0 is motor off and 1 is motor on) and 'Speed' as the motor speed (0-100). If the motor speed is the same every hour, 'Speed' will be an integer; otherwise, 'Speed' will be a tuple of 24 items.

> > **Return type**
> > dict[str,int|tuple[int]]

static **DecodeLCDSchedule**(*schedule: bytes*) → dict[str, str | list[int]]

> Interprets the return bytes from the command #202 'LCD SET DAY SCHEDULE'.

> > **Parameters**
> > **schedule** (`bytes`) – 4 Byte long bitstring. Byte 3 is weekday, Byte 2 is hours 0-7, Byte 1 is hours 8-15, and byte 0 is hours 16-23.

> > **Returns**
> > Dictionary with Day as the day of the week, and Hours containing a list of 24 0/1 values (one for each hour). Each bit represents the motor state in that hour, 1 for on and 0 for off.

> > **Return type**
> > dict[str,int|list[int]]

static **GetCurrentTime**() → tuple[int]

> Gets a tuple to use as the argument for command #140 SET TIME containing values for the current time.

> > **Returns**
> > Tuple of 7 integer values. The format is (Seconds, Minutes, Hours, Day, Month, Year (without century, so 23 for 2023), Weekday (0 for Sunday))

> > **Return type**
> > tuple[int]

ReadPODpacket(*validateChecksum: bool = True*, *timeout_sec: int | float = 5*) → *PacketStandard*

> Reads a complete POD packet, either in standard or binary format, beginning with STX and ending with ETX. Reads first STX and then starts recursion.

> > **Parameters**
> > - **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
> > - **timeout_sec** (`int|float, optional`) – Time in seconds to wait for serial data. Defaults to 5.

> > **Returns**
> > POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).

> > **Return type**
> > *Packet*

WritePacket(*cmd: str | int*, *payload: int | bytes | tuple[int | bytes] | None = None*) → bytes

> Builds a POD packet and writes it to the POD device.

> > **Parameters**
> > - **cmd** (`str | int`) – Command number.
> > - **payload** (`int | bytes | tuple[int | bytes], optional`) – None when there is no payload. If there is a payload, set to an integer value, bytes string, or tuple. Defaults to None.

> > **Returns**
> > Bytes string that was written to the POD device.

> > **Return type**
> > bytes

static **_CodeDecimalAsHex**(*val: int*) → int

> Builds an integer that equals the val argument when converted into hexadecimal. All integers are converted to hexadecimal ASCII encoded bytes. Some commands (i.e. 8229 #140) need decimal ASCII encoded bytes; to do this, give the return value of _CodeDecimalAsHex() as the payload. Example: I want a number that is equal to 16 in hex. 1*16^1 + 6*16^0 = 22. Calling _CodeDecimalAsHex(16) will return 22.

> > **Parameters**
> > > **val** (`int`) – Unsigned integer number.
> >
> > **Returns**
> > > integer that equals the val argument when converted into hexadecimal.
> >
> > **Return type**
> > > int

static **_Custom140SETTIME**(*payload: tuple[int]*) → tuple[int]

> Custom function to translate the payload for command #140 SET TIME.

> > **Parameters**
> > > **payload** (`tuple[int]`) – Default translated payload.
> >
> > **Returns**
> > > Tuple of times.
> >
> > **Return type**
> > > tuple[int]

static **_DecodeDecimalAsHex**(*val: int*) → int

> Interprets an integer that was converted to a hexadecimal representation of a decimal value. In other words, this method reverses _CodeDecimalAsHex().

> > **Parameters**
> > > **val** (`int`) – Unsigned integer that was converted to a hexadecimal representation of a decimal value.
> >
> > **Returns**
> > > Unsigned integer as a true decimal number.
> >
> > **Return type**
> > > int

static **_Validate_Day**(*day: str | int*) → int

> Raises an exception if the day is incorrectly formatted. If the day is given as a string, it will be converted to its integer code.

> > **Parameters**
> > > **day** (`str | int`) – String day of the week or its repsective integer code.
> >
> > **Raises**
> > > - **Exception** – The day integer argument must be 0-6.
> > > - **Exception** – The day argument must be a str or int.
> >
> > **Returns**
> > > Integer code representing a day of the week.
> >
> > **Return type**
> > > int

static **_Validate_Hours**(*hours: list | tuple[bool | int]*) → list[bool | int]

> Raises an exception if the hours is incorrectly formatted. Converts the hours into a list before returning it.

> **Parameters**
>> **hours** (`list | tuple[bool | int]`) – Array with 24 items with values of 1/0 representing each hour
>
> **Raises**
>> - **Exception** – The hours argument must be a list or tuple.
>>
>> - **Exception** – The hours argument must have exactly 24 items.
>>
>> - **Exception** – The hours items must be 0 or 1.
>
> **Returns**
>> List with 24 items for each hour. The values are 1/0.
>
> **Return type**
>> list[bool|int]

static **_Validate_Schedule**(*schedule: bytes*, *size: int*) → bytes

> Raises an exception if the schedule is incorrectly formatted
>
> **Parameters**
>> - **schedule** (`bytes`) – Bytes string containing the day schedule.
>>
>> - **size** (`int`) – Number of U8 bytes.
>
> **Raises**
>> - **Exception** – The schedule must be bytes.
>>
>> - **Exception** – The schedule is the incorrect size
>
> **Returns**
>> Same as the schedule argument.
>
> **Return type**
>> bytes

static **_Validate_Speed**(*speed: int | list | tuple[int]*) → list[int]

> Raises an exception if the speed is incorrectly formatted. If an integer speed is given, it will convert it to a list.
>
> **Parameters**
>> **speed** (`int | list | tuple[int]`) – Motor speed (0-100). This can either be an integer or a tuple/list of 24 speeds.
>
> **Raises**
>> - **Exception** – The speed argument must be an int, list, or tuple.
>>
>> - **Exception** – The speed must be between 0 and 100.
>>
>> - **Exception** – The speed argument must have exactly 24 items as a list/tuple.
>>
>> - **Exception** – The speed must be between 0 and 100 for every list/tuple item.
>
> **Returns**
>> List containing 24 motor speeds.
>
> **Return type**
>> list[int]

PodApi.Devices.PodDevice_8401HR module

class PodApi.Devices.PodDevice_8401HR.**Pod8401HR**(*port: str | int*, *ssGain: tuple | list | dict[str, int | None]* *= {'A': None, 'B': None, 'C': None, 'D': None}*, *preampGain: tuple | list | dict[str, int | None] = {'A': None, 'B': None, 'C': None, 'D': None}*, *baudrate: int = 9600*)

> Bases: *Pod*

> POD_8401HR handles communication using an 8401-HR POD device.

> **_ssGain**

> > Instance-level dictionary storing the second-stage gain for all four channels.

> > **Type**

> > > dict[str,int|None]

> **_preampGain**

> > Instance-level dictionary storing the pramplifier gain for all four channels.

> > **Type**

> > > dict[str,int|None]

> static **CalculateBiasDAC_GetDACValue**(*vout: int | float*) → int

> > Calculates the DAC value given the output voltage. Used for 'GET/SET BIAS' commands.

> > **Parameters**

> > > **vout** (`int | float`) – Output voltage (+/- 2.048 V).

> > **Returns**

> > > Integer of the DAC value (16 bit 2's complement).

> > **Return type**

> > > int

> static **CalculateBiasDAC_GetVout**(*value: int*) → float

> > Calculates the output voltage given the DAC value. Used for 'GET/SET BIAS' commands.

> > **Parameters**

> > > **value** (`int`) – DAC value (16 bit 2's complement).

> > **Returns**

> > > Float of the output bias voltage [V].

> > **Return type**

> > > float

> static **DecodeChannelBitmask**(*channels: bytes*) → dict[str, int]

> > Converts the channel bitmask byte to a dictionary with each channel value. Use for 'GET INPUT GROUND' command payloads.

> > **Parameters**

> > > **channels** (`bytes`) – U8 byte containing the channel configuration.

> > **Returns**

> > > Dictionary with the channels as keys and values as the state. 0=Grounded and 1=Connected to Preamp.

> > **Return type**

> > > dict[str,int]

---

static **DecodeSSConfigBitmask**(*config: bytes*)

> Converts the SS configuration byte to a dictionary with the high-pass and gain. Use for 'GET SS CONFIG' command payloads.
>
> > **Parameters**
> > > **config** (*bytes*) – U8 byte containing the SS configurtation. Bit 0 = 0 for 0.5Hz Highpass, 1 for DC Highpass. Bit 1 = 0 for 5x gain, 1 for 1x gain.

static **DecodeTTLByte**(*ttlByte: bytes*) → dict[str, int]

> Converts the TTL bytes argument into a dictionary of integer TTL values.
>
> > **Parameters**
> > > **ttlByte** (*bytes*) – U8 byte containing the TTL bitmask.
> >
> > **Returns**
> > > Dictinoary with TTL name keys and integer TTL values.
> >
> > **Return type**
> > > dict[str,int]

static **DecodeTTLPayload**(*payload: bytes*) → tuple[dict[str, int]]

> Decodes a paylaod with the two TTL bytes.
>
> > **Parameters**
> > > **payload** (*bytes*) – Bytes string of the POD packet payload.
> >
> > **Returns**
> > > Tuple with two TTL dictionaries.
> >
> > **Return type**
> > > tuple[dict[str, int]]

static **GetChannelBitmask**(*a: bool*, *b: bool*, *c: bool*, *d: bool*) → int

> Gets a bitmask, represented by an unsigned integer, used for 'SET INPUT GROUND' command.
>
> > **Parameters**
> > > - **a** (*bool*) – State for channel A, 0=Grounded and 1=Connected to Preamp.
> > > - **b** (*bool*) – State for channel B, 0=Grounded and 1=Connected to Preamp.
> > > - **c** (*bool*) – State for channel C, 0=Grounded and 1=Connected to Preamp.
> > > - **d** (*bool*) – State for channel D, 0=Grounded and 1=Connected to Preamp.
> >
> > **Returns**
> > > Integer representing a bitmask.
> >
> > **Return type**
> > > int

static **GetChannelMapForPreampDevice**(*preampName: str*) → dict[str, str] | None

> Get the channel mapping (channel labels for A,B,C,D) for a given device.
>
> > **Parameters**
> > > **preampName** (*str*) – String for the device/sensor name.
> >
> > **Returns**
> > > Dictionary with keys A,B,C,D with values of the channel names. Returns None if the device name does not exist.
> >
> > **Return type**
> > > dict[str,str]|None

static GetSSConfigBitmask(*gain: int*, *highpass: float*) → int

> Gets a bitmask, represented by an unsigned integer, used for 'SET SS CONFIG' command.
>
> > **Parameters**
> >
> > - **gain** (`int`) – 1 for 1x gain. else for 5x gain.
> >
> > - **highpass** (`float`) – 0 for DC highpass, else for 0.5Hz highpass.
> >
> > **Returns**
> >   Integer representing a bitmask.
> >
> > **Return type**
> >   int

static GetSupportedPreampDevices() → list[str]

> Gets a list of device/sensor names used for channel mapping.
>
> > **Returns**
> >   List of string names of all supported sensors.
> >
> > **Return type**
> >   list[str]

static GetTTLbitmask(*ext0: bool = 0*, *ext1: bool = 0*, *ttl4: bool = 0*, *ttl3: bool = 0*, *ttl2: bool = 0*, *ttl1: bool = 0*) → int

> Builds an integer, which represents a binary mask, that can be used for TTL command arguments.
>
> > **Parameters**
> >
> > - **ext0** (`bool, optional`) – boolean bit for ext0. Defaults to 0.
> >
> > - **ext1** (`bool, optional`) – boolean bit for ext1. Defaults to 0.
> >
> > - **ttl4** (`bool, optional`) – boolean bit for ttl4. Defaults to 0.
> >
> > - **ttl3** (`bool, optional`) – boolean bit for ttl3. Defaults to 0.
> >
> > - **ttl2** (`bool, optional`) – boolean bit for ttl2. Defaults to 0.
> >
> > - **ttl1** (`bool, optional`) – boolean bit for ttl1. Defaults to 0.
> >
> > **Returns**
> >   Integer number to be used as a bit mask.
> >
> > **Return type**
> >   int

static IsPreampDeviceSupported(*name: str*) → bool

> Checks if the argument exists in channel map for all preamp sensors.
>
> > **Parameters**
> >   **name** (`str`) – name of the device
> >
> > **Returns**
> >   True if the name exists in __CHANNELMAPALL, false otherwise.
> >
> > **Return type**
> >   bool

ReadPODpacket(*validateChecksum: bool = True*, *timeout_sec: int | float = 5*) → *Packet*

> Reads a complete POD packet, either in standard or binary format, beginning with STX and ending with ETX. Reads first STX and then starts recursion.
>
> > **Parameters**

- **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.

- **timeout_sec** (`int|float, optional`) – Time in seconds to wait for serial data. Defaults to 5.

> **Returns**
> POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).
>
> **Return type**
> *[Packet](#)*

WritePacket(*cmd: str | int*, *payload: int | bytes | tuple[int | bytes] | None = None*) → *[PacketStandard](#)*

> Builds a POD packet and writes it to the POD device.
>
> **Parameters**
>
> - **cmd** (`str | int`) – Command number.
>
> - **payload** (`int | bytes | tuple[int | bytes], optional`) – None when there is no payload. If there is a payload, set to an integer value, bytes string, or tuple. Defaults to None.
>
> **Returns**
> Packet that was written to the POD device.
>
> **Return type**
> Packet_Standard

static _FixABCDtype(*info: tuple | list | dict*, *thisIs: str = ''*) → dict

> Converts the info argument into a dictionary with A, B, C, and D as keys.
>
> **Parameters**
>
> - **info** (`tuple | list | dict`) – Variable to be converted into a dictionary.
>
> - **thisIs** (`str, optional`) – Description of the info argument, which is used in Exception statements. Defaults to ''.
>
> **Raises**
>
> - **Exception** – The dictionary has improper keys; keys must be ['A','B','C','D'].
>
> - **Exception** – The argument must have only four values.
>
> - **Exception** – The argument must be a tuple, list, or dict.
>
> **Returns**
> The info argument converted to a dictionary with A, B, C, and D as keys.
>
> **Return type**
> dict

_Read_Binary(*prePacket: bytes*, *validateChecksum: bool = True*)

> After receiving the prePacket, it reads the 23 bytes (binary data) and then reads to ETX.
>
> **Parameters**
>
> - **prePacket** (`bytes`) – Bytes string containing the beginning of a POD packet: STX (1 byte) + command number (4 bytes).
>
> - **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.

> **Raises**
>> **Exception** – Bad checksum for binary POD packet read.

static **_ValidatePreampGain**(*preampGain: dict*) → None

> Checks that the preamplifier gain dictionary has proper values (10, 100, or None).

>> **Parameters**
>>> **preampGain** (`dict`) – preamplifier gain dictionary.

>> **Raises**
>>> **Exception** – EEG/EMG preampGain must be 10 or 100. For biosensors, the preampGain is None.

static **_ValidateSsGain**(*ssgain: dict*)

> Checks that the second stage gain dictionary has proper values (1, 5, or None).

>> **Parameters**
>>> **ssgain** (`dict`) – Second stage gain dictionary.

>> **Raises**
>>> **Exception** – The ssGain must be 1 or 5; set ssGain to None if no-connect.

__CHANNELMAPALL: dict[str, dict[str, str]] = {'8406-2BIO': {'A': 'Bio1', 'B': 'Bio2', 'C': 'NC', 'D': 'NC'}, '8406-BIO': {'A': 'Bio', 'B': 'NC', 'C': 'NC', 'D': 'NC'}, '8406-EEG2BIO': {'A': 'Bio1', 'B': 'EEG1', 'C': 'EMG', 'D': 'Bio2'}, '8406-SE': {'A': 'Bio', 'B': 'EEG1', 'C': 'EMG', 'D': 'EEG2'}, '8406-SE3': {'A': 'Bio', 'B': 'EEG1', 'C': 'EEG3', 'D': 'EEG2'}, '8406-SE31M': {'A': 'EMG', 'B': 'EEG1', 'C': 'EEG3', 'D': 'EEG2'}, '8406-SE4': {'A': 'EEG4', 'B': 'EEG1', 'C': 'EEG3', 'D': 'EEG2'}, '8406-SL': {'A': 'Bio', 'B': 'EEG1', 'C': 'EMG', 'D': 'EEG2'}, '8407-SE': {'A': 'Bio', 'B': 'EEG1', 'C': 'EMG', 'D': 'EEG2'}, '8407-SE-2BIO': {'A': 'Bio1', 'B': 'Bio2', 'C': 'EMG', 'D': 'EEG2'}, '8407-SE3': {'A': 'Bio', 'B': 'EEG1', 'C': 'EEG3', 'D': 'EEG2'}, '8407-SE31M': {'A': 'EEG3', 'B': 'EEG1', 'C': 'EMG', 'D': 'EEG2'}, '8407-SE4': {'A': 'EEG4', 'B': 'EEG1', 'C': 'EEG3', 'D': 'EEG2'}, '8407-SL': {'A': 'Bio', 'B': 'EEG1', 'C': 'EMG', 'D': 'EEG2'}, '8407-SL-2BIO': {'A': 'Bio1', 'B': 'Bio2', 'C': 'EMG', 'D': 'EEG2'}}

> Class-level dictionary containing the channel map for all preamplifier devices.

PodApi.Devices.PodDevice_8480SC module

class PodApi.Devices.PodDevice_8480SC.**Pod8480SC**(*port: str | int, baudrate: int = 9600*)

> Bases: [*Pod*](#)

> POD_8480SC handles communication using an 8480-SC POD device.

> static **DecodeStimulusConfigBits**(*config: int*) → dict

>> Converts an integer into 3 values, representing 3 individual bits of the Stimulus Config Bits.

>>> **Parameters**
>>>> **config** (`int`) – an integer is passed in, and it represents the Config Flag byte.

>>> **Returns**
>>>> Keys as the names of the bits, the values representing values at each bit.

>>> **Return type**
>>>> dict

static DecodeSyncConfigBits(*config: int*) → dict

> Converts an integer into 3 values, representing 3 individual bits of the Sync Config Bits.
>
> > **Parameters**
> > > **config** (`int`) – an integer is passed in, and it represents the Sync Config Flag byte.
> >
> > **Returns**
> > > Keys as the names of the bits, the values representing values at each bit.
> >
> > **Return type**
> > > dict

static DecodeTTlConfigBits(*config: int*) → dict

> Converts an interger into 3 values, representing 3 individual bits of the TTL Config Bits.
>
> > **Parameters**
> > > **config** (`int`) – an integer is passed in, and it represents the TTL Setup Config Flag Byte.
> >
> > **Returns**
> > > Keys as the names of the bits, the values representing values at each bit.
> >
> > **Return type**
> > > dict

ReadPODpacket(*validateChecksum: bool = True*, *timeout_sec: int | float = 5*) → *PacketStandard*

> Reads a complete POD packet, either in standard or binary format, beginning with STX and ending with ETX. Reads first STX and then starts recursion.
>
> > **Parameters**
> > > - **validateChecksum** (`bool, optional`) – Set to True to validate the checksum. Set to False to skip validation. Defaults to True.
> > > - **timeout_sec** (`int|float, optional`) – Time in seconds to wait for serial data. Defaults to 5.
> >
> > **Returns**
> > > POD packet beginning with STX and ending with ETX. This may be a standard packet, binary packet, or an unformatted packet (STX+something+ETX).
> >
> > **Return type**
> > > *Packet*

static StimulusConfigBits(*optoElec: bool*, *monoBiphasic: bool*, *Simul: bool*) → int

> Incoming inputs are bitmasked into an integer value. This value is later given as part of a payload to command #102 'SET STIMULUS'.
>
> > **Parameters**
> > > - **optoElec** (`bool`) – Bit is Opto/Electrical.
> > > - **monoBiphasic** (`bool`) – Bit 1 is Monophasic/Biphasic.
> > > - **Simul** (`bool`) – Bit 2 is Simultaneous.
> >
> > **Returns**
> > > which represents the Config flag byte in the Stimulus Command. The return value is the seventh item in the payload for command 'SET STIMULUS'.
> >
> > **Return type**
> > > int

static SyncConfigBits(*sync_level: bool*, *sync_idle: bool*, *signal_trigger: bool*) → int

Incoming inputs are bitmasked into an integer value. This value is later given as the payload to command #127 'SET SYNC CONFIG'.

> **Parameters**
>
> - **sync_level** (`bool`) – Bit 0 is Sync Level.
>
> - **sync_idle** (`bool`) – Bit 1 is Stimulus Triggering.
>
> - **signal_trigger** (`bool`) – Bit 2 is Signal/Trigger.
>
> **Returns**
> which represents the Sync Config Bits format value.
>
> **Return type**
> int

static TtlConfigBits(*trigger: bool*, *stimtrig: bool*, *input_sync: bool*) → int

Incoming inputs are bitmasked into an integer value. This value is later given as part of the payload to command #109 'SET TTL SETUP'. This commands accepts 3 items in the payload, and the return value of this function is given as the second item.

> **Parameters**
>
> - **trigger** (`bool`) – Bit 0 is 0 for rising edge, 1 for falling edge.
>
> - **stimtrig** (`bool`) – Bit 1 is 0 for TTL event notifications, 1 for TTL inputs as triggers.
>
> - **input_sync** (`bool`) – Bit 7 is 0 for normal TTL operation, 1 for TTL pin operates as a sync output.
>
> **Returns**
> which represents the TTL Config Bits Format value.
>
> **Return type**
> int

WritePacket(*cmd: str | int*, *payload: int | bytes | tuple[int | bytes] | None = None*) → *PacketStandard*

Builds a POD packet and writes it to the POD device.

> **Parameters**
>
> - **cmd** (`str | int`) – Command number.
>
> - **payload** (`int | bytes | tuple[int | bytes], optional`) – None when there is no payload. If there is a payload, set to an integer value, bytes string, or tuple. Defaults to None.
>
> **Returns**
> Packet that was written to the POD device.
>
> **Return type**
> Packet_Standard

static _Custom108GETTTLSETUP(*payload: bytes*) → tuple[int | dict]

Custom function to translate the TTL setup for command #108 GET TTL SETUP.

> **Parameters**
> **payload** (`bytes`) – Bytes string of the POD packet payload.
>
> **Returns**
> Tuple of the TTL setup.

> **Return type**
>> tuple[int|dict]

static **_Custom109SETTTLSETUP**(*payload: bytes*) → tuple[int | dict]

> Custom function to translate the TTL setup for command #109 SET TTL SETUP.

>> **Parameters**
>>> **payload** (`bytes`) – Bytes string of the POD packet payload.

>> **Returns**
>>> Tuple of the TTL setup.

>> **Return type**
>>> tuple[int|dict]

static **_CustomSTIMULUS**(*payload: bytes*, *defaultPayload: tuple*) → tuple

> _summary_

>> **Parameters**

>>> • **payload** (`bytes`) – Bytes string of the POD packet payload.

>>> • **defaultPayload** (`tuple`) – Default translated payload.

>> **Returns**
>>> Tuple of the translated stimulus payload.

>> **Return type**
>>> tuple

static **_CustomSYNCCONFIG**(*payload: bytes*) → dict

> Custom function to translate the sync config.

>> **Parameters**
>>> **payload** (`bytes`) – Bytes string of the POD packet payload.

>> **Returns**
>>> Keys as the names of the bits, the values representing values at each bit.

>> **Return type**
>>> dict

Module contents

1.1.1.3 PodApi.Packets package

Submodules

PodApi.Packets.Binary module

class PodApi.Packets.Binary.**PacketBinary**(*pkt: bytes*, *commands:* CommandSet *| None = None*)

> Bases: `Packet`

> Container class that stores a standard binary command packet for a POD device. The format is STX (1 byte) + command number (4 bytes) + length of binary (4 bytes) + checksum (2 bytes) + ETX (1 bytes) + binary (LENGTH bytes) + checksum (2 bytes) + ETX (1 bytes)

binaryLength

Number of bytes of binary data from the packet.

> **Type**
> bytes

binaryData

Variable length binary datafrom the packet.

> **Type**
> bytes

BinaryLength() → int

Translate the binary ASCII encoding of the binary data length into a readable integer

> **Returns**
> Integer of the binary data length.
>
> **Return type**
> int

static CheckIfPacketIsValid(*msg: bytes*)

Raises an Exception if the packet is incorrectly formatted.

> **Parameters**
> **msg** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
>
> **Raises**
>
> - **Exception** – Packet is too small to be a standard packet.
>
> - **Exception** – A standard binary packet must have an ETX before the binary bytes.

GetBinaryData() → bytes

Gets the binary data from a POD packet.

> **Parameters**
> **pkt** (*bytes*) – Bytes string containing a POD packet.
>
> **Returns**
> Bytes string containg binary data.
>
> **Return type**
> bytes

static GetBinaryLength(*pkt: bytes*) → bytes

Gets the length, or number of bytes, of the binary data in a POD packet.

> **Parameters**
> **pkt** (*bytes*) – Bytes string containing a POD packet.
>
> **Returns**
> Bytes string of the length of the binary data.
>
> **Return type**
> bytes

static GetMinimumLength() → int

Gets the number of bytes in the smallest possible packet; STX (1 byte) + something + ETX (1 byte).

> **Returns**
>> integer representing the minimum length of a binary POD command packet. Format is STX (1 byte) + command number (4 bytes) + length of binary (4 bytes) + checksum (2 bytes) + ETX (1 bytes) + binary (LENGTH bytes) + checksum (2 bytes) + ETX (1 bytes)
>
> **Return type**
>> int

TranslateAll() → dict[str, Any]

> Builds a dictionary containing all parts of the POD packet in readable values.
>
> **Returns**
>> Dictionary with the command number, binary packet length, and binary data.
>
> **Return type**
>> dict[str,Any]

UnpackAll() → dict[str, bytes]

> Builds a dictionary containing all parts of the POD packet in bytes.
>
> **Returns**
>> Dictionary with the command number, binary packet length, and binary data.
>
> **Return type**
>> dict[str,bytes]

PodApi.Packets.Binary4 module

class PodApi.Packets.Binary4.**PacketBinary4**(*pkt: bytes*, *preampGain: int*, *commands:* CommandSet | *None = None*)

Bases: *Packet*

Container class that stores a binary4 command packet for a POD device. The format is STX (1 byte) + command (4 bytes) + packet number (1 byte) + TTL (1 byte) + CH0 (2 bytes) + CH1 (2 bytes) + CH2 (2 bytes) + checksum (2 bytes) + ETX (1 byte).

_preampGain

> Preamplifier gain. This should be 10 or 100 for an 8206-HR device.
>
> **Type**
>> int

packetNumber

> Packet number for this POD packet.
>
> **Type**
>> bytes

ttl

> TTL data for this packet.
>
> **Type**
>> bytes

ch0

> channel 0 data for this packet.
>
> **Type**
>> bytes

ch1

channel 1 data for this packet.

> **Type**
> bytes

ch2

channel 2 data for this packet.

> **Type**
> bytes

static **BinaryBytesToVoltage**(*value: bytes*, *preampGain: int*) → float

Converts a binary bytes value read from POD device and converts it to the real voltage value at the pream-plifier input.

> **Parameters**
> **value** (*bytes*) – Bytes string containing voltage measurement.

> **Returns**
> A number containing the voltage in Volts [V].

> **Return type**
> float

Ch(*n: int*) → float

Translates the binary channel n bytes into a voltage.

> **Parameters**
> **n** (*int*) – Channel number. Should be 0, 1, or 2.

> **Raises**
> **Exception** – Channel does not exist.

> **Returns**
> Voltage of channel n in Volts.

> **Return type**
> float

static **CheckIfPacketIsValid**(*msg: bytes*)

Raises an Exception if the packet is incorrectly formatted.

> **Parameters**
> **msg** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.

> **Raises**
> **Exception** – Packet the wrong size to be a binary4 packet.

static **GetBinaryLength**() → int

Gets the number of bytes of binary data in a binary4 packet.

> **Returns**
> Integer representing the number of binary encoded bytes in a binary4 packet.

> **Return type**
> int

static **GetCh**(*n: int*, *pkt: bytes*) → bytes

Gets the channel n bytes from a POD packet.

> **Parameters**
>
> - **n** (`int`) – Channel number. Should be 0, 1, or 2.
>
> - **pkt** (`bytes`) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
>
> **Returns**
> Bytes string of the channel 0 data.
>
> **Return type**
> bytes
>
> **Returns**
> Channel does not exist.
>
> **Return type**
> bytes

static GetMinimumLength() → int

> Gets the number of bytes in the smallest possible binary4 packet; STX (1 byte) + command (4 bytes) + packet number (1 byte) + TTL (1 byte) + CH0 (2 bytes) + CH1 (2 bytes) + CH2 (2 bytes) + checksum (2 bytes) + ETX (1 byte).
>
> **Returns**
> Integer representing the minimum length of a binary4 POD packet.
>
> **Return type**
> int

static GetPacketNumber(*pkt: bytes*) → bytes

> Gets the packet number in bytes from a POD packet.
>
> **Parameters**
> **pkt** (`bytes`) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
>
> **Returns**
> Bytes string of the packet number.
>
> **Return type**
> bytes

static GetTTL(*pkt: bytes*) → bytes

> Gets the TTL bytes from a POD packet
>
> **Parameters**
> **pkt** (`bytes`) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
>
> **Returns**
> Bytes string of the TTL data.
>
> **Return type**
> bytes

PacketNumber() → int

> Translates the binary packet number into a readable integer.
>
> **Returns**
> Integer of the packet number.

> **Return type**
>> int

TranslateAll() → dict[str, Any]

Builds a dictionary containing all parts of the POD packet in readable values.

> **Returns**
>> Dictionary with the command number, packet number, TTL and channels 0, 1, and 2.

> **Return type**
>> dict[str, Any]

static TranslateBinaryTTLbyte(*ttlByte: bytes*) → dict[str, int]

Separates the bits of each TTL (0-3) from a binary encoded byte.

> **Parameters**
>> **ttlByte** (*bytes*) – One byte string for the TTL (binary encoded).

> **Returns**
>> Dictionary of the TTLs. Values are 1 when input, 0 when output.

> **Return type**
>> dict[str,int]

Ttl() → dict[str, int]

Translates the binary TTL bytes into a dictionary containing each TTL value.

> **Returns**
>> Dictionary with TTL name keys and TTL data as values.

> **Return type**
>> dict[str,int]

UnpackAll() → dict[str, bytes]

Builds a dictionary containing all parts of the POD packet in bytes.

> **Returns**
>> Dictionary with the command number, packet number, TTL and channels 0, 1, and 2.

> **Return type**
>> dict[str,bytes]

PodApi.Packets.Binary5 module

class PodApi.Packets.Binary5.**PacketBinary5**(*pkt: bytes*, *ssGain: dict[str, int | None] = {'A': None, 'B': None, 'C': None, 'D': None}*, *preampGain: dict[str, int | None] = {'A': None, 'B': None, 'C': None, 'D': None}*, *commands:* CommandSet *| None = None*)

Bases: *Packet*

Container class that stores a binary5 command packet for a POD device. The format is STX (1 byte) + command (4 bytes) + packet number (1 byte) + status (1 byte) + channels (9 bytes) + AEXT0 (2 bytes) + AEXT1 (2 bytes) + ATTL1 (2 bytes) + ATTL2 (2 bytes) + ATTL3 (2 bytes) + ATTL4 (2 bytes) + checksum (2 bytes) + EXT (1 byte)

_ssGain

Dictionary with A, B, C, D keys and second stage gain values (1, 5, or None).

> **Type**
>> dict[str,int|None]

_preampGain

> Dictionary with A, B, C, D keys and preamplifier gain values (10, 100, or None).
>
> > **Type**
> > > dict[str,int|None]

packetNumber

> Packet number for this POD packet.
>
> > **Type**
> > > bytes

status

> Status for this POD packet.
>
> > **Type**
> > > bytes

channels

> channel A, B, C, and D data for this POD packet.
>
> > **Type**
> > > bytes

aEXT0

> Analog EXT0 data for this POD packet.
>
> > **Type**
> > > bytes

aEXT1

> Analog EXT1 data for this POD packet.
>
> > **Type**
> > > bytes

aTTL1

> Analog TTL1 data for this POD packet.
>
> > **Type**
> > > bytes

aTTL2

> Analog TTL2 data for this POD packet.
>
> > **Type**
> > > bytes

aTTL3

> Analog TTL3 data for this POD packet.
>
> > **Type**
> > > bytes

aTTL4

> Analog TTL4 data for this POD packet.
>
> > **Type**
> > > bytes

AnalogEXT(*n: int*) → float

    Translates the analog EXT value into a voltage.

        Parameters

            n (`int`) – Analog EXT number. Should be 0 or 1.

        Raises

            Exception – AEXT does not exist.

        Returns

            Analog EXT voltage in volts (V).

        Return type

            float

AnalogTTL(*n: int*) → float

    Translates the analog TTL value into a voltage.

        Parameters

            n (`int`) – Analog TTL number. Should be 1, 2, 3, or 4.

        Raises

            Exception – ATTL does not exist.

        Returns

            Analog TTL voltage in volts (v).

        Return type

            float

Channel(*c: str*) → float

    Translates the channel data into a voltage.

        Parameters

            c (`str`) – Channel character. Should be A, B, C, or D.

        Raises

            Exception – Channel does not exist.

        Returns

            Voltage of the channel in volts (V).

        Return type

            float

static CheckIfPacketIsValid(*msg: bytes*)

    Raises an Exception if the packet is incorrectly formatted.

        Parameters

            msg (`bytes`) – Bytes string containing a POD packet. Should begin with STX and end with ETX.

        Raises

            Exception – Packet the wrong size to be a binary5 packet.

static GetAnalogEXT(*n: int*, *pkt: bytes*) → bytes

    Gets the analog EXT from a POD packet.

        Parameters

            • **n** (`int`) – Analog EXT number. Should be 0 or 1.

- **pkt** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.

> **Raises**
> **Exception** – AEXT does not exist.

> **Returns**
> Bytes string of the AEXT.

> **Return type**
> bytes

static **GetAnalogTTL**(*n: int*, *pkt: bytes*) → bytes

> Gets the analog TTL from a POD packet.

> **Parameters**
>
> - **n** (*int*) – Analog TTL number. Should be 1, 2, 3, or 4.
>
> - **pkt** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.

> **Raises**
> **Exception** – ATTL does not exist.

> **Returns**
> Bytes string of the ATTL.

> **Return type**
> bytes

static **GetBinaryLength**() → int

> Gets the number of bytes of binary data in a binary5 packet.

> **Returns**
> Integer representing the number of binary encoded bytes in a binary5 packet.

> **Return type**
> int

static **GetChannels**(*pkt: bytes*) → bytes

> Gets the channel bytes for channels A, B, C, and D together from a POD packet.

> **Parameters**
> **pkt** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.

> **Returns**
> Bytes string of the channels A, B, C, and D together.

> **Return type**
> bytes

static **GetMinimumLength**() → int

> Gets the number of bytes in the smallest possible binary4 packet; STX (1 byte) + command (4 bytes) + packet number (1 byte) + status (1 byte) + channels (9 bytes) + AEXT0 (2 bytes) + AEXT1 (2 bytes) + ATTL1 (2 bytes) + ATTL2 (2 bytes) + ATTL3 (2 bytes) + ATTL4 (2 bytes) + checksum (2 bytes) + EXT (1 byte)

> **Returns**
> Integer representing the minimum length of a binary5 POD packet.

> **Return type**
> > int

static GetPacketNumber(*pkt: bytes*) → bytes

> Gets the packet number in bytes from a POD packet.
>
> > **Parameters**
> > > **pkt** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
> >
> > **Returns**
> > > Bytes string of the packet number.
> >
> > **Return type**
> > > bytes

static GetStatus(*pkt: bytes*) → bytes

> Gets the status value in bytes from a POD packet.
>
> > **Parameters**
> > > **pkt** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
> >
> > **Returns**
> > > Bytes string of the status.
> >
> > **Return type**
> > > bytes

PacketNumber() → int

> Translates the binary packet number into a readable integer.
>
> > **Returns**
> > > Integer of the packet number.
> >
> > **Return type**
> > > int

Status() → int

> Translates the binary status value into a readable integer
>
> > **Returns**
> > > Integer status value.
> >
> > **Return type**
> > > int

TranslateAll() → dict[str, Any]

> Builds a dictionary containing all parts of the POD packet in readable values.
>
> > **Returns**
> > > Dictionary with the command number, packet number, status, channels, analog EXT, and analog TTL.
> >
> > **Return type**
> > > dict[str, bytes]

UnpackAll() → dict[str, bytes]

> Builds a dictionary containing all parts of the POD packet in bytes.

> **Returns**
> Dictionary with the command number, packet number, status, channels, analog EXT, and analog TTL.
>
> **Return type**
> dict[str, bytes]

static **_Voltage_PrimaryChannels**(*value: int*, *ssGain: int | None = None*, *PreampGain: int | None = None*) → float

> Converts a value to a voltage for a primary channel.
>
> **Parameters**
> * **value** (`int`) – Value to be converted to voltage.
> * **ssGain** (`int | None, optional`) – Second stage gain. Defaults to None.
> * **PreampGain** (`int | None, optional`) – Preamplifier gain. Defaults to None.
>
> **Returns**
> Number of the voltage in volts [V]. Returns value if no gain is given (no-connect).
>
> **Return type**
> float

static **_Voltage_PrimaryChannels_Biosensor**(*value: int*, *ssGain: int*) → float

> Converts a value to a voltage for a biosensor primary channel.
>
> **Parameters**
> * **value** (`int`) – Value to be converted to voltage.
> * **ssGain** (`int`) – Second stage gain.
>
> **Returns**
> Number of the voltage in volts [V].
>
> **Return type**
> float

static **_Voltage_PrimaryChannels_EEGEMG**(*value: int*, *ssGain: int*, *PreampGain: int*) → float

> Converts a value to a voltage for an EEG/EMG primary channel.
>
> **Parameters**
> * **value** (`int`) – Value to be converted to voltage.
> * **ssGain** (`int`) – Second stage gain.
> * **PreampGain** (`int`) – Preamplifier gain.
>
> **Returns**
> Number of the voltage in volts [V].
>
> **Return type**
> float

static **_Voltage_SecondaryChannels**(*value: int*) → float

> Converts a value to a voltage for a secondary channel.
>
> **Parameters**
> **value** (`int`) – Value to be converted to voltage.
>
> **Returns**
> Number of the voltage in volts [V].

> **Return type**
> float

PodApi.Packets.Packet module

class PodApi.Packets.Packet.**Packet**(*pkt: bytes*, *commands:* CommandSet *| None = None*)

> Bases: `object`
>
> Container class that stores a command packet for a POD device. The format is STX (1 byte) + command number (4 bytes) + data (? bytes) + ETX (1 byte). This class also has a collection of methods for creating and interpreting POD packets.
>
> **_commands**
>
> > Available commands for a POD device.
> >
> > **Type**
> > POD_Commands | None
>
> **rawPacket**
>
> > Bytes string containing a POD packet. Should begin with STX and end with ETX.
> >
> > **Type**
> > bytes
>
> **commandNumber**
>
> > Command number from the Pod packet.
> >
> > **Type**
> > bytes | None
>
> **static ASCIIbytesToInt_Split**(*msg: bytes*, *keepTopBits: int*, *cutBottomBits: int*) → int
>
> > Converts a specific bit range in an ASCII-encoded bytes object to an integer.
> >
> > **Parameters**
> >
> > - **msg** (`bytes`) – Bytes message holding binary information to be converted into an integer.
> > - **keepTopBits** (`int`) – Integer position of the msb of desired bit range.
> > - **cutBottomBits** (`int`) – Integer number of lsb to remove.
> >
> > **Returns**
> > Integer result from the ASCII-encoded bytes message in a given bit range.
> >
> > **Return type**
> > int
>
> **static AsciiBytesToInt**(*msg_b: bytes*, *signed: bool = False*) → int
>
> > Converts a ASCII-encoded bytes message into an integer. It does this using a base-16 conversion. If the message is signed and the msb is '1', the integer will be converted to it's negative 2's complement.
> >
> > **Parameters**
> >
> > - **msg_b** (`bytes`) – Bytes message to be converted to an integer. The bytes must be base-16 or the conversion will fail.
> > - **signed** (`bool, optional`) – True if the message is signed, false if unsigned. Defaults to False.
> >
> > **Returns**
> > Integer result from the ASCII-encoded byte conversion.

Return type
> int

static BinaryBytesToInt(*msg: bytes*, *byteorder: str = 'big'*, *signed: bool = False*) → int

Converts binary-encoded bytes into an integer.

Parameters

- **msg** (`bytes`) – Bytes message holding binary information to be converted into an integer.

- **byteorder** (`str, optional`) – Ordering of bytes. 'big' for big endian and 'little' for little endian. Defaults to 'big'.

- **signed** (`bool, optional`) – Boolean flag to mark if the msg is signed (True) or unsigned (False). Defaults to False.

Returns
> Integer result from the binary-encoded bytes message.

Return type
> int

static BinaryBytesToInt_Split(*msg: bytes*, *keepTopBits: int*, *cutBottomBits: int*, *byteorder: str = 'big'*, *signed: bool = False*) → int

Converts a specific bit range in a binary-encoded bytes object to an integer.

Parameters

- **msg** (`bytes`) – Bytes message holding binary information to be converted into an integer.

- **keepTopBits** (`int`) – Integer position of the msb of desired bit range.

- **cutBottomBits** (`int`) – Integer number of lsb to remove.

- **byteorder** (`str, optional`) – Ordering of bytes. 'big' for big endian and 'little' for little endian. Defaults to 'big'.

- **signed** (`bool, optional`) – Boolean flag to mark if the msg is signed (True) or unsigned (False). Defaults to False.

Returns
> Integer result from the binary-encoded bytes message in a given bit range.

Return type
> int

static BuildPODpacket_Standard(*commandNumber: int*, *payload: bytes | None = None*) → bytes

Builds a standard POD packet as bytes: STX (1 byte) + command number (4 bytes) + optional packet (? bytes) + checksum (2 bytes)+ ETX (1 bytes).

Parameters

- **commandNumber** (`int`) – Integer representing the command number. This will be converted into a 4 byte long ASCII-encoded bytes string.

- **payload** (`bytes | None, optional`) – bytes string containing the payload. Defaults to None.

Returns
> Bytes string of a complete standard POD packet.

Return type
> bytes

---

static CheckIfPacketIsValid(*msg: bytes*)

> Raises an Exception if the packet is incorrectly formatted.
>
> > **Parameters**
> > > **msg** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
> >
> > **Raises**
> > > - **Exception** – Packet must begin with STX.
> > >
> > > - **Exception** – Packet must end in ETX

static Checksum(*bytesIn: bytes*) → bytes

> Calculates the checksum of a given bytes message. This is achieved by summing each byte in the message, inverting, and taking the last byte.
>
> > **Parameters**
> > > **bytesIn** (*bytes*) – Bytes message containing POD packet data.
> >
> > **Returns**
> > > Two ASCII-encoded bytes containing the checksum for bytesIn.
> >
> > **Return type**
> > > bytes

CommandNumber() → int

> Translate the binary ASCII encoding into a readable integer
>
> > **Returns**
> > > Integer of the command number.
> >
> > **Return type**
> > > int

static ETX() → bytes

> Get end-of-transmission (ETX) character in bytes. ETX marks the end byte of a POD Packet.
>
> > **Returns**
> > > Bytes for ETX(0x03).
> >
> > **Return type**
> > > bytes

static GetCommandNumber(*pkt: bytes*) → bytes | None

> Gets the command number bytes from a POD packet.
>
> > **Parameters**
> > > **pkt** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.
> >
> > **Returns**
> > > Bytes string of the command number, if available.
> >
> > **Return type**
> > > bytes|None

static GetMinimumLength() → int

> Gets the number of bytes in the smallest possible packet; STX (1 byte) + something + ETX (1 byte).
>
> > **Returns**
> > > integer representing the minimum length of a generic bytes string.

> **Return type**
>> int

HasCommandNumber() → bool

> Checks if the packet has a command number.

>> **Returns**
>>> True if the packet has a command number, False otherwise.

>> **Return type**
>>> bool

HasCommands() → bool

> Checks if the Packet instance has commands set.

>> **Returns**
>>> True if the commands have been set, false otherwise.

>> **Return type**
>>> bool

static **IntToAsciiBytes**(*value: int*, *numChars: int*) → bytes

> Converts an integer value into ASCII-encoded bytes.

> First, it converts the integer value into a usable uppercase hexadecimal string. Then it converts the ASCII code for each character into bytes. Lastly, it ensures that the final message is the desired length.

> Example: if value=2 and numBytes=4, the returned ASCII will show b'0002', which is '0x30 0x30 0x30 0x32' in bytes. Uses the 2's complement if the val is negative.

>> **Parameters**

>>> • **value** (`int`) – Integer value to be converted into ASCII-encoded bytes.

>>> • **numChars** (`int`) – Number characters to be the length of the ASCII-encoded message.

>> **Returns**
>>> Bytes that are ASCII-encoded conversions of the value parameter.

>> **Return type**
>>> bytes

static **PayloadToBytes**(*payload: int | bytes | tuple[int | bytes]*, *argSizes: tuple[int]*) → bytes

> Converts a payload into a bytes string.

>> **Parameters**

>>> • **payload** (`int | bytes | tuple[int | bytes]`) – Integer, bytes, or tuple containing the payload.

>>> • **argSizes** (`tuple[int]`) – Tuple of the argument sizes.

>> **Raises**

>>> • **Exception** – Payload requires multiple arguments, use a tuple.

>>> • **Exception** – Payload is the wrong size.

>>> • **Exception** – Payload has an incorrect number of items.

>>> • **Exception** – Payload has invalid values.

>>> • **Exception** – Payload is an invalid type.

>> **Returns**
>>> Bytes string of the payload.

Return type
> bytes

static STX() → bytes

> Get start-of-transmission (STX) character in bytes. STX marks the starting byte of a POD Packet.

> **Returns**
> > Bytes for STX (0x02).

> **Return type**
> > bytes

TranslateAll() → dict[str, Any]

> Builds a dictionary containing all parts of the POD packet in readable values.

> **Raises**
> > **Exception** – Nothing to translate.

> **Returns**
> > Dictionary with the command number.

> **Return type**
> > dict[str,Any]

static TwosComplement(*val: int*, *nbits: int*) → int

> Gets the 2's complement of the argument value.

> **Parameters**
> > • **val** (*int*) – Value to be complemented.
> >
> > • **nbits** (*int*) – Number of bits in the value.

> **Returns**
> > Integer of the 2's complement for the val.

> **Return type**
> > int

UnpackAll() → dict[str, bytes]

> Builds a dictionary containing all parts of the POD packet in bytes.

> **Raises**
> > **Exception** – Nothing to unpack.

> **Returns**
> > Dictionary with the command number.

> **Return type**
> > dict[str,bytes]

PodApi.Packets.Standard module

class PodApi.Packets.Standard.**PacketStandard**(*pkt: bytes*, *commands:* CommandSet)

> Bases: *Packet*

> Container class that stores a standard command packet for a POD device. The format is STX (1 byte) + command number (4 bytes) + optional payload (? bytes) + checksum (2 bytes) + ETX (1 bytes)

_customPayload

Optional function to translate the payload.

> **Type**
>> Callable[[Any],tuple]|None

_customPayloadArgs

Optional arguments for the _customPayload.

> **Type**
>> tuple[Any]|None

payload

Optional payload from the packet.

> **Type**
>> bytes

static CheckIfPacketIsValid(*msg: bytes*)

Raises an Exception if the packet is incorrectly formatted.

> **Parameters**
>> **msg** (*bytes*) – Bytes string containing a POD packet. Should begin with STX and end with ETX.

> **Raises**
>> **Exception** – Packet is too small to be a standard packet.

DefaultPayload() → tuple[int]

Splits the payload up into its components and translates the binary ASCII encoding into a readable integer.

> **Returns**
>> Tuple with integer values for each component of the payload.

> **Return type**
>> tuple[int]

static GetMinimumLength() → int

Gets the number of bytes in the smallest possible packet.

> **Returns**
>> integer representing the minimum length of a standard POD command packet. Format is STX (1 byte) + command number (4 bytes) + optional packet (? bytes) + checksum (2 bytes) + ETX (1 bytes)

> **Return type**
>> int

static GetPayload(*pkt: bytes*) → bytes | None

Gets the payload from a POD packet, if available.

> **Parameters**
>> **pkt** (*bytes*) – Bytes string containing a POD packet.

> **Returns**
>> Bytes string of the payload, if available.

> **Return type**
>> bytes|None

`HasCustomPayload()` → bool

> Checks if a custom payload has been set.
>
> > **Returns**
> >
> > > True if there is a custom payload, False otherwise.
> >
> > **Return type**
> >
> > > bool

`HasPayload()` → bool

> Checks if this Packet_Standard instance has a payload.
>
> > **Returns**
> >
> > > True if there is a payload, false otherwise.
> >
> > **Return type**
> >
> > > bool

`Payload()` → tuple | None

> Gets the payload as a readable tuple of values.
>
> > **Returns**
> >
> > > Translated payload, if available.
> >
> > **Return type**
> >
> > > tuple|None

`SetCustomPayload`(*func: Callable[[Any], tuple]*, *args: tuple[Any] | None = None*) → None

> Sets a custom function with optional arguments to translate the payload.
>
> > **Parameters**
> >
> > > - `func` (`Callable[[Any],tuple]`) – Function to translate the payload.
> > >
> > > - `args` (`tuple[Any], optional`) – Arguments . Defaults to None.

`TranslateAll()` → dict[str, Any]

> Builds a dictionary containing all parts of the POD packet in readable values.
>
> > **Returns**
> >
> > > Dictionary with the command number and payload.
> >
> > **Return type**
> >
> > > dict[str,Any]

`UnpackAll()` → dict[str, bytes]

> Builds a dictionary containing all parts of the POD packet in bytes.
>
> > **Returns**
> >
> > > Dictionary with the command number and payload.
> >
> > **Return type**
> >
> > > dict[str,bytes]

Module contents

1.1.1.4 PodApi.Parameters package

Submodules

PodApi.Parameters.Params8206HR module

class PodApi.Parameters.Params8206HR.**Params8206HR**(*port: str*, *sampleRate: int*, *preamplifierGain: int*, *lowPass: tuple[int]*, *checkForValidParams: bool = True*)

> Bases: *Params*
>
> Container class that stores parameters for an 8206-HR POD device.
>
> **port**
>> Name of the COM port.
>>
>>> **Type**
>>>> str
>
> **sampleRate**
>> Sample rate in 100-2000 Hz range.
>>
>>> **Type**
>>>> int
>
> **preamplifierGain**
>> Preamplifier gain. Should be 10x or 100x.
>>
>>> **Type**
>>>> int
>
> **lowPass**
>> Low-pass for EEG/EMG in 11-500 Hz range.
>>
>>> **Type**
>>>> tuple[int]
>
> **EEG1**() → int
>> Gets the filter value of EEG1 in Hz from the low-pass.
>>
>>> **Returns**
>>>> EEG1 low-pass filter in Hz.
>>
>>> **Return type**
>>>> int
>
> **EEG2**() → int
>> Gets the filter value of EEG2 in Hz from the low-pass.
>>
>>> **Returns**
>>>> EEG2 low-pass filter in Hz.
>>
>>> **Return type**
>>>> int

EEG3_EMG() → int

> Gets the filter value of EEG3/EMG in Hz from the low-pass.
>
> > **Returns**
> > > EEG3/EMG low-pass filter in Hz.
> >
> > **Return type**
> > > int

GetInit() → str

> Builds a string that represents the Params_8206HR constructor with the arguments set to the values of this class instance.
>
> > **Returns**
> > > String that represents the Params_8206HR constructor.
> >
> > **Return type**
> > > str

_CheckParams() → None

> Throws an exception if Params_8206HR instance variable is an invalid value.
>
> > **Raises**
> >
> > - **Exception** – Sample rate must be between 100-2000 Hz.
> > - **Exception** – Preamplidier gain must be 10x or 100x.
> > - **Exception** – Low-pass EEG/EMG must be between 11-500 Hz.

lowPassLabels: tuple[str] = ('EEG1', 'EEG2', 'EEG/EMG')

> Tuple describing the items in the lowPass.

PodApi.Parameters.Params8229 module

class PodApi.Parameters.Params8229.**Params8229**(*port: str*, *systemID: int*, *motorDirection: bool*, *motorSpeed: int*, *randomReverse: bool*, *mode: int*, *reverseBaseTime: int | None = None*, *reverseVarTime: int | None = None*, *schedule: dict[str, tuple[bool]] | None = None*, *checkForValidParams: bool = True*)

> Bases: *Params*
>
> Container class that stores parameters for an 8229 POD device.
>
> **port**
>
> > Name of the COM port.
> >
> > > **Type**
> > > > str
>
> **systemID**
>
> > ID of this 8229 POD system. Must be a positive integer.
> >
> > > **Type**
> > > > int
>
> **motorDirection**
>
> > False for clockwise and true for counterclockwise.

> **Type**
>> bool

motorSpeed

> Motor speed as a percentage 0-100%.
>
> > **Type**
> >> int

randomReverse

> True to enable random reverse, False otherwise. The random reverse time will be reverseBaseTime + random value in reverseVarTime range.
>
> > **Type**
> >> bool

reverseBaseTime

> Base time for a random reverse in seconds. Must be a positive integer.
>
> > **Type**
> >> int

reverseVarTime

> Variable time for a random reverse in seconds. Must be a positive integer.
>
> > **Type**
> >> int

mode

> System mode; 0 = Manual, 1 = PC Control, and 2 = Internal Schedule.
>
> > **Type**
> >> int

schedule

> Schedule for a week. The keys are the weekdays (Sunday-Saturday). The values are a tuple of 24 bools that are either 1 for motor on or 0 for motor off
>
> > **Type**
> >> dict[str, tuple[int]]

static BuildEmptySchedule() → dict[str, tuple[bool]]

> Creates a schedule where the motor is off for all hours of every day.
>
> > **Returns**
> >> Dictionary of the empty schedule. The keys are the days of the week. The values are tuples of 24 zeros.
> >
> > **Return type**
> >> dict[str, tuple[bool]]

GetInit() → str

> Builds a string that represents the Params_8229 constructor with the arguments set to the values of this class instance.
>
> > **Returns**
> >> String that represents the Params_Interface constructor.
> >
> > **Return type**
> >> str

_CheckParams() → None

> Throws an exception if Params_8229 instance variable is an invalid value.

> > **Raises**
> >
> > - **Exception** – The system ID must be a positive integer.
> >
> > - **Exception** – The motor speed must be between 0-100%.
> >
> > - **Exception** – The reverse base time (sec) must be a positive integer.
> >
> > - **Exception** – The reverse variable time (sec) must be a positive integer.
> >
> > - **Exception** – The mode must be 0, 1, or 2.
> >
> > - **Exception** – The schedule must have exactly ('Sunday','Monday','Tuesday', 'Wednesday','Thursday','Friday','Saturday') as keys.
> >
> > - **Exception** – There must be 24 items in the schedule for each day.

hoursPerDay: int = 24

> Integer storing the number of hours in a day.

week: tuple[str] = ('Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday')

> Tuple containing strings of the 7 days of the week.

PodApi.Parameters.Params8401HR module

class PodApi.Parameters.Params8401HR.**Params8401HR**(*port: str*, *preampDevice: str*, *sampleRate: int*, *muxMode: bool*, *preampGain: tuple[int]*, *ssGain: tuple[int]*, *highPass: tuple[float]*, *lowPass: tuple[int]*, *bias: tuple[float]*, *dcMode: tuple[str]*, *checkForValidParams: bool = True*)

> Bases: *Params*

> Container class that stores parameters for an 8401-HR POD device.

> **port**
>
> > Name of the COM port.
> >
> > > **Type**
> > > str

> **preampDevice**
>
> > Name of the mouse/rat preamplifier device.
> >
> > > **Type**
> > > str

> **sampleRate**
>
> > Sample rate (2000-20000 Hz).
> >
> > > **Type**
> > > int

> **muxMode**
>
> > Using mux mode when True, false otherwise.
> >
> > > **Type**
> > > bool

preampGain

Preamplifier gain (1, 10, or 100) for all channels.

> **Type**
>> tuple[int]

ssGain

Second stage gain (1 or 5) for all channels.

> **Type**
>> tuple[int]

highPass

High-pass filter (0, 0.5, 1, or 10 Hz) for all channels.

> **Type**
>> tuple[float]

lowPass

Low-pass filter (21-15000 Hz) for all channels.

> **Type**
>> tuple[int]

bias

Bias voltage (+/- 2.048 V) for all channels.

> **Type**
>> tuple[float]

dcMode

DC mode (VBIAS or AGND) for all channels.

> **Type**
>> tuple[str]

GetInit() → str

Builds a string that represents the Params_8401HR constructor with the arguments set to the values of this class instance.

> **Returns**
>> String that represents the Params_8401HR constructor.

> **Return type**
>> str

_CheckParams() → None

Throws an exception if Params_8401HR instance variable is an invalid value.

> **Raises**
> - **Exception** – Mouse/rat preamplifier does not exist.
> - **Exception** – Sample rate must be between 2000-20000 Hz.
> - **Exception** – EEG/EMG preamplifier gain must be 10x or 100x. For biosensors, the preampGain is None.
> - **Exception** – The second stage gain must be 1x or 5x; set gain to None if no-connect.
> - **Exception** – The high-pass filter must be 0.5, 1, or 10 Hz. If the channel is DC, input 0.
> - **Exception** – The low-pass filter must be between 21-15000 Hz.

- **Exception** – The bias voltage must be +/- 2.048 V.

- **Exception** – The DC mode must be VBIAS or AGND.

`channelLabels: tuple[str] = ('A', 'B', 'C', 'D')`

> Tuple listing the four channel characters in order.

PodApi.Parameters.Params8480SC module

class PodApi.Parameters.Params8480SC.**Params8480SC**(*port: str, stimulus=tuple[int], preamp=<class 'int'>, ledCurrent=tuple[int], ttlPullups=<class 'int'>, estimCurrent=tuple[int], syncConfig=<class 'int'>, ttlSetup=tuple[int], checkForValidParams: bool = True*)

> Bases: [Params](#)

> Container class that stores parameters for an 8401-HR POD device.

> **port**
>> Name of the COM port.
>>
>>> **Type**
>>>> str

> **stimulus**
>> Stimulus configuration on selected channel.
>>
>>> **Type**
>>>> tuple[int]

> **preamp**
>> Preamp value (0-1023).
>>
>>> **Type**
>>>> int

> **ledCurrent**
>> Led-Current (0-600 mA) for both channels.
>>
>>> **Type**
>>>> tuple[int]

> **ttlPUllups**
>> TTL Pullups disabled for value 0, pullups enabled for values that are non-zero.
>>
>>> **Type**
>>>> int

> **estimCurrent**
>> Estim-Current (0-100 %) for both channels.
>>
>>> **Type**
>>>> tuple[int]

> **syncConfig**
>> Sets Sync-Config byte.
>>
>>> **Type**
>>>> int

ttlSetup

    TTL-Setup for selected channel.

        Type

            tuple[int]

GetInit() → str

    Builds a string that represents the Params_8480SC constructor with the arguments set to the values of this class instance.

        Returns

            String that represents the Params_8480SC constructor.

        Return type

            str

_CheckParams() → None

    Throws an exception if Params_8206HR member variable is an invalid value.

        Raises

            • **Exception** – The preamp must be between 0-1023.

            • **Exception** – Led-Curent must be between 0-600.

            • **Exception** – Estim-Current must be between 0-100.

estimCurrent_CH0() → int

    Gets the estimCurrent value for Channel 0.

        Returns

            Channel 0 estimCurrent in percentage.

        Return type

            int

estimCurrent_CH1() → int

    Gets the estimCurrent value for Channel 1.

        Returns

            Channel 1 estimCurrent in percentage.

        Return type

            int

ledCurrent_CH0() → int

    Gets the ledCurrent value for Channel 0.

        Returns

            Channel 0 ledCurrent in mA.

        Return type

            int

ledCurrent_CH1() → int

    Gets the ledCurrent value for Channel 1.

        Returns

            Channel 1 ledCurrent in mA.

        Return type

            int

PodApi.Parameters.ParamsBasic module

class PodApi.Parameters.ParamsBasic.**Params**(*port: str*, *checkForValidParams: bool = True*)

> Bases: object
>
> Interface for a container class that stores parameters for a POD device.
>
> **port**
>
> > Name of the COM port.
> >
> > > **Type**
> > > > str
>
> **GetInit**() → str
>
> > Builds a string that represents the Params_Interface constructor with the arguments set to the values of this class instance.
> >
> > > **Returns**
> > > > String that represents the Params_Interface constructor.
> > >
> > > **Return type**
> > > > str
>
> **_CheckParams**() → None
>
> > Throws an exception if Params_Interface instance variable is an invalid value.
> >
> > > **Raises**
> > > > **Exception** – The port name must begin with COM.
>
> **static _FixTypeInTuple**(*arr: tuple*, *itemType: type*) → tuple['type']
>
> > Retypes each item of the arr arguemnt to itemType.
> >
> > > **Parameters**
> > > > - **arr** (*tuple*) – Tuple of items to be re-typed.
> > > > - **itemType** (*type*) – Type to be casted to each tuple item.
> > >
> > > **Returns**
> > > > Tuple with values of all itemType types.
> > >
> > > **Return type**
> > > > tuple[type]

Module contents

1.1.2 Module contents

1.2 Setup package

1.2.1 Subpackages

1.2.1.1 Setup.Inputs package

Submodules

Setup.Inputs.GetUserInput module

class Setup.Inputs.GetUserInput.**UserInput**

> Bases: object
>
> UserInput contains several methods for getting user input for POD device setup.
>
> **static AskForBool**(*prompt: str*) → bool
>
> > Asks user for bool type input.
> >
> > > **Parameters**
> > > > **prompt** (`str`) – Statement requesting input from the user.
> > >
> > > **Returns**
> > > > Boolean type input from user.
> > >
> > > **Return type**
> > > > bool
>
> **static AskForFloat**(*prompt: str*) → float
>
> > Asks user for float type input.
> >
> > > **Parameters**
> > > > **prompt** (`str`) – Statement requesting input from the user.
> > >
> > > **Returns**
> > > > Float type input from user.
> > >
> > > **Return type**
> > > > float
>
> **static AskForFloatInList**(*prompt: str*, *goodInputs: list*, *badInputMessage: str | None = None*) → float
>
> > Asks the user for a float that exists in the list of valid options.
> >
> > > **Parameters**
> > > > - **prompt** (`str`) – Statement requesting input from the user.
> > > > - **goodInputs** (`list`) – List of valid input options.
> > > > - **badInputMessage** (`str | None, optional`) – Error message to be printed if invalid input is given. Defaults to None.
> > >
> > > **Returns**
> > > > User's choice from the options list as a float.
> > >
> > > **Return type**
> > > > float
>
> **static AskForFloatInRange**(*prompt: str*, *minimum: float*, *maximum: float*, *thisIs: str = 'Input'*, *unit: str = ''*) → float
>
> > Asks the user for an float value that falls in a given range.
> >
> > > **Parameters**
> > > > - **prompt** (`str`) – Statement requesting input from the user.
> > > > - **minimum** (`float`) – Minimum value of range.
> > > > - **maximum** (`float`) – Maximum value of range.
> > > > - **thisIs** (`str, optional`) – Description of the input/what is being asked for. Used when printing the error message. Defaults to 'Input'.

- **unit** (`str`, `optional`) – Unit of the requested value. Use when printing the error message. Defaults to ''.

> **Returns**
> Float value given by the user that falls in the given range.

> **Return type**
> float

static **AskForInput**(*prompt: str*, *append: str = ': '*) → str

> Asks user for input given a prompt. Will append a colon ':' to the end of prompt by default

> **Parameters**
>
> - **prompt** (`str`) – Statement requesting input from the user
>
> - **append** (`str`, `optional`) – Appended to the end of the prompt. Defaults to ': '.

> **Returns**
> String of the user input

> **Return type**
> str

static **AskForInt**(*prompt: str*) → int

> Asks user for int type input.

> **Parameters**
> **prompt** (`str`) – Statement requesting input from the user.

> **Returns**
> Integer type input from user.

> **Return type**
> int

static **AskForIntInList**(*prompt: str*, *goodInputs: list*, *badInputMessage: str | None = None*) → int

> Asks the user for an integer that exists in the list of valid options.

> **Parameters**
>
> - **prompt** (`str`) – Statement requesting input from the user
>
> - **goodInputs** (`list`) – List of valid input options.
>
> - **badInputMessage** (`str | None`, `optional`) – Error message to be printed if invalid input is given. Defaults to None.

> **Returns**
> User's choice from the options list as an integer.

> **Return type**
> int

static **AskForIntInRange**(*prompt: str*, *minimum: int*, *maximum: int*, *thisIs: str = 'Input'*, *unit: str = ''*) → int

> Asks the user for an integer value that falls in a given range.

> **Parameters**
>
> - **prompt** (`str`) – Statement requesting input from the user.
>
> - **minimum** (`int`) – Minimum value of range.
>
> - **maximum** (`int`) – Maximum value of range.

- **thisIs** (`str, optional`) – Description of the input/what is being asked for. Used when printing the error message. Defaults to 'Input'.

- **unit** (`str, optional`) – Unit of the requested value. Use when printing the error message. Defaults to ''.

> **Returns**
> Integer value given by the user that falls in the given range.

> **Return type**
> int

static **AskForStrInList**(*prompt: str*, *goodInputs: list*, *badInputMessage: str | None = None*) → str

Asks the user for a string that exists in the list of valid options.

> **Parameters**
>
> - **prompt** (`str`) – Statement requesting input from the user.
>
> - **goodInputs** (`list`) – List of valid input options
>
> - **badInputMessage** (`str | None, optional`) – Error message to be printed if invalid input is given. Defaults to None.

> **Returns**
> User's choice from the options list as a string.

> **Return type**
> str

static **AskForType**(*typecast: function*, *prompt: str*) → int | float | str

Ask user for input of a specific data type. If invalid input is given, an error message will print and the user will be prompted again.

> **Parameters**
>
> - **typecast** (`function`) – Datatype to cast the user input (ex. _CastInt, _CastFloat, _CastStr)
>
> - **prompt** (`str`) – Statement requesting input from the user

> **Returns**
> Input from user as the requested type.

> **Return type**
> int|float|str

static **AskForTypeInList**(*typecast: function*, *prompt: str*, *goodInputs: list*, *badInputMessage: str | None = None*) → int | float | str

Asks the user for a value of a given type that exists in the list of valid options. If invalid input is given, an error message will print and the user will be prompted again.

> **Parameters**
>
> - **typecast** (`function`) – Datatype to cast the user input (ex. _CastInt, _CastFloat, _CastStr).
>
> - **prompt** (`str`) – Statement requesting input from the user.
>
> - **goodInputs** (`list`) – List of valid input options.
>
> - **badInputMessage** (`str | None, optional`) – Error message to be printed if invalid input is given. Defaults to None.

> **Returns**
>> User's choice from the goodInputs list as the given datatype.
>
> **Return type**
>> int|float|str

static **AskForTypeInRange**(*typecast: function*, *prompt: str*, *minimum: int | float*, *maximum: int | float*,
     thisIs: str = 'Input', *unit: str = ''*) → int | float

> Asks user for a numerical value that falls between two numbers. If invalid input is given, an error message
> will print and the user will be prompted again.
>
> **Parameters**
>
>> - **typecast** (`function`) – Datatype to cast the user input (ex. _CastInt, _CastFloat, _Cast-Str).
>>
>> - **prompt** (`str`) – Statement requesting input from the user.
>>
>> - **minimum** (`int | float`) – Minimum value of range.
>>
>> - **maximum** (`int | float`) – Maximum value of range.
>>
>> - **thisIs** (`str, optional`) – Description of the input/what is being asked for. Used when
>>   printing the error message. Defaults to 'Input'.
>>
>> - **unit** (`str, optional`) – Unit of the requested value. Use when printing the error mes-sage. Defaults to ''.
>
> **Returns**
>> Numerical value given by the user that falls in the given range.
>
> **Return type**
>> int|float

static **AskYN**(*question: str*, *append: str = ' (y/n): '*) → bool

> Asks the user a yes or no question. If invalid input is given, an error message will print and the user will
> be prompted again.
>
> **Parameters**
>
>> - **question** (`str`) – Statement requesting input from the user.
>>
>> - **append** (`str, optional`) – Appended to the end of the question. Defaults to ' (y/n): '.
>
> **Returns**
>> True for yes, false for no.
>
> **Return type**
>> bool

static **CastFloat**(*value*) → float

> Casts the argument as an float.
>
> **Parameters**
>> **value** – Value to type casted.
>
> **Returns**
>> Value type casted as a float.
>
> **Return type**
>> float

static CastInt(*value*) → int

    Casts the argument as an integer.

        Parameters
            value – Value to type casted.

        Returns
            Value type casted as an integer.

        Return type
            int

static CastStr(*value*) → str

    Casts the argument as an string.

        Parameters
            value – Value to type casted.

        Returns
            Value type casted as a string.

        Return type
            str

static CheckFileExt(*f: str*, *fIsExt: bool = True*, *goodExt: list[str] = ['.txt']*, *printErr: bool = True*) → bool

    Checks if a file name has a valid extension.

        Parameters

            • **f** (`str`) – file name or extension

            • **fIsExt** (`bool, optional`) – Boolean flag that is true if f is an extension, false otherwise. Defaults to True.

            • **goodExt** (`list[str], optional`) – List of valid file extensions. Defaults to ['.txt',].

            • **printErr** (`bool, optional`) – Boolean flag that, when true, will print an error statement. Defaults to True.

        Returns
            True if extension is in goodExt list, False otherwise.

        Return type
            bool

static GetFileName(*goodExt: list[str] = ['.txt']*) → str

    Asks the user for a filename. :param goodExt: List of valid file extensions. Defaults to ['.txt']. :type goodExt: list[str], optional

        Returns
            String of the file name and extension.

        Return type
            str

static GetFilePath(*prompt: str | None = None*, *goodExt: list[str] = ['.txt']*) → str

    Asks the user for a file path and file name.

        Parameters

            • **prompt** (`str | None, optional`) – Text to print to the user before requesting the path. Defaults to None.

- **goodExt** (`list[str], optional`) – List of valid file extensions. Defaults to ['.txt'].

Returns
> File path and name.

Return type
> str

Module contents

1.2.1.2 Setup.SetupAllDevices package

Submodules

Setup.SetupAllDevices.Setup_PodDevices module

class Setup.SetupAllDevices.Setup_PodDevices.**SetupAll**(*saveFileDict: dict[str, str] | None = None*, *podParametersDict: dict[str, dict[int,* [PodApi.Parameters.ParamsBasic.Params](#)*] | None] | None = None*)

> Bases: `object`
>
> Setup_PodDevices allows a user to set up and stream from any number of POD devices. The streamed data is saved to a file.
>
> REQUIRES FIRMWARE 1.0.2 OR HIGHER.
>
> **_Setup_PodDevices**
> > Dictionary containing the Setup_Interface subclasses for each POD device.
> >
> > **Type**
> > > dict[str,dict[int,Params_Interface]]
>
> **_options**
> > Dictionary listing the different options for the user to complete.
> >
> > **Type**
> > > dict[int,str]
>
> **GetOptions**() → dict[int, str]
> > Gets the dictionary of setup options.
> >
> > **Returns**
> > > Dictionary listing the different options for the user to complete (_options).
> >
> > **Return type**
> > > dict[int,str]
>
> **GetPODparametersInit**() → str
> > Sets up each POD device type. Used in initialization. :returns: String representing a dictionary of all POD devices initialization. The keys are the device name and the entries are the initialization dictionaries. :rtype: str
>
> **GetSaveFileNames**() → str
>
> **Run**() → None
> > Prints the options and askes the user what to do. Loops until 'Quit' is chosen.

SetupPODparameters(*podParametersDict: dict[str, dict[int,* PodApi.Parameters.ParamsBasic.Params*]* | *None]*) → None

> Sets up each POD device type. Used in initialization.

>> **Parameters**
>>> **podParametersDict** (`dict[str,dict[int,Params_Interface] | None]`) – Dictionary of all POD devices initialization. The keys are the device name and the entries are the initialization dictionaries.

SetupSaveFile(*saveFileDict: dict[str, str] | None = None*) → None

> Gets the path/file name from the user and stores it. Used in initialization.

>> **Parameters**
>>> **saveFile** (`dict[str,str|None] | None, optional`) – String of the save file, which includes the directory path, filename, and file extension. Defaults to None.

_AskOption() → int

> Asks user which option to do.

>> **Returns**
>>> Integer number representing an option key.

>> **Return type**
>>> int

_AskToStopStream() → None

> Asks user to press enter to stop streaming. The program will then prompt all POD devices to end stream.

static _AskUserForDevices()

> Asks the user what POD devices they want to use.

_CheckForValidParams(*podParametersDict: dict[str, None | dict]*) → bool

> Checks if the parameters are correctly formatted.

>> **Parameters**
>>> **podParametersDict** (`dict[str,None | dict]`) – Dictionary with keys as the device names and values as None or the respective parameter dictionary.

>> **Raises**
>>> • **Exception** – Parameters must be dictionary type.
>>> • **Exception** – Parameters dictionary is empty.
>>> • **Exception** – Invalid device name in paramater dictionary.

>> **Returns**
>>> True if the parameters are correctly formatted.

>> **Return type**
>>> bool

_ConnectNewDevice() → None

> Asks the user for the POD device type, then it sets up that device.

_DoOption(*choice: int*) → None

> Performs the methods associated with the user selected option.

>> **Parameters**
>>> **choice** (`int`) – Integer number representing an option key.

_EditCheckConnect() → None

> Displays the POD devices parameters, asks the user to edit the device, and then reconnects the device for each POD device type.

_EditSaveFilePath() → None

> Asks the user for the POD device type, then asks the user for a new file name and path, then sets the value to the POD devices.

_GetChosenDeviceType(*question: str*) → str

> Asks the user which type of POD device they want.

> > **Parameters**
> > > **question** (*str*) – Question to ask the user.

> > **Returns**
> > > String of the user input (may be invalid POD device type).

> > **Return type**
> > > str

_GetParams(*podParametersDict: None | dict[str, None]*) → dict[str, dict | None]

> If no parameters are give, this asks user which types of POD devices they want to use. Then it checks if the parameters are valid.

> > **Parameters**
> > > **podParametersDict** (*None | dict[str,None]*) – Dictionary of all POD devices initialization. The keys are the device name and the entries are the initialization dictionaries.

> > **Returns**
> > > Dictionary whose keys are the POD device name, and value the setup dictionary.

> > **Return type**
> > > dict[str,dict|None]

_PrintInitCode() → None

> Prints code that can be used to initialize and run SetupPodDevices with the current parameters.

_PrintOptions() → None

> Prints options available for user.

_Reconnect() → bool

> Reconnects all POD devices.

> > **Returns**
> > > Bool that is true if all devices were successfully connected. False otherwise.

> > **Return type**
> > > bool

_RemoveDevice() → None

> Displays the POD devices parameters, asks the user which device ro remove, and then deletes that POD device.

_Set_Setup_PodDevices(*podParametersDict: dict[str, dict | None]*) → None

> Sets the _Setup_PodDevices varaible to have keys as the POD device name and values as the setup class.

> > **Parameters**
> > > **podParametersDict** (*dict[str,dict | None]*) – Dictionary with keys as the device names and values as None or the respective parameter dictionary.

_ShowCurrentSettings() → None

>   Displays the POD device settings for all devices, and then prints the save file name.

_Stream() → float

>   Streams data from all POD devices and prints the execution time.

>> **Returns**
>>>   Float of the execution time in seconds.

>> **Return type**
>>>   float

_StreamAllDevices() → None

>   Streams data from all the devices. User is asked to click enter to stop streaming. Data is saved to file. Uses threading.

static _TimeFunc(*func: function*) → float

>   Runs a function and gets the calculated execultion time.

>> **Parameters**
>>>   **func** (`function`) – Function/method name.

>> **Returns**
>>>   Float of the execution time in seconds rounded to 3 decimal places.

>> **Return type**
>>>   float

Module contents

1.2.1.3 Setup.SetupOneDevice package

Submodules

Setup.SetupOneDevice.Setup_8206HR module

class Setup.SetupOneDevice.Setup_8206HR.**Setup8206HR**

>   Bases: *SetupInterface*

>   Setup_8206HR provides the setup functions for an 8206-HR POD device.

>   **static GetDeviceName**() → str

>>   Returns the name of the 8206-HR POD device.

>>>   **Returns**
>>>>   8206-HR.

>>>   **Return type**
>>>>   str

>   **StopStream**() → None

>>   Write a command to stop streaming data to all POD devices.

>   **static _ChoosePreampGain**() → int

>>   Asks user for the preamplifier gain of their POD device.

>>>   **Returns**
>>>>   Integer 10 or 100 for the preamplifier gain.

> **Return type**
> > int

_ConnectPODdevice(*deviceNum: int*, *deviceParams:* Params8206HR) → bool

> Creates a POD_8206HR object and write the setup parameters to it.
>
> > **Parameters**
> >
> > • **deviceNum** (`int`) – Integer of the device's number.
> >
> > • **deviceParams** (`Params_8206HR`) – Device's parameters.
> >
> > **Returns**
> > > True if connection was successful, false otherwise.
> >
> > **Return type**
> > > bool

_GetPODdeviceParameterTable() → Texttable

> Builds a table containing the parameters for all POD devices.
>
> > **Returns**
> > > Texttable containing all parameters.
> >
> > **Return type**
> > > Texttable

_GetParam_onePODdevice(*forbiddenNames: list[str] = []*) → *Params8206HR*

> Asks the user to input all the device parameters.
>
> > **Parameters**
> > > **forbiddenNames** (`list[str]`) – List of port names already used by other devices.
> >
> > **Returns**
> > > Dictionary of device parameters.
> >
> > **Return type**
> > > dict[str,(str|int|dict[str,int])]

_OpenSaveFile_EDF(*fname: str*, *devNum: int*) → EdfWriter

> Opens EDF file and write header.
>
> > **Parameters**
> >
> > • **fname** (`str`) – String filename.
> >
> > • **devNum** (`int`) – Integer device number.
> >
> > **Returns**
> > > Opened file.
> >
> > **Return type**
> > > EdfWriter

_OpenSaveFile_TXT(*fname: str*) → IOBase

> Opens a text file and writes the column names. Writes the current date/time at the top of the txt file.
>
> > **Parameters**
> > > **fname** (`str`) – String filename.
> >
> > **Returns**
> > > Opened file.

> > > **Return type**
> > > IOBase

> > **_PHYSICAL_BOUND_uV: int = 2046**

> > > Class-level integer representing the max/-min physical value in uV. Used for EDF files.

> > **_StreamThreading**() → dict[int, threading.Thread]

> > > Opens a save file, then creates a thread for each device to stream and write data from.

> > > **Returns**
> > > Dictionary with keys as the device number and values as the started Thread.

> > > **Return type**
> > > dict[int,Thread]

> > **_StreamUntilStop**(*pod:* Pod8206HR, *file: IOBase | EdfWriter*, *sampleRate: int*) → None

> > > Streams data from a POD device and saves data to file. Stops looking when a stop stream command is read. Calculates average time difference across multiple packets to collect a continuous time series data.

> > > **Parameters**

> > > > • **pod** (*POD_8206HR*) – POD device to read from.

> > > > • **file** (*IOBase | EdfWriter*) – open file.

> > > > • **sampleRate** (*int*) – Integer sample rate in Hz.

> > **static _WriteDataToFile_EDF**(*file: EdfWriter*, *data: list[numpy.ndarray]*)

> > > Writes data to an open EDF file.

> > > **Parameters**

> > > > • **file** (*EdfWriter*) – opened EDF file.

> > > > • **data** (*list[np.ndarray]*) – List of 3 items, one for each channel.

> > **static _WriteDataToFile_TXT**(*file: IOBase*, *data: list[numpy.ndarray]*, *t: ndarray*)

> > > Writes data to an open text file.

> > > **Parameters**

> > > > • **file** (*IOBase*) – opened write file.

> > > > • **data** (*list[np.ndarray]*) – List of 3 items, one for each channel.

> > > > • **t** (*np.ndarray*) – list with the time stamps (in seconds).

Setup.SetupOneDevice.Setup_8229 module

class Setup.SetupOneDevice.Setup_8229.**Setup8229**

> Bases: *SetupInterface*

> Setup_8229 provides the setup functions for an 8229 POD device.

> **_streamMode**

> > True when the user wants to stream data from an 8229 POD device, False otherwise.

> > **Type**
> > bool

---

static GetDeviceName() → str

> Returns the name of the POD device.
>
> > **Returns**
> > 8229.
> >
> > **Return type**
> > str

static GetSupportedFileExtensions() → list[str]

> Returns a list containing valid file extensions.
>
> > **Returns**
> > List of string file extensions.
> >
> > **Return type**
> > list[str]

StopStream() → None

> Update the state flag to signal to stop streaming data.

_ConnectPODdevice(*deviceNum: int*, *deviceParams:* Params8229) → bool

> Creates a 8229 POD device object and write the setup parameters to it.
>
> > **Parameters**
> > - **deviceNum** (`int`) – Integer of the device's number.
> > - **deviceParams** (`Params_8229`) – Device parameters.
> >
> > **Returns**
> > True if connection was successful, false otherwise.
> >
> > **Return type**
> > bool

_GetPODdeviceParameterTable() → Texttable

> Builds a table containing the parameters for all POD devices.
>
> > **Returns**
> > Table containing all parameters.
> >
> > **Return type**
> > Texttable

_GetParam_onePODdevice(*forbiddenNames: list[str] = [ ]*) → *Params8229*

> Asks the user to input all the device parameters.
>
> > **Parameters**
> > **forbiddenNames** (`list[str], optional`) – List of port names already used by other devices. Defaults to [].
> >
> > **Returns**
> > Device parameters.
> >
> > **Return type**
> > Params_8229

static _GetScheduleForWeek() → dict[str, tuple[int]]

> Asks the user to input if the motor is on/off for each hour of each day of the week.

> **Returns**
>> Dictionary with the schedule. The keys are the days of the week (Sunday, Monday, . . .). The values are a tuple of 24 items for each hour; the items are 1 if the motor is on or 0 if the motor is off

> **Return type**
>> dict[str, tuple[int]]

_OpenSaveFile_EDF(*fname: str*, *devNum: int*)

> EDF files are not supported for 8229 POD devices. Overwrites the parent's method, which would open an EDF file and write the header.

> **Parameters**
>> - **fname** (`str`) – String filename. Not used.
>> - **devNum** (`int`) – Integer device number. Not used.

> **Raises**
>> **Exception** – EDF filetype is not supported for 8229 POD devices.

_OpenSaveFile_TXT(*fname: str*) → IOBase

> Opens a text file and writes the column names. Writes the current date/time at the top of the txt file.

> **Parameters**
>> **fname** (`str`) – String filename.

> **Returns**
>> Opened file.

> **Return type**
>> IOBase

_StreamThreading() → dict[int, threading.Thread]

> Opens a save file, then creates a thread for each device to stream and write data from.

> **Returns**
>> Dictionary with keys as the device number and values as the started Thread.

> **Return type**
>> dict[int,Thread]

_StreamUntilStop(*pod:* Pod8229, *file: IOBase*) → None

> Saves a log of all packets recieved from the 8229 POD device until the user decides to stop streaming.

> **Parameters**
>> - **pod** (`POD_8229`) – POD device to read from.
>> - **file** (`IOBase`) – Opened text file to save data to.

Setup.SetupOneDevice.Setup_8401HR module

class Setup.SetupOneDevice.Setup_8401HR.**Setup8401HR**

> Bases: *SetupInterface*

> Setup_8401HR provides the setup functions for an 8206-HR POD device. REQUIRES FIRMWARE 1.0.2 OR HIGHER.

---

static GetDeviceName() → str

returns the name of the 8401-HR POD device.

> **Returns**
>> 8401-HR.
>
> **Return type**
>> str

StopStream() → None

Write a command to stop streaming data to all POD devices.

static _CodeDCmode(*dcMode: str*) → int

gets the integer payload to use for 'SET DC MODE' commands given the mode.

> **Parameters**
>> **dcMode** (`str`) – DC mode VBIAS or AGND.
>
> **Raises**
>> **Exception** – DC Mode value is not supported.
>
> **Returns**
>> Integer code representing the DC mode.
>
> **Return type**
>> int

static _CodeHighpass(*highpass: float*) → int

Gets the integer payload to use for 'SET HIGHPASS' given a highpass value.

> **Parameters**
>> **highpass** (`float`) – Highpass value in Hz.
>
> **Raises**
>> **Exception** – High-pass value is not supported.
>
> **Returns**
>> Integer code representing the highpass value.
>
> **Return type**
>> int

_ConnectPODdevice(*deviceNum: int*, *deviceParams:* Params8401HR) → bool

Creates a POD_8206HR object and write the setup parameters to it.

> **Parameters**
>
> - **deviceNum** (`int`) – Integer of the device's number.
>
> - **deviceParams** (`Params_8401HR`) – Device parameters.
>
> **Returns**
>> True if connection was successful, false otherwise.
>
> **Return type**
>> bool

_GetPODdeviceParameterTable() → Texttable

Builds a table containing the parameters for all POD devices.

> **Returns**
>> Texttable containing all parameters.

> > > **Return type**
> > > Texttable

_GetParam_onePODdevice(*forbiddenNames: list[str] = []*) → *Params8401HR*

> Asks the user to input all the device parameters.
>
> > **Parameters**
> > **forbiddenNames** (`list[str]`) – List of port names already used by other devices.
> >
> > **Returns**
> > Device parameters.
> >
> > **Return type**
> > Params_8401HR

_GetPreampDeviceName() → str

> Asks the user to select a mouse/rat preamplifier.
>
> > **Returns**
> > String of the chosen preamplifier.
> >
> > **Return type**
> > str

_NiceABCDtableText(*abcdValues: list[int | str | None]*, *channelMap: dict[str, str]*) → str

> Builds a string that formats the channel values to be input into the parameter table.
>
> > **Parameters**
> >
> > - **abcdValueDict** (`dict[str,int | str | None]`) – Dictionary with ABCD keys.
> >
> > - **channelMap** (`dict[str,str]`) – Maps the ABCD channels to the sensor's channel name.
> >
> > **Returns**
> > String with "channel name: value newline…" for each channel.
> >
> > **Return type**
> > str

_OpenSaveFile_EDF(*fname: str*, *devNum: int*) → EdfWriter

> Opens EDF file and write header.
>
> > **Parameters**
> >
> > - **fname** (`str`) – String filename.
> >
> > - **devNum** (`int`) – Integer device number.
> >
> > **Returns**
> > Opened file.
> >
> > **Return type**
> > EdfWriter

_OpenSaveFile_TXT(*fname: str*) → IOBase

> Opens a text file and writes the column names. Writes the current date/time at the top of the txt file.
>
> > **Parameters**
> > **fname** (`str`) – String filename.
> >
> > **Returns**
> > Opened file.

> **Return type**
>> IOBase

_PHYSICAL_BOUND_uV: int = 2046

> Class-level integer representing the max/-min physical value in uV. Used for EDF files.

static _SetBias(*channelName: str*) → float

> Asks the user for the bias voltage in V (+/-2.048V).

>> **Parameters**
>>> **channelName** (*str*) – Name of the channel.

>> **Returns**
>>> A float for thebias voltage in V.

>> **Return type**
>>> float

static _SetDCMode(*channelName: str*) → str

> Asks the user for the DC mode (VBIAS or AGND).

>> **Parameters**
>>> **channelName** (*str*) – Name of the channel.

>> **Returns**
>>> String for the DC mode.

>> **Return type**
>>> str

_SetForMappedChannels(*message: str*, *channelMap: dict[str, str]*, *func: function*) → tuple[int | None]

> Asks the user to input values for all channels (excluding no-connects).

>> **Parameters**

>>> • **message** (*str*) – Message to ask the user.

>>> • **channelMap** (*dict[str,str]*) – Maps the ABCD channels to the sensor's channel name.

>>> • **func** (*function*) – a function that asks the user for an input. takes one string parameter and returns one value.

>> **Returns**
>>> Tuple with user inputs for values for the ABCD channels

>> **Return type**
>>> tuple[int|None]

static _SetHighpass(*channelName: str*) → float | None

> Asks the user for the high-pass in Hz (0.5,1,10Hz, or DC).

>> **Parameters**
>>> **channelName** (*str*) – Name of the channel.

>> **Returns**
>>> A float for the high-pass frequency in Hz, or None if DC.

>> **Return type**
>>> float|None

static _SetLowpass(*channelName: str*) → int | None

> Asks the user for the low-pass in Hz (21-15000Hz).

> **Parameters**
>> **channelName** (`str`) – Name of the channel.
>
> **Returns**
>> An integer for the low-pass frequency in Hz.
>
> **Return type**
>> int|None

static _SetPreampGain(*channelName: str*) → int | None

> Asks the user for the preamplifier gain.
>
>> **Parameters**
>>> **channelName** (`str`) – Name of the channel.
>>
>> **Returns**
>>> An integer for the gain, or None if no gain.
>>
>> **Return type**
>>> int|None

static _SetSSGain(*channelName: str*) → int

> Asks the user for the second stage gain.
>
>> **Parameters**
>>> **channelName** (`str`) – Name of the channel.
>>
>> **Returns**
>>> An integer for the gain.
>>
>> **Return type**
>>> int

_StreamThreading() → dict[int, threading.Thread]

> Opens a save file, then creates a thread for each device to stream and write data from.
>
>> **Returns**
>>> Dictionary with keys as the device number and values as the started Thread.
>>
>> **Return type**
>>> dict[int,Thread]

_StreamUntilStop(*pod:* Pod8401HR, *file: IOBase | EdfWriter*, *sampleRate: int*, *devNum: int*) → None

> Streams data from a POD device and saves data to file. Stops looking when a stop stream command is read. Calculates average time difference across multiple packets to collect a continuous time series data.
>
>> **Parameters**
>>
>> - **pod** (*POD_8206HR*) – POD device to read from.
>> - **file** (*IOBase | EdfWriter*) – open file.
>> - **sampleRate** (*int*) – Integer sample rate in Hz.

static _WriteDataToFile_EDF(*file: EdfWriter*, *data: list[numpy.ndarray]*)

> Writes data to an open EDF file.
>
>> **Parameters**
>>
>> - **file** (*EdfWriter*) – opened EDF file.
>> - **data** (*list[np.ndarray]*) – List with one item for each channel.

static _WriteDataToFile_TXT(*file: IOBase*, *data: list[numpy.ndarray]*, *t: ndarray*)

   Writes data to an open text file.

   > **Parameters**
   >
   > > - **file** (`IOBase`) – opened write file.
   > > - **data** (`list[np.ndarray]`) – List with one item for each channel.
   > > - **t** (`np.ndarray`) – list with the time stamps (in seconds).

Setup.SetupOneDevice.Setup_8480SC module

class Setup.SetupOneDevice.Setup_8480SC.Setup8480SC

   Bases: [`SetupInterface`](#)

   Setup_8480SC provides the setup functions for an 8480-SC POD device.

   _streamMode

   > Set to True when the user wants to start streaming data from an 8480 POD device, False otherwise.
   >
   > > **Type**
   > > bool

   static GetDeviceName() → str

   > Returns the name of the POD device.
   >
   > > **Returns**
   > > String of _NAME.
   > >
   > > **Return type**
   > > str

   static GetSupportedFileExtensions() → list[str]

   > Returns a list containing valid file extensions.
   >
   > > **Returns**
   > > List of string file extensions.
   > >
   > > **Return type**
   > > list[str]

   StopStream() → None

   > Update the state flag to signal to stop streaming data.

   static _ChoosePeriod() → tuple[int]

   > Asks the user an input value for Stimulus Period, which is then seperated into Period_ms and Period_us. Seperation is required because the 'SET STIMULUS' requires 7 items in payload, and the second and third items is the Period_ms and Period_us.
   >
   > > **Returns**
   > > Formatted period into millisecs and microsecs.
   > >
   > > **Return type**
   > > tuple[int]

   static _ChooseRepeat() → int

   > Asks the user to input a value for the number of stimulus pulses to perform.
   >
   > > **Returns**
   > > representing repeat count for command 'SET STIMULUS'.

> **Return type**
> > int

static _ChooseStimulusConfig() → int

> Asks the user to input values for Config format of Stimulus
>
> > **Returns**
> > > Formatted Stimulus Config value.
> >
> > **Return type**
> > > (int)

static _ChooseSyncConfig() → int

> Asks the user to input values for Sync Config bits.
>
> > **Returns**
> > > Value calculated from the input bits, this value would be given as payload.
> >
> > **Return type**
> > > int

static _ChooseWidth() → tuple[int]

> Asks the user an input value for Stimulus width, which is then seperated into width_ms and width_us. Seperation is required because the 'SET STIMULUS' requires 7 items in payload, and the fourth and fifth items is the width_ms and width_us.
>
> > **Returns**
> > > Formatted given width into millisecs and microsecs
> >
> > **Return type**
> > > tuple[int]

_ConnectPODdevice(*deviceNum: int*, *deviceParams:* Params8480SC) → bool

> Creates a POD_8206HR object and write the setup parameters to it.
>
> > **Parameters**
> > - **deviceNum** (`int`) – Integer of the device's number.
> > - **deviceParams** (`Params_8480SC`) – Device parameters.
>
> > **Returns**
> > > True if connection was successful, false otherwise.
> >
> > **Return type**
> > > bool

_GetPODdeviceParameterTable() → Texttable

> Builds a table containing the parameters for all POD devices.
>
> > **Returns**
> > > Texttable containing all parameters.
> >
> > **Return type**
> > > Texttable

_GetParam_onePODdevice(*forbiddenNames: list[str] = []*) → *Params8480SC*

> Asks the user to input all the device parameters.
>
> > **Parameters**
> > > **forbiddenNames** (`list[str]`) – List of port names already used by other devices.

Returns
Device parameters.

Return type
Params_8480SC

_OpenSaveFile_EDF(*fname: str*, *devNum: int*)

EDF files are not supported for 8480 POD devices. Overwrites the parent's method, which would open an EDF file and write the header.

Parameters

- **fname** (`str`) – String filename. Not used.

- **devNum** (`int`) – Integer device number. Not used.

Raises
Exception – EDF filetype is not supported for 8480 POD devices.

_OpenSaveFile_TXT(*fname: str*) → IOBase

Opens a text file and writes the column names. Writes the current date/time at the top of the txt file.

Parameters
fname (`str`) – String filename.

Returns
Opened file.

Return type
IOBase

_StreamThreading() → dict[int, threading.Thread]

Opens a save file, then creates a thread for each device to stream and write data from.

Returns
Dictionary with keys as the device number and values as the started Thread.

Return type
dict[int,Thread]

_StreamUntilStop(*pod:* Pod8480SC, *file: IOBase*) → None

Saves a log of all packets recieved from the 8480 POD device until the user decides to stop streaming.

Parameters

- **pod** (`POD_8480`) – POD device to read from.

- **file** (`IOBase`) – Opened text file to save data to.

static _TtlSetup() → int

Asks the user to input values for Config format values of TTL Setup.

Returns
Formatted TTL Setup config value.

Setup.SetupOneDevice.Setup_PodInterface module

class Setup.SetupOneDevice.Setup_PodInterface.**SetupInterface**

    Bases: `object`

    Setup_Interface provides the basic interface of required methods for subclasses to implement. SetupPodDevices.py is designed to handle any of these children.

    _podDevices

        Instance-level dictionary of pod device objects. MUST have keys as device number.

        Type

            dict[int,POD_Basics]

    _podParametersDict

        Instance-level dictionary of device information. MUST have keys as device number.

        Type

            dict[int,Params_Interface]

    _saveFileName

        Instance-level string filename: <path>/file.ext. The device name and number will be appended to the filename.

        Type

            str

    AddPODdevice() → None

        Asks the user for the parameters for the new device. A new device number is generated.

    AreDeviceParamsValid(*paramDict: None | dict[int,* PodApi.Parameters.ParamsBasic.Params*]*)

        Checks if the parameters dictionary is valid.

        Parameters

            paramDict (`None | dict[int,Params_Interface]`) – Dictionary of parameters for all POD devices.

        Raises

            • **Exception** – Parameters must be contained in a dictionary.

            • **Exception** – Device keys must be integer type.

            • **Exception** – Device parameters must be dictionary type.

            • **Exception** – Device parameters dictionary is empty.

    ConnectAllPODdevices() → bool

        Connects all setup POD devices.

        Returns

            True if all devices are successfully connected, false otherwise.

        Return type

            bool

    DisplayPODdeviceParameters() → None

        Display all the pod device parameters in a table.

static GetDeviceName() → str

> returns the name of the POD device.
>
> > **Returns**
> >   GENERIC.
> >
> > **Return type**
> >   str

GetPODparametersInit() → str

> Gets a dictionary whose keys are the device number and the value is the device parameters dict. :returns: String representation of a dictionary of POD device parameters. The keys are the device number. :rtype: str

GetSaveFileName() → str

> Gets the path and filename where streaming data is saved to.
>
> > **Returns**
> >   String of the save file name and path (_saveFileName).
> >
> > **Return type**
> >   str

static GetSupportedFileExtensions() → list[str]

> Returns a list containing valid file extensions.
>
> > **Returns**
> >   List of string file extensions.
> >
> > **Return type**
> >   list[str]

PrintSaveFile() → None

> Prints the file path and name that data is saved to. Note that the device name and number will be appended to the end of the filename.

SetupFileName(*fileName: str | None = None*) → None

> Gets the file path and name to save streaming data to. Note that the device name and number will be appended to the end of the filename.
>
> > **Parameters**
> >   **fileName** (`str | None, optional`) – Name and path of the file, if known. Defaults to None.

SetupPODparameters(*podParametersDict: dict[int,* PodApi.Parameters.ParamsBasic.Params*] | None = None*) → None

> Sets the parameters for the POD devices.
>
> > **Parameters**
> >   **podParametersDict** (`dict[int,Params_Interface] | None, optional`) – dictionary of the device parameters for all devices. Defaults to None.

StopStream() → None

> Write a command to stop streaming data to all POD devices.

Stream() → dict[int, threading.Thread]

> Tests that all devices are connected then starts streaming data.
>
> > **Raises**
> >   **Exception** – Test connection failed.

> **Returns**
>> Dictionary with integer device number keys and Thread values.
>
> **Return type**
>> dict[int,Thread]

ValidateParams() → None

> Displays a table of the parameters of all devices, then asks the user if everything is correct. The user can then edit the parameters of a device.

_BuildFileName(*devNum: int*) → str

> Appends the device name and number to the end of the file name.
>
> **Parameters**
>> **devNum** (`int`) – Integer of the device number.
>
> **Returns**
>> String file name.
>
> **Return type**
>> str

static _ChoosePort(*forbidden: list[str] = []*) → str

> Systems checks user's Operating System, and chooses ports accordingly.
>
> **Parameters**
>> **forbidden** (`list[str], optional`) – List of port names that are already used. Defaults to [].
>
> **Returns**
>> String name of the port.
>
> **Return type**
>> str

_ConnectPODdevice(*deviceNum: int*, *deviceParams:* Params) → bool

> Creates a POD device object and write the setup parameters to it.
>
> **Parameters**
>> - **deviceNum** (`int`) – Integer of the device's number.
>> - **deviceParams** (`Params_Interface`) – Device parameters.
>
> **Returns**
>> True if connection was successful, false otherwise.
>
> **Return type**
>> bool

_DisconnectAllPODdevices() → None

> Disconnects all POD devices by deleted all POD obejcts.

_EditParams() → None

> Asks the user which device to edit, and then asks them to re-input the device parameters.

_GetForbiddenNames(*exclude: str | None = None*) → list[str]

> Generates a list of port names used by the active pod devices. There is an option to exclude an additional name from the list.
>
> **Parameters**
>> **exclude** (`str | None, optional`) – String port name to exclude from the returned list. Defaults to None.

>> **Returns**
>>> List of string names of ports in use.

>> **Return type**
>>> list[str]

_GetPODdeviceParameterTable() → Texttable

> Builds a table containing the parameters for all POD devices.

>> **Returns**
>>> Table containing all parameters.

>> **Return type**
>>> Texttable

_GetParam_allPODdevices() → dict[int, *PodApi.Parameters.ParamsBasic.Params*]

> First gets the number of POD devices, then asks the user for the information for each device.

>> **Returns**
>>> Dictionary with device numbers for keys and parameters for values.

>> **Return type**
>>> dict[int,Params_Interface]

_GetParam_onePODdevice(*forbiddenNames: list[str] = []*) → *Params*

> Asks the user to input all the device parameters.

>> **Parameters**
>>> **forbiddenNames** (`list[str]`) – List of port names already used by other devices. Defaults to [].

>> **Returns**
>>> Device parameters.

>> **Return type**
>>> Params_Interface

static _GetTimeHeader_forTXT() → str

> Builds a string containing the current date and time to be written to the text file header.

>> **Returns**
>>> String containing the date and time. Each line begins with '#' and ends with a newline.

>> **Return type**
>>> str

_OpenSaveFile(*devNum: int*) → IOBase | EdfWriter

> Opens a save file for a given device.

>> **Parameters**
>>> **devNum** (`int`) – Integer of the device number.

>> **Returns**
>>> Opened file. IOBase for a text file, or EdfWriter for EDF file.

>> **Return type**
>>> IOBase | EdfWriter

_OpenSaveFile_EDF(*fname: str*, *devNum: int*) → EdfWriter

> Opens EDF file and write header.

>> **Parameters**

- **fname** (`str`) – String filename.

- **devNum** (`int`) – Integer device number.

> **Returns**
> Opened file.

> **Return type**
> EdfWriter

_OpenSaveFile_TXT(*fname: str*) → IOBase

Opens a text file and writes the column names. Writes the current date/time at the top of the txt file.

> **Parameters**
> **fname** (`str`) – String filename.

> **Returns**
> Opened file.

> **Return type**
> IOBase

static _PrintDeviceNumber(*num: int*) → None

Prints a title with the device number.

> **Parameters**
> **num** (`int`) – Integer of the device number.

_RemoveDevice() → None

Asks the user for a device number to remove, then deletes that device. This will only remove a device if there are more than one options.

_SelectDeviceFromDict(*action: str*) → int

Asks the user to select a valid device number. The input must be an integer number of an existing device.

> **Parameters**
> **action** (`str`) – Description of the action to be performed on the device.

> **Returns**
> Integer for the device number.

> **Return type**
> int

static _SetNumberOfDevices(*name: str*) → int

Asks the user for how many devices they want to setup.

> **Parameters**
> **name** (`str`) – Name of the POD device type.

> **Returns**
> Integer number of POD devices desired by the user.

> **Return type**
> int

_StreamThreading() → dict[int, threading.Thread]

Opens a save file, then creates a thread for each device to stream and write data from.

> **Returns**
> Dictionary with keys as the device number and values as the started Thread.

Return type
dict[int,Thread]

static _TestDeviceConnection(*pod:* Pod) → bool

Tests if a POD device can be read from or written. Sends a PING command.

Parameters
pod (*POD_Basics*) – POD device to read to and write from.

Returns
True for successful connection, false otherwise.

Return type
bool

_TestDeviceConnection_All() → bool

Tests the connection of all setup POD devices.

Returns
True when all devices are successfully connected, false otherwise

Return type
bool

static _uV(*voltage: float | int*) → float

Converts volts to microVolts, rounded to 6 decimal places.

Parameters
voltage (*float | int*) – number of volts.

Returns
voltage in of uV.

Return type
float

Module contents

1.2.2 Module contents

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

Symbols

V

ValidateParams() (*Setup.SetupOneDevice.Setup_PodInterface.SetupInterface method*), [74](#)

W

week (*PodApi.Parameters.Params8229.Params8229 attribute*), [47](#)

Write() (*PodApi.Devices.SerialPorts.SerialComm.PortIO method*), [8](#)

WritePacket() (*PodApi.Devices.BasicPodProtocol.Pod method*), [11](#)

WritePacket() (*PodApi.Devices.PodDevice_8229.Pod8229 method*), [16](#)

WritePacket() (*PodApi.Devices.PodDevice_8401HR.Pod8401HR method*), [22](#)

WritePacket() (*PodApi.Devices.PodDevice_8480SC.Pod8480SC method*), [25](#)

WriteRead() (*PodApi.Devices.BasicPodProtocol.Pod method*), [11](#)