Pykarbon

Release 1.1.0

Jul 06, 2023

Contents

The Pykarbon module provides a set of tools for interfacing with hardware devices on OnLogic's 'Karbon' series rugged PCs. These interfaces include the onboard CAN bus, Digital IO, automotive features, and a few other hardware devices. There are several benefits to leveraging Pykarbon:

- You can start testing and developing right out of the box – setting up a Karbon with Pykarbon and sending some test messages over CAN takes only a couple of minutes.
- High level interface offers powerful tools that can be immediately applied to a broad variety of problems.
- Low level hardware access layer gives developers granular control when they need it.
- Abstracts complex hardware controls into simplified interfaces.
- Free and open; Pykarbon can be downloaded, modified, and used in any application.

The ultimate goal of this package is to provide a simple, but powerful, base platform that will allow for the quick and easy integration of a Karbon system into a variety of applications.

To get started, see the rest of the documentation:

CHAPTER 1

Intro

1.1 Pykarbon: Hardware Made Possible

The Pykarbon module provides a set of tools for interfacing with the hardware devices on OnLogics's 'Karbon' series industrial PCs. These interfaces include the onboard CAN bus, Digital IO, and a few other hardware devices.

The goal of this package is to provide a simple, but powerful, base platform that will allow you to quickly and easily integrate a Karbon into your own application.

1.2 Automatic Operation

Getting started with Pykarbon doesn't take a lot of code; many use-cases can be handled in only *five lines:*

```
import pykarbon.pykarbon as pk

with pk.Karbon() as dev:
  dev.write() # Your commands here
  dev.can.data
  dev.terminal.data
```

This command automatically discovers and prepares your system's hardware interfaces by performing several tasks:

- Discovers the two COM ports owned by your Karbon's microcontroller
- Opens a connection with both ports, and acquires a lock
- Starts background threads for monitoring and processing incoming data
- Attempts to detect the baudrate of connected CAN devices
- Reads configuration information about your microcontroller

All of that setup happens in the background and, once complete, you are free to access configuration settings, the CAN interface, digital IO, and so on. When the context session ends, the connection will be automatically cleaned up, and the interfaces closed.

1.3 The Toolbox

Pykarbon offers several modules that grant different levels of access to the hardware. These include:

1.3.1 pykarbon

This is the highest level module. Its single 'Karbon' class creates a control object that you can use to write to both the Karbon system's serial terminal and CAN port. It also starts background monitoring on both of those ports, and allows you to access any data streamed to either port from their respective data queues. In most cases, the best way to use this module is as a command line controller for your hardware.

1.3.2 core

This core module provides access to basic commands in a blocking, low-overhead, way. It is able sniff packets on the CAN bus, read back user configuration information, and toggle digital IO.

1.3.3 terminal & can

The terminal and can modules hold the tools for creating, controlling and monitoring their respective virtual terminals. They also allow you to disable automatic background monitoring, and take control of reads and writes in a very direct way.

These modules also provide a "reaction" class that may be subclassed in order to generate your own, custom, responses to bus events. Reactions have the built-in ability to respond over the port they are using, and they will automatically execute your self-defined callback function. This means that you can register *any python function* to be called when a certain message is detected on the bus.

1.3.4 hardware

The low-level access layer takes care of discovering and claiming ports, and offers an object with read and write methods. Developers that want fine-grain control over how their system operates, or who just don't need any bells and whistles, can use this module as a launching point.

CHAPTER 2

Quickstart

2.1 Prerequisites

Before running through this guide, make sure you have these things ready:

- *A Karbon_ series computer.*
- Python 3 downloaded and installed on the system.
  - Be sure to add Python to your path if installing on Windows
- An internet connection on the target Karbon.

2.2 Installation

Installing Pykarbon is as simple as opening up a terminal and running:

```
> pip install pykarbon --user
```

or, in Ubuntu:

```
$ python3 -m pip install pykarbon --user
```

2.3 Usage

Launch a Python REPL in your terminal with `python` in Windows or `sudo python3` in Ubuntu. Now you're just a few commands away from talking with your hardware:

```
# Import the module
import pykarbon.core as pkcore
```

(continues on next page)

(continued from previous page)

```
# Open a terminal session and print out your firmware version
with pkcore.Terminal() as dev:
    dev.print_command('version')

# Open an can session and start listening for packets
with pkcore.Can() as dev:
    dev.sniff()
```

In the example above, we used *core* as a simple, and basic, interface tool. More advanced and useful features can be found by using the dedicated *can* and *terminal* modules. You can read more about them, and see some examples in the API.

CHAPTER 3

API

3.1 pykarbon

Pykarbon is designed to wrap interfacing with the microntroller on the Karbon series in an object, with the explicit goal of being able to treat this object in a more pythonic manner. In this manner most serial interactions with carbon are greatly simplified, and are more versatile.

Note: *pykarbon.pykarbon*'s only class, Karbon, will claim and use both microntroller interfaces. Addtionally, the full featuresets of both *pykarbon.terminal.Session* and *pykarbon.can.Session* are accessable via *Karbon.terminal* and *Karbon.can* respectively

---

Example

```
import pykarbon.pykarbon as pk

with pk.Karbon() as dev:
    dev.write(0x123, 0x11223344)  # Send a message over the can interface
    dev.can.data  # List of receive can messages

    # Karbon.write uses input length and types to determine what action to perform!
    dev.write(0, '1')  # Set digital output zero high
```

class pykarbon.pykarbon.**Karbon**(*automon=True*, *timeout=0.01*, *baudrate=None*)

Handles interactions with both virtual serial ports.

When initialized, this will scan the systems COM ports for the expected hardware/product id. The ports reporting this id are then attached to the class for easy recall and access.

can

A pykarbon.can session object – used to interface with karbon can bus

terminal
a pykarbon.terminal session object – used to interface with karbon terminal.

autobaud(*baudrate: int*) → str
Autodetect the bus baudrate

If the passed argument 'baudrate' is None, the baudrate will be autodetected, otherwise, the bus baudrate will be set to the passed value.

Parameters **baudrate** – The baudrate of the bus in thousands. Set to 'None' to autodetect

Returns The discovered or set baudrate

close()
Close both ports

open()
Opens both ports: Only needs to be called if 'close' has been called

read(*port_name='terminal'*, *print_output=False*)
Get the next line sent from a port

Parameters

- **port_name** (*str, optional*) – Will read from CAN if 'can' is in the port name. Reads from the terminal port by default.
- **print_output** (*bool, optional*) – Set to false to not print read line

Returns Raw string of the line read from the port

show_info()
Updates and prints configuration information

write(**args*)
Takes a command input and interperets it into a serial task:

If given two integer args, it will send a CAN message. If given two string args, it will set a terminal parameter. If given an integer as the first arg, and a str as the second arg, it will attempt to set the corrospponding digital output.

A single string argument will simply be sent to the terminal.

Returns None

3.2 core

A set of core functions for when you don't need anything fancy.

Examples

Print firmware version:

```
import pykarbon.core as pkcore

with pkcore.Terminal() as dev:
    dev.print_command('version')
```

Read can messages:

```
import pykarbon.core as pkcore

with pkcore.Can() as dev:
    dev.sniff()
```

Note: The two exposed classes, *Terminal* and *Can*, may be used within context managers, but are also able to use inhereted methods from *pykarbon.hardware.Interface* to claim and release ports. This can be usefull when you want to use a port for the duration of your application.

Example:

```
import pykarbon.core as pkcore:

dev = pkcore.Terminal()
dev.claim()

dev.set_high(0)  # Set digital output zero high

# Some code here, occasionally using device

dev.set_low(0)  # Return digital output zero to low state
dev.release()
```

class pykarbon.core.**Can**

Exposes methods for blocking read/write control of the serial terminal

pykarbon.core.Can is a subclass of pykarbon.hardware.Interface('can', timeout=.01). It does not log and monitor bus events in background, and logging messages is a blocking task. However, it can be very usefull when you are simply trying to diagnose or monitor bus messages.

Parameters **messages** (*list*) – List of read messages

send(*data_id*, *data*, *length=None*)

Send a can message.

Message properties will be inferred from id and data.

Parameters

- **data_id** (*int*) – Hex value data id. If it is larger that 0x7FF, the message will be transmitted as CAN 2.0B (extended) format
- **data** (*int*) – Hex valued data. Message length will be dervived from this. If the data is None, a remote request frame will be sent instead.
- **length** (*int*) – Length in bytes of expected message, should be specified for remote.

sniff()

Read messages and print, until stopped. Messages will be saved

Additionally logs time delta between received messages, alongside id and data. Outputs are additionally saved in dictionary format to self.messages, in the order that they are recieved.

class pykarbon.core.**Terminal**(*timeout=0.05*, *max_poll=100*)

Exposes methods for blocking read/write control of the serial terminal

pykarbon.core.Terminal is a subclass of pykarbon.hardware.Interface('terminal', timeout=.01). It uses a simplified, blocking, method for reading device information and setting digital output states. Digital input events will not be automatically logged, so a polling approach should be implemented while waiting for an input event.

Parameters

- **timeout** (*float, optional*) – The maximum amount of time, in seconds, that functions will block while waiting for a response.
- **max_poll** (*int, optional*) – The hard maximum on number of times the system will poll with receiving any response. Puts a hard-cap on timeout.

voltage
The last read system voltage, initialized to 0

Type float

cleanout()
Flush the input buffer, discarding the contents

command(*command*)
Writes a literal command and returns the result

Parameters **command** (*str*) – String that will be written to the serial terminal

get_state(*pin*)
Returns the current state of a given digital input, updates only that state.

Parameters **pin** (*int*) – 0-3, the digital input to read

grepall(*expression*, *default=None*)
Calls readall and returns the first output of a re.search of the output.

Parameters

- **expression** (*str*) – The regular expression to match against
- **default** (*optional*) – What to return if findall fails, default None

input_states()
Returns the current state of every single input, and updates all stored states

print_command(*command*)
Calls 'command' and prints the output

readall(*container*)
Read lines until they stop coming, and save them into a container

Parameters **container** (*list*) – List that each line of response will be appended to. It is both passed in and returned so it can be pre-loaded.

set_high(*pin*)
Sets the given digital output high

Parameters **pin** (*int*) – 0-3, the index of digital output to set high.

set_low(*pin*)
Sets the given digital output low

Parameters **pin** (*int*) – 0-3, the index of digital output to set low.

update_voltage()
Reads updated voltage and parses it into a float

3.3 can

Tool for running a session with the can interface.

Example

```
import pykarbon.can as pkc
from time import sleep

with pkc.Session() as dev:
    dev.write(0x123, 0x11223344)  # Send a message

    sleep(5)  # Your code here!

    dev.storedata('can_messages')  # Save messages that we receive while we waited
```

Lets us autodetect the can bus baudrate, write data to the can bus, wait for some messages to be receive, and finally save those messages to can_messages.csv

class pykarbon.can.**Reactions**(*canwrite*, *data_id*, *action*, ***kwargs*)

A class for performing automated responses to certain can messages.

If the action returns a dict of hex id and data, then the reaction will automatically respond with this id and data. If the dict has 'None' for id, then the reaction will respond with the originating frame's id and then returned data.

Note: Example action response: {'id': 0x123, 'data': 0x11223344}

---

data_id
The can data id registered with this reaction

action
Function called by this reaction

remote_only
If the reaction will respond to non-remote request frames

run_in_background
If reaction will run as background thread

auto_response
If reaction will automatically reply

canwrite
Helper to write out can messages

bgstart(*hex_data*)
Call start as a background thread

Returns The thread of the background action

respond(*returned_data*)
Automatically respond to frames, if requested

Parameters **returned_data** – A dict of id and data. If None, no response will be sent

start(*hex_data*)
Run the action in a blocking manner

Parameters **hex_data** – The hex data of the message that invoked this reaction. Should be the string 'remote' for remote frames.

class pykarbon.can.**Session**(*baudrate='autobaud'*, *timeout=0.01*, *automon=True*, *reaction_poll_delay=0.01*)

Attaches to CAN serial port and allows reading/writing from the port.

Automatically performs port discovery on linux and windows. Then is able to take ownership of a port and perform read/write operations. Also offers an intelligent method of sending can messages that will automatically determine frame format, type, and data length based only on the message id and data.

There is additional support for registering a function to certain can data ids. When the interface receives a registered message, it will call the function and send the returned data. This features requires running the session with automonitoring enabled.

By default, the session will also try to automatically discover the bus baudrate.

Parameters

- **baudrate** (*int/str, optional*) – *None* -> Disable setting baudrate altogther (use mcu stored value)

  'autobaud' -> Attempt to automatically detect baudrate

  100 - 1000 -> Set the baudrate to the input value, in thousands
- **timeout** (*float, optional*) – Time until read/write attempts stop in seconds. (None disables)
- **automon** (*bool, optional*) – Automatically monitor incoming data in the background.
- **reaction_poll_delay** (*float, optional*) – Time between checking received data for a registered value. Decreasing this delay will consume more unused CPU time.

If the baudrate option is left blank, the device will instead attempt to automatically detect the baudrate of the can-bus. When 'automon' is set to 'True', this object will immediately attempt to claim the CAN connection that it discovers. Assuming the connection can be claimed, the session will then start monitoring all incoming data in the background.

This data is stored in the the session's 'data' attribute, and can be popped from the queue using the 'popdata' method. Additionally, the entire queue may be purged to a csv file using the 'storedata' method – it is good practice to occasionally purge the queue.

interface
pykarbon.hardware.Interface

pre_data
Data before it has been parsed by the registry service.

data
Queue for holding the data read from the port

isopen
Bool to indicate if the interface is connected

baudrate
Reports the discovered or set baudrate

registry
Dict of registered DIO states and function responses

bgmon
Thread object of the bus background monintor

autobaud(*baudrate: int*) → str
Autodetect the bus baudrate

If the passed argument 'baudrate' is None, the baudrate will be autodetected, otherwise, the bus baudrate will be set to the passed value.

When attempting to auto-detect baudrate, the system will time-out after 3.5 seconds.

Parameters **baudrate** – The baudrate of the bus in thousands. Set to 'None' to autodetect

Returns The discovered or set baudrate

bgmonitor()
Start monitoring the canbus in the background

Uses python threading module to start the monitoring process.

Returns The 'thread' object of this background process

check_action(*line*)
Check is message has an action attached, and execute if found

Parameters **line** – Can message formatted as [id] [data]

close()
Release the interface so that other session may interact with it

Any existing background monitor session will also be closed. If this session re-opens the connection, background monitoring will need to be manually restarted with the 'bgmonitor' method.

static format_message(*id*, *data*, ***kwargs*)
Takes an id and data and determines other message characteristics

When keyword arguments are left blank, this function will extrapolate the correct frame information based on the characteristics of the passed id and data. If desired, all of the automatically determined characteristics may be overwritten.

Parameters

- **data_id** – Data id of the message, in hex (0x123, '0x123', '123')
- **data** – Message data, in hex – if 'None', the device will send a remote frame. NOTE: Use string version of hex to send leading zeroes ('0x00C2' or '00C2')
- ****kwargs** – *format*: Use standard or extended frame data id ('std' or 'ext')

  length: Length of data to be transmitted, in bytes (11223344 -> 4)

  type: Type of frame ('remote' or 'data')

monitor()
Watches port for can data while connection is open.

The loop is predicated on the connection being open; closing the connection will stop the monitoring session.

Parameters **session** – A canbus session object

Returns The method used to stop monitoring. (str)

open()
Claim the interface (only one application may open the serial port)

popdata()
If there is data in the queue, pop an entry and return it.

Uses queue behavior, so data is returned with 'first in first out' logic

Returns String of the data read from the port. Returns empty string if the queue is empty

pushdata(*line: str*)
Add data to the end of the session queue.

NOTE: Strips EoL characters.

Parameters **line** – Data that will be pushed onto the queue

readline()
Reads a single line from the port, and stores the output in self.data

If no data is read from the port, then nothing is added to the data queue.

Returns The data read from the port

register(*data_id*, *action*, ***kwargs*)
Automatically perform action upon receiving data_id

Register an action that should be automatically performed when a certain data id is read. By default the action will be performed when the id is attached to any frame type, and the action's returned data will be checked – if the data can be formatted as a can message, it will automatically be transmitted as a reply.

Actions should be a python function, which will be automatically wrapped in a pykarbon.can.Reactions object by this function. When the passed action is called Reactions will try to pass it the hex id and data as the first and second positional arguments. If thrown a TypeError, it will call the action without any arguments.

Example

```
>>> Session.register(0x123, action)
```

Note: If the frame is a remote request frame, the passed data will be 'remote' instead of an int!

Parameters

- **data_id** – The hex data_id that the action will be registered to
- **action** – The python function that will be performed.
- **kwargs** – remote_only: Respond only to remote request frames (Default: False) run_in_background: Run action as background task (Default: True) auto_response: Automatically reply with returned message (Default: True)

Returns The 'Reaction' object that will be used in responses to this data_id

registry_service()
Check if receive line has a registered action.

If the receive line does have an action, perform it, and then move the data into the main data queue. Otherwise, just move the data.

send_can(*message*) → str
Transmits the passed message on the canbus

Parameters **message** – A dictionary containing the data required to build a can message

Returns The string version of the transmitted message

storedata(*filename: str*, *mode='a+'*)
Pops the entire queue and saves it to a csv.

This method clears the entire queue: once you have called it, all previously received data will no longer be stored in the sessions 'data' attribute. Instead, this data will now reside in the specified .csv file.

Each received can message has its own line of the format: id,data.

By default, if a file that already exists is specified, the data will append to the end of this file. This behavior can be changed by setting 'mode' to any standard 'file.write' mode.

Parameters

- **filename** – Name of file that will be created.
- **mode** (*str, optional*) – The file write mode to be used.

write(*can_id*, *data*)
Auto-format and transmit message

For the large majority of use cases, this is the simplest and best method to send a packet of data over the canbus. Only message id and the data need to specified as hex values. All other information about the packet will be extrapolated.

Parameters

- **can_id** – The hex id of the data
- **data** – The hex formatted data

pykarbon.can.**hardware_reference**(*device='K300'*)
Print useful hardware information about the device

Displays hardware information about the CAN device, such as pinouts. Then pinouts assume that the user is facing the front of the device, and that the fins are pointed up.

Parameters **device** (*str, optional*) – The karbon series being used. Defaults to the K300

pykarbon.can.**hexify**(*value*)
Takes variously formatted hex values and outputs them as a int

pykarbon.can.**stringify**(*value*)
Takes variously formatted hex values and outputs them in simple string format

3.4 terminal

Tools for sending commands to the microcontroller, as well as using the DIO

Example

```
import pykarbon.terminal as pkt

with pkt.Session() as dev:
    dev.update_info(print_info=True) # Update and print configuration info

    dev.set_do(0, True) # Set digital output zero high
```

This snippet will update and print the microntrollers configuration information, and then set digital output zero high.

class pykarbon.terminal.**Reactions**(*set_all_do*, *info*, *action*, ***kwargs*)
A class for performing automated responses to certain dio transitions.

If the action returns a list digital output states, then the reaction will set each of these states. If the action returns None, no digital outputs will be set.

Example

```
>>> ['0', '1', '1', '0'] # Example action response
```

Note: When manually building reactions, you will need to pass in a pointer to the set_all_do function of a claimed interface.

info
The input number and state that trigger this reaction

dio_state
Mask reaction to this dio state

action
Function called by this reaction

transition_only
If the reaction will respond to non-transition events

run_in_background
If reaction will run as background thread

auto_response
If reaction will automatically reply

set_do
Helper to set digital output state

bgstart(*current_state*)
Call start as a background thread

Returns The thread of the background action

respond(*returned_data*)
Automatically respond to frames, if requested

Parameters **returned_data** – A list of DIO states. If none, no states will be set

start(*current_state*)
Run the action in a blocking manner

Parameters **current_state** – The current state of the dio

class pykarbon.terminal.**Session**(*timeout=0.01*, *automon=True*)
Attaches to terminal serial port and allows reading/writing from the port.

Automatically performs port discovery on linux and windows. Then is able to take ownership of a port and perform read/write operations. Also offers a method for setting various mcu control properties.

There is additional support for registering a function to certain DIO states, or input pin transitions. When the interface receives a registered event, it will call the function and optionally set the digital outputs to the returned state. This features requires running the session with automonitoring enabled.

Digital IO events will be recorded in the data queues, while configuration information will overwite the 'info' dictionary.

Parameters

- **timeout** (*int, optional*) – Time until read attempts stop in seconds. (None disables)
- **automon** (*bool, optional*) – Automatically monitor incoming data in the background.

When 'automon' is set to 'True', this object will immediately attempt to claim the terminal connection that it discovers. Assuming the connection can be claimed, the session will then start monitoring all incoming data in the background.

This data is stored in the the session's 'data' attribute, and can be popped from the queue using the 'popdata' method. Additionally, the entire queue may be purged to a csv file using the 'storedata' method – it is good practice to occasionally purge the queue.

interface
pykarbon.hardware.Interface

pre_data
Data before it has been parsed by the registry service.

data
Queue for holding the data read from the port

isopen
Bool to indicate if the interface is connected

info
Dictionary of information about the configuration of the mcu.

registry
Dict of registered DIO states and function responses

bgmon
Thread object of the bus background monintor

bgmonitor()
Start monitoring the terminal in the background

Uses python threading module to start the monitoring process.

Returns The 'thread' object of this background process

check_action(*line*, *prev_line=None*)
Check is message has an action attached, and execute if found

Parameters

- **line** – Dio state formatted as '[0-1]{4} [0-1]{4}'
- **prev_line** – The previously known state of the bus

close()
Release the interface so that other session may interact with it

Any existing background monitor session will also be closed. If this session re-opens the connection, background monitoring will need to be manually restarted with the 'bgmonitor' method.

get_previous_state(*index=-1*)
Returns the previous state of the digital io

monitor()
Watches port for incoming data while connection is open.

The loop is predicated on the connection being open; closing the connection will stop the monitoring session.

Returns The method used to stop monitoring. (str)

open()
Claim the interface (only one application may open the serial port)

parse_line(*line*)
Parse a non-dio line into mcu configuration info

popdata()
If there is data in the queue, pop an entry and return it.

Uses queue behavior, so data is returned with 'first in first out' logic

Returns String of the data read from the port. Returns empty string if the queue is empty

print_info()
Prints out mcu configuration information

pushdata(*line: str*)
Add data to the end of the session queue.

NOTE: Does not push empty strings, and strips EoL characters.

Parameters **line** – Data that will be pushed onto the queue

readline()
Reads a single line from the port, and stores the output in self.data

If no data is read from the port, then nothing is added to the data queue.

Returns The data read from the port

register(*input_num*, *state*, *action*, ***kwargs*)
Automatically perform action upon receiving data_id

Register an action that should be automatically performed when a certain digital input state is read. By default, this action will only be performed when the digital input first transitions to a state – subsequent bus reads will be ignored:

Example

```
>>> Session.register(1, 'low', action)
```

Input 1 : 1 –> 0 (*Execute Action*)

Input 1 : 0 –> 0 (*Do nothing*)

Input 1 : 0 –> 1 (*Do nothing*)

Input 1 : 1 –> 0 (*Execute Action*)

Actions should be a python function, which will be automatically wrapped in a pykarbon.terminal.Reactions object by this function. When the passed action is called Reactions will try to pass it the current dio state as the first positional argument. If thrown a TypeError, it will call the action without any arguments.

There is addtional support for masking input events with a particular bus state. That is, if an input event occurs, but the bus does not match the state, the action will not be executed.

Example

```
>>> Session.register(1, 'high', action, dio_state='---0 ---1')
```

Input 1 : 0 –> 1, Bus State: 0011 1111 (*Do nothing*)

Input 1 : 1 –> 1, Bus State: 0000 1111 (*Do nothing*)

Input 1 : 1 –> 0, Bus State: 0000 1111 (*Do nothing*)

Input 1 : 0 –> 1, Bus State: 0000 1111 (*Execute Action*)

Note: Bus state format is digital output 0-4 space digital input 0-4. Dashes are 'don't care'

Parameters

- **dio_state** (*str*) – Shorthand for the state of the dio, a dash will ignore the value.
- **action** – The python function that will be performed.
- **kwargs** – transition_only: Act only when a state is true by transition (Default: True) dio_state: Mask performing action with dio state (Default: —- —-) run_in_background: Run action as background task (Default: True) auto_response: Automatically reply with returned message (Default: True)

Returns The 'Reaction' object that will be used in responses to this data_id

registry_service()
Check if receive line has a registered action.

If the receive line does have an action, perform it, and then move the data into the main data queue. Otherwise, just move the data.

set_all_do(*states*)
Sets all digital outputs based on a list of states

Parameters **states** (*list*) – A list of '1', '0', or '-' corrospponding to the state of each output. Note: A '-' will skip setting the corrosponding output

Example

```
>>> set_all_do(['0', '0', '0', '0'])  # turn all outputs off
```

set_do(*number*, *state*)
Set the state of a single digital output

Maps different input formats into a unified format, and then calls a write method that sets a single output.

Example

```
>>> set_do(0, True)
>>> set_do('two', 0)
```

set_param(*parameter: str*, *value: str*, *update=True*, *save_config=True*)
Sets a mcu configuration parameter

Parameters

- **parameter** – Paramter to change
- **value** – Parameter will be set to this value
- **update** – Call update info to reflect param changes

Returns one or zero to indicate sucess or failure

storedata (*filename: str*, *mode='a+'*)
Pops the entire queue and saves it to a csv.

This method clears the entire queue: once you have called it, all previously received data will no longer be stored in the sessions 'data' attribute. Instead, this data will now reside in the specified .csv file.

Each received dio event has its own line of the format: outputs,inputs.

By default, if a file that already exists is specified, the data will append to the end of this file. This behavior can be changed by setting 'mode' to any standard 'file.write' mode.

Parameters

- **filename** – Name of file that will be created.
- **mode** (*str, optional*) – The file write mode to be used.

update_info (*print_info=False*)
Request configuration information from MCU

Parameters **print** (*bool, optional*) – Print out info after update. (Default: False)

update_voltage (*timeout=2*)
Update the system input voltage

Parameters **timeout** (*optional*) – Set how long, in seconds to wait for voltage readout.

write (*command*)
Write an arbitrary string to the serial terminal

pykarbon.terminal.**hardware_reference** (*device='K300'*)
Print useful hardware information about the device

Displays hardware information about the DIO device, such as pinouts. The pinouts assume that the user is facing the front of the device, and that the fins are pointed up.

Parameters **device** (*str, optional*) – The karbon series being used. Defaults to the K300

3.5 i2c

Read and write using the MCUs i2c bus

This module allows you to read and write from any accessible device on the Karbon's I2C bus. It is not reccommend that you write to any of the existing devices unless you're absolutely certain of what you're doing.

Note: Existing devices:

K700: 0x21 – Onboard PoE 0x28 – Modbay PoE (Expansion ONLY) 0x40 – Humidity/Temperature Sensor 0x60 – Cryptographic Secure Element

K300: 0x20 – Onboard PoE 0x60 – Cryptographic Secure Element

Example

```
import pykarbon.i2c as pki

device_id = 0x21
register = 0x99
```

(continues on next page)

(continued from previous page)

```
dev = pki.Device(device_id)

val = dev.read(register)

print("Read {} from {}", val, register)
```

This will connect to the microcontroller via the serial interface, and then attempt to read the value of register 0x99 from the device at address 0x21.

class pykarbon.i2c.**Device**(*device_id*, *timeout=0.05*)

Opens an I2C device and exposes read/write commands for that device.

Device implements a simple blocking read/write methodology to talk with i2c devices. It not neccesary to close one device before opening and using another – however, you may only talk with one device at a time.

Parameters

- **device_id** (*int*) – The device address to read/write.
- **timeout** (*float, optional*) – The maximum amount of time, in seconds, that the function will block while waiting for a response.

read(*reg*, *length=1*)

Reads data from the selected register

Parameters

- **reg** (*int*) – The device register to read.
- **len** (*int*) – The number of bytes (uint8) to read.

Returns The hex value read from the device. val (str): String returned, if any

Return type val (int)

verified_write(*reg*, *data*)

Writes data to the selected register and verifies that it was written correctly

Parameters

- **reg** (*int*) – The device register to write
- **data** (*int*) – The data to write

Returns True if passed, False if failed

Return type success (bool)

write(*reg*, *data*)

Writes data to the selected register

Parameters

- **reg** (*int*) – The device register to write.
- **data** (*int*) – The data to write.

Returns None if successful, error response if failed

3.6 hardware

Discovery and control of hardware interfaces.

You can use this module when you want things to go as fast as possible, or when you just need serial read/write hooks for your own application.

Example

```
import pykarbon.hardware as pkh

with pkh.Interface('terminal') as dev:
    dev.cwrite('version')
    line = ''
    while not line:
        line = dev.cread()[0].strip('\n\r') # Strip termination, only reading one␣
↪line.

print(line)
```

This will discover and open a connection with the serial terminal interface on the MCU. It then asks the microntroller to report it's firmware version before polling for the response.

class pykarbon.hardware.**Hardware**
Has methods for performing various hardware tasks: includes port discovery, etc.

ports (
obj:'dict'): The two virtual serial ports enumerated by the MCU.

static check_port_kind(*port_name: str*) → str
Checks if port is used for CAN or as the terminal

Parameters **port_name** – The hardware device name.

Returns 'can' or 'terminal'

Return type The kind of port

get_ports() → dict
Scans system serial devices and returns the two Karbon serial interfaces

Returns A dictionary with the keys 'can' and 'terminal' assigned hardware port names.

class pykarbon.hardware.**Interface**(*port_name: str*, *timeout=0.01*)
Hardware subclass interface – controls interactions with the karbon serial interfaces.

port
The hardware name of the serial interface

ser
A serial object connection to the port.

sio
An io wrapper for the serial object.

multi_line_response
The number of lines returned when special commands are transmitted.

claim()
Claims the serial interface for this instance.

cread(*nlines=1*)
Reads n lines from the serial terminal.

Parameters **nlines** (*int, optional*) – How many lines to try and read

Returns The combined output of each requested read transaction

cwrite(*command: str*)
Writes a command string to the serial terminal and gets the response.

Parameters **command** – Action to be executed on the mcu

Returns None

release()
Release the interface, and allow other applications to use this port

Python Module Index

p

Index

M

O

P

R

S

T

U

V

W