Controller Area Network SDK API with CryptoCAN *for* Python on CANPico

canislabs.com
CTO blog kentindell.github.io
CANIS AUTOMOTIVE LABS

CryptoCAN class

Creating

```
cc = CryptoCAN(transmit=False,
               encryption_key=HSM.SHE_KEY_1,
               authentication_key=HSM.SHE_KEY_2,
               b_flag=0,
               anti_replay=False,
               alt_freshness=False,
               no_encrypt=False)
```

Note
- b_flag is measured from the least-significant CAN ID bit
- encryption_key key slot must be set in the HSM
- authentication_key key slot must be set in the HSM with the KEY_USAGE flag set
- HSM must be initialized

Status

```
cc.get_status()
```

Returns: Last return code of a CryptoCAN API call

CryptoCAN error codes

```
CC_ERC_NO_ERROR
CC_ERC_FIRST_FRAME
CC_ERC_VERIFY_FAILED
CC_ERC_SHE_ERROR
CC_ERC_FRAME_SYNC
CC_ERC_RANGE
CC_ERC_ID
CC_ERC_REPLAY
```

Sending

```
cc.create_frames(frame,
                 freshness=0)
```

Note
- frame is an instance of CANFrame
- freshness is a 31-bit unsigned integer
- Returns a list with A and B instances of CANFrame
- Raises an exception if an HSM error

Receiving

```
cc.receive_frame(frame,
                 freshness=0,
                 alt_freshness=0)
```

Note
- frame is an instance of CANFrame
- freshness is a 31-bit unsigned integer
- alt_freshness is a 31-bit unsigned integer
- Returns An instance of CANFrame if decoded OK
- Raises an exception if an HSM error

HSM class

Initializing

```
h = HSM(secret_key=None)
```

Note
- secret_key must be None or 16 bytes
- Factory reset if secret_key is not None

Key management

```
h.load_key(m1,
           m2,
           m3)
```

Note
- m1 must be 16 bytes of M1
- m2 must be 32 bytes of M2
- m3 must be 16 bytes of M3
- Returns a 2-tuple of:
  - M4 (32 bytes)
  - M5 (16 bytes)

```
h.load_plain_key(bytes)
h.export_ram_key()
```

Returns
A 5-tuple of:
- M1 (16 bytes)
- M2 (32 bytes)
- M3 (16 bytes)
- M4 (32 bytes)
- M5 (16 bytes)

Encryption

```
h.enc_ecb(plaintext,
          key=SHE_KEY_1)
```

Note
- plaintext must be 16 bytes
- Returns 16 bytes of ciphertext

```
h.dec_ecb(ciphertext,
          key=SHE_KEY_1)
```

Note
- ciphertext must be 16 bytes
- Returns 16 bytes of plaintext

Authentication

```
h.generate_mac(message,
               key=SHE_KEY_1)
```

Note
- message must be a multiple of 16 bytes
- Returns 16 bytes of MAC
- Key must not have VERIFY_ONLY property set
- Key must have KEY_USAGE property set

```
h.verify_mac(message,
             mac,
             mac_length=128,
             key=SHE_KEY_1)
```

Note
- message must be a multiple of 16 bytes
- mac must be 16 bytes
- mac_length must be between 0 and 128
- Key must have KEY_USAGE property set
- Returns True if MAC verified

SHE key numbers

```
SHE_SECRET_KEY
SHE_MASTER_ECU_KEY
SHE_BOOT_MAC_KEY
SHE_KEY_1
SHE_KEY_2
SHE_KEY_3
SHE_KEY_4
SHE_KEY_5
SHE_KEY_6
SHE_KEY_7
SHE_KEY_8
SHE_KEY_9
SHE_KEY_10
SHE_RESERVED
SHE_RAM_KEY
```

SHE error codes

```
SHE_ERC_NO_ERROR
SHE_ERC_SEQUENCE_ERROR
SHE_ERC_KEY_NOT_AVAILABLE
SHE_ERC_KEY_INVALID
SHE_ERC_KEY_EMPTY
SHE_ERC_MEMORY_FAILURE
SHE_ERC_BUSY
SHE_ERC_GENERAL_ERROR
SHE_ERC_KEY_WRITE_PROTECTED
SHE_ERC_KEY_UPDATE_ERROR
SHE_ERC_RNG_SEED
```

Random numbers

```
h.init_rng()
h.rnd(as_int=False)
```

Returns
- If as_int is True then returns a 128 bit integer
- If as_int is False then returns 16 bytes

```
h.extend_seed(entropy=bytes)
```

Debugging

```
h.backdoor_set_key( key=SHE_KEY_1,
                    key_value,
                    authentication_key=False,
                    store_keys=False)
```

Note
- key_value must be 16 bytes
- If authentication_key is True then sets KEY_USAGE property
- If store_keys is True then keys are written to NVRAM
- Backdoor API call only for development with a software HSM
- authentication_key must be:
  - True if key used for CryptoCAN MAC
  - False if key used for CryptoCAN encryption

v220819-1

CAN class

Starting controller

```
c = CAN(profile=CAN.CAN_BITRATE_500K_75,
        id_filters=None,
        hard_reset=False,
        brp=None,
        tseg1=10,
        tseg2=3,
        sjw=2,
        recv_errors=False,
        mode=CAN.NORMAL,
        tx_open_drain=False,
        reject_remote=True,
        rx_callback_fn=None,
        recv_overflows=False)
```

Note
- id_filters is a {integer: CANIDFilter} dictionary
- If brp is defined then profile is overridden
- rx_callback_fn is a Python function called from the ISR with the received CANFrame
- Set tx_open_drain=True if CANHack used

Bit rates

```
CAN_BITRATE_500K_75
CAN_BITRATE_250K_75
CAN_BITRATE_125K_75
CAN_BITRATE_1M_75
CAN_BITRATE_500K_50
CAN_BITRATE_250K_50
CAN_BITRATE_125K_50
CAN_BITRATE_1M_50
CAN_BITRATE_2M_50
CAN_BITRATE_4M_90
CAN_BITRATE_2_5M_75
CAN_BITRATE_2M_80
CAN_BITRATE_500K_875
CAN_BITRATE_250K_875
CAN_BITRATE_125K_875
CAN_BITRATE_1M_875
CAN.CAN_BITRATE_CUSTOM
```

Modes

```
CAN_MODE_NORMAL
CAN_MODE_LISTEN_ONLY
CAN_MODE_ACK_ONLY
CAN_MODE_OFFLINE
```

Triggers

```
c.set_trigger(on_error=False,
              on_canid=None,
              as_bytes=None,
              on_tx=False,
              on_rx=True)
```

Note
- as_bytes is None or type bytes
- on_can_id is None or CANID
- (as_bytes overrides)

```
c.clear_trigger()
c.pulse_trigger()
```

Time

```
c.get_time()
c.get_time_hz()
```

Sending frames

```
c.send_frame(frame, fifo=False)
c.send_frames([frame], fifo=False)
c.recv_tx_events(limit=CAN.CAN_TX_EVENT_FIFO_SIZE,
                 as_bytes=False)
```

Returns
- list of CANFrame instances sent, CANOverflow
- or bytes

```
c.recv_tx_events_pending()
```

Receiving frames

```
c.recv(limit=CAN.CAN_RX_FIFO_SIZE,
       as_bytes=False)
```

Returns
- list of CANFrame, CANError, CANOverflow
- or bytes

```
c.recv_pending()
```

Status

```
c.get_status()
```

Returns: 4-tuple of (bus off, error passive, TEC, REC)

CANIDFilter class

Making

```
filter = CANID(filter_str=None)
```

Note: filter_str is 11 or 29 '1', '0' or 'X' characters

CANError class

Reading

```
error.get_timestamp()
error.is_crc_error()
error.is_stuff_error()
error.is_form_error()
error.is_ack_error()
error.is_bit1_error()
error.is_bit0_error()
error.is_bus_off()
```

CANID class

Making

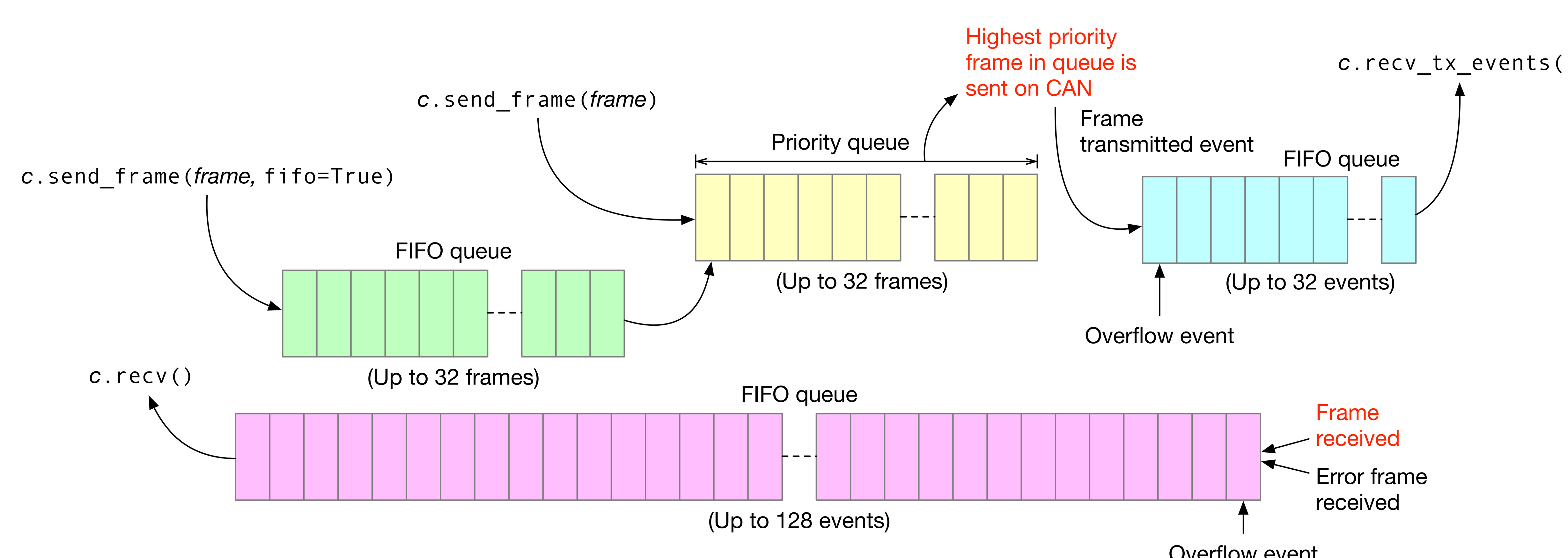
```
id = CANID(arbitration_id, extended=False)
```

Reading

```
id.is_extended()
id.get_arbitration_id()
id.get_id_filter()
```

Returns: a CANIDFilter matching the ID

Queues



CANFrame class

Making

```
frame = CANFrame(can_id,
                 data=None,
                 remote=False,
                 tag=0,
                 dlc=None)
```

Note
- can_id is a CANID
- data is of type bytes
- tag is an integer
- DLC defaults to length of data if dlc not set

```
CANFrame.from_bytes(bytes)
```

Note: returns a list of CANFrame

Reading

```
frame.get_data()
frame.get_dlc()
frame.get_tag()
frame.get_timestamp()
```

Note: returns None if not sent or received yet

```
frame.get_index()
```

Note: returns index of acceptance ID filter

```
frame.is_remote()
frame.is_extended()
frame.get_arbitration_id()
frame.get_canid()
```

Note: returns CANID

Printing

```
>>> f = CANFrame(CANID(0x123), data=b'hello')
>>> print(f)
CANFrame(CANID(id=S123), dlc=5, data=68656c6c6f)
```

S = 11-bit CAN ID
E = 29-bit CAN ID

* = 0 byte payload
R = remote frame

CANOverflow class

Reading

```
overflow.get_timestamp()
overflow.get_frame_cnt()
overflow.get_error_cnt()
```

CANHack class

Creating

```
ch = CANHack(bit_rate=500)
```

Note: bit_rate can be 500, 250 or 125

Frames

```
ch.set_frame(can_id=0x7ff,
             remote=False,
             extended=False,
             data=None,
             set_dlc=False,
             dlc=0
             second=False)
```

Note
- DLC set by default from data length
- data is 0..8 bytes
- second sets the Janus attack alternative value

```
ch.print_frame()
ch.send_frame(timeout=50000000,
              second=False,
              retries=0,
              repeat=1)
```

Bus Off and Error Passive attacks

```
ch.error_attack(repeat=2,
                timeout=50000000)
```

Note: Attacks the frame set with set_frame()

Freeze Doom Loop attack

```
ch.freeze_doom_loop_attack(repeat=2,
                           timeout=50000000)
```

Note: Attacks the frame set with set_frame()

Double Receive attack

```
ch.double_receive_attack(repeat=2,
                         timeout=50000000)
```

Note: Attacks the frame set with set_frame()

Diagnostics

```
ch.set_can_tx(recessive=False)
```

Returns: True if RX is recessive

```
ch.send_square_wave()
```

Note: Sends a square wave on TX for 160 bit times

```
ch.loopback()
```

Note: Waits for falling edge then transmits on TX what is read on RX for 160 bit times

Janus attack

```
ch.send_janus_frame(sync_time=50,
                    split_time=155,
                    timeout=50000000,
                    retries=0)
```

Note
- split_time default is 62.5% (bit time is 249)
- sync_time default is 20%
- timeout default is about 17 seconds

Spoof attacks

```
ch.spoof_frame(timeout=50000000,
               overwrite=False,
               sync_time=0,
               split_time=0,
               second=False,
               retries=0,
               loopback_offset=93)
```

Note
- if second is True then will spoof using a Janus frame
- if overwrite is True then sends an error passive spoof
- loopback_offset only used if overwrite is True