BLMC CAN Interface

Table of Contents

- Table of Contents
- Estimation of Data Throughput
- Communication Protocol
    - Messages sent from the board
        - Status Message
            - Error Codes
    - Messages sent to the board
        - Command Codes
    - CAN Receive Timeout
- CAN API on the LaunchPad
- Bit Timing Configuration
- Using CAN with Python
    - Setting up python-can for PCAN (Windows)
    - Setting up python-can for Socketcan (Linux)
    - Example Code
    - More Python Examples
    - References

Estimation of Data Throughput

Just a very rough estimation of the frequency with which we can push data through the CAN bus.

- Data on CAN bus is sent in "frames". Using standard CAN, one frame is 108 bit big and can carry 8 bytes of data.
- Maximum transfer rate is 1 Mbit/s = $1\,\text{Mbit/s} = \frac{1000000}{108} \approx 9259\,\text{frames/s}$.
- Sending one frame with 1 Mbit/s takes roughly **0.11 ms**
- So if we want to get data through at 1 kHz, **we must not send more than 9 frames per cycle**.
- Using one frame, we can send 4 bytes per motor which is exactly one IQ value.

We are sending the following stuff from board to PC:

- torque (1 frame)
- position (1 frame)
- velocity (1 frame)
- additional sensors (ADC 6) (1 frame)
- some status messages (1 frame with only one byte payload). Only send with 1 Hz, therefore not so relevant for the calculation.

At the same time we want to send from PC to board:

- torque reference (1 frame)
- gains and desired reference positions / velocities for PD controller on the card (1 frame each)
- some configuration stuff (turn on/off, ...), but this is typically only send once during initialization, so I omit this here.

Other Sensors, etc.

- OptoForce foot sensor (2 frames) -- *can run with up to 1 kHz but maybe less is enough?*
- some other stuff we don't know yet (4 frames?)

All in all (ignoring status and configuration): **7 frames**

So currently we can run one board (= two motors) and an OptoForce sensor at a **1kHz** rate, but there is not much room left for additional data. The PCI-CAN cards we are using have two CAN interfaces, though, which can be used in parallel (so this would kind of double your bandwidth).

Communication Protocol

Each frame has an 11bit identifier, which is used to distinguish different types of messages. In the following the preliminary protocol (which identifier is used for which data and how the data is stored inside the frame) is defined.

The 8 byte of data of a frame are split into a "lower" and a "higher" part of 4 bytes each. Following the names used in the CAN module they are referred to as MDL and MDH here.

Messages sent from the board

ID	Meaning	Size [Bytes]	Structure of the data
0x010	Status	1	see below

0x020	Current I_q (both motors)	8	MDL = Iq of motor 1 MDH = Iq of motor 2
0x030	Encoder Position (both motors)	8	MDL = Position of motor 1 MDH = Position of motor 2
0x040	Velocity (both motors)	8	MDL = Velocity of motor 1 MDH = Velocity of motor 2
0x050	Additional ADC inputs (e.g. used for potentiometers or other sensors)	8	MDL = ADC reading of input A6 MDH = ADC reading of input B6
0x060	Encoder Index Position *This is not sent with a fixed rate but whenever the encoder index is detected.*	5	MDL (4 Bytes) = Motor position at the index tick MDH (1 Byte) = ID of the motor

Please note that motor data (current, position, velocity) are send as Q24 values. Divide by 2^{24} to get the float representation.

The IDs 0x100 and 0x101 are by default used by the OptoForce sensor. To keep things simple, we should not use these IDs for other data, so we don't have to reconfigure the sensors.

Status Message

The status message contains one byte of data which is structured as follows:

Bit 5-7: Error Code	Bit 4: Flag motor 2 ready	Bit 3: Flag motor 2 enabled	Bit 2: Flag motor 1 ready	Bit 1: Flag motor 1 enabled	Bit 0: Flag system enabled

A motor is "ready" when the alignment process is finished.

Error Codes

Error Code	Meaning
0	No Error
1	Encoder Error
2	CAN Receive Timeout
3	Critical Motor Temperature (currently unused)
4	Some error in the Position Converter module
5	Position Rollover (position exceeded max value) *This error is by default disabled. To enable it, send a ENABLE_POS_ROLLOVER_ERROR=1 command.*
7	Other Error (to be used if we run out of free error codes)

Note that the error code for critical motor temperature won't be used for now, as we don't have any temperature sensing. As this might be added in the future, I already reserve an error code for this, though. The "Other Error" code is also not used so far. It may be useful if we run out of error codes at some point.

Messages sent to the board

ID	Name	Meaning	Structure of the data	Unit
0x000	CAN_ID_COMMANDS	Command	MDL = value MDH = command code (see below)	
0x005	CAN_ID_IqRef	Current I_q Reference	MDL = IqRef for motor 1 MDH = IqRef for motor 2	A
0x006	CAN_ID_KP	Gains for P controller	MDL = P gain for motor 1 MDH = P gain for motor 2	A / motor_rotations
0x007	CAN_ID_KD	Gains for D controller	MDL = D gain for motor 1 MDH = D gain for motor 2	A / kilo_rotations_per_minute
0x008	CAN_ID_POS_REF	Reference for P controller	MDL = Reference position for motor 1 MDH = Reference position for motor 2	motor_rotations
0x009	CAN_ID_VEL_REF	Reference for D controller	MDL = Reference velocity for motor 1 MDH = Reference velocity for motor 2	kilo_rotations_per_minute

Please note that current reference values have to be send as Q24 values. To convert float to Q24 multiply by 2^{24} and round to an integer.

Command Codes

A command message (ID = 0) consists of two parts. The high bytes (MDH) contain a code that is associated with a specific parameter (see table below) while the lower bytes (MDL) contain the value that is to be assigned to the parameter.

Code	Name	Meaning	Value	Default	Unit
1	ENABLE_SYS	Enable the system.	0/1	1	
2	ENABLE_MTR1	Enable Motor 1	0/1	0	
3	ENABLE_MTR2	Enable Motor 2	0/1	0	
4	ENABLE_VSPRING1	Enable virtual spring mode for motor 1	0/1	0	
5	ENABLE_VSPRING2	Enable virtual spring mode for motor 2	0/1	0	
12	SEND_CURRENT	Send motor currents via CAN	0/1	0	
13	SEND_POSITION	Send encoder positions via CAN	0/1	0	
14	SEND_VELOCITY	Send motor velocities via CAN	0/1	0	
15	SEND_ADC6	Send ADC inputs A6/B6 via CAN	0/1	0	
20	SEND_ALL	Disable/Enable all of the configurable CAN messages	0/1	0	
30	SET_CAN_RECV_TIMEOUT	Set CAN Receive Timeout in milliseconds. Set to zero to disable timeout.	uint32	0	
31	ENABLE_POS_ROLLOVER_ERROR	Enable the position rollover error	0/1	0	
40	CAN_CMD_P_CONTROLLER_LIMIT_IQ_MTR1	Set the current limit for the P controller of motor 1	IQ24	0	A
41	CAN_CMD_P_CONTROLLER_LIMIT_IQ_MTR2	Set the current limit for the P controller of motor 2	IQ24	0	A
42	CAN_CMD_D_CONTROLLER_LIMIT_IQ_MTR1	Set the current limit for the D controller of motor 1	IQ24	0	A
43	CAN_CMD_D_CONTROLLER_LIMIT_IQ_MTR2	Set the current limit for the D controller of motor 2	IQ24	0	A

Example: To enable motor 1, set MDH = 2 and MDL = 1.

Nomenclature: When refering to sending commands in this wiki or other parts of the documentation, the following nomenclature is used: NAME=value. Example: ENABLE_SYS=1 to enable the system.

CAN Receive Timeout

TODO Move this somewhere else? This is more about the board firmware and not about the CAN protocol.

The embedded software on the board provides a security feature that disables the motors in case the CAN connection is interrupted or the controller on the PC exits without properly shutting down the system. This is done by simply checking the time since the last current I_q reference was received and raising an error if it exceeds a specified timeout.

Note that by default, this feature is disabled! If you want to use it, you have to enable it by specifying a timeout duration greater than zero (see Command Codes above). There are a few consequences that have to be kept in mind:

- Before enabling the motors, set the current references to zero, otherwise the timeout may be trigger immediately when enabled. Note that this is good practice anyway as it clears potentially dangerous previous reference values.
- Current references have to be sent in a loop, even if the values do not change.

The timeout is only checked when motors are enabled and current references are not zero. This means that, as long as the current reference is zero, it is okay to enable the timeout during intialization even if current commands are not send immediately.

When the timeout is triggered, an error is set and the system is disabled. You can simply reenable it by sending (in this order) enable system command, a current=0 command and enable motor commands.

CAN API on the LaunchPad

InstaSPIN contains a driver for CAN, however, it does not contain a nice understandable API but only provides access to the registers. The eCAN module and the meaning of the registers are descriped in SPRUH18F, section 16.

Do **not** use SPRU074F (TMS320x281x Enhanced Controller Area Network (eCAN) Reference Guide). On the first glance it is the same as section 16 of SPRUH18F but there are actually some small differences which can be confusing (e.g. the formula for bit rate computation)

There is a custom API that builds upon the CAN driver and wraps all the nasty register reads/writes. It is still under developement and currently located in canapi.h/canapi.cpp in the dual_motor_torque_ctrl project.

Bit Timing Configuration

The bitrate for the CAN module on the LauchPad is set implicitly by configuring the bit timing. Therefore special care has to be taken when configuring these values.

The bit timing configuration is explained in SPRUH18F, section 16.10. The parameters to be set are `CANBTC.BRPREG`, `CANBTC.TSEG1REG` and `CANBTC.TSEG2REG`. The resulting bit rate is computed as

$$\text{bitrate} = \frac{\text{SYSCLOCK/2}}{(\text{BRPREG} + 1) \cdot (\text{TSEG1REG} + \text{TSEG2REG} + 3)}$$

There is a nice tool on http://www.bittiming.can-wiki.info that helps finding bit timing values for specific bit rates. Take care however, when using this tool:

1. As "Clock Rate" use SYSCLOCK/2 (typically 45 MHz for the F28069M)
2. Decrement all values by 1 when transferring them to your configuration! For some reason the values of the registers are incremented by one before they are used (i.e. BRP = BRPREG + 1, etc).

Example configuration to run CAN with 1MBit/s:

```
// 1 MBit/s
ECanaShadow.CANBTC.bit.BRPREG = 2;
ECanaShadow.CANBTC.bit.TSEG1REG = 11;
ECanaShadow.CANBTC.bit.TSEG2REG = 1;
ECanaShadow.CANBTC.bit.SJWREG = 1;
ECanaShadow.CANBTC.bit.SAM = 1;
ECanaRegs.CANBTC.all = ECanaShadow.CANBTC.all;
```

The OptoForce sensor is running its CAN interface with 1Mbit/s and it does not seem to be a way to change this. That is as long as board and sensor use the same bus, we have to use CAN with 1Mbit/s (which is what we want anyway).

Using CAN with Python

Just a brief note: I successfully used python-can to talk to the board from Python. However, **do not use the version from pypi.python.org** (version 1.5.0) when using a PCAN controller. There is a issue with the PCAN driver which is not yet resolved in 1.5.0. Use the newest code from the Bitbucket repository instead, there the issue is fixed.

Setting up python-can for PCAN (Windows)

From the README:

```
To use the PCAN-Basic API as the backend (which has only been tested
with Python 2.7):

1. Download the latest version of the `PCAN-Basic
API <http://www.peak-system.com/Downloads.76.0.html?>`__. (link broken  remove??)

Is this the right link? https://www.peak-system.com/fileadmin/media/files/pcan-basic.zip

2. Extract ``PCANBasic.dll`` from the Win32 subfolder of the archive or
the x64 subfolder depending on whether you have a 32-bit or 64-bit
installation of Python.

3. Copy ``PCANBasic.dll`` into the working directory where you will be
running your python script. There is probably a way to install the
dll properly, but I'm not certain how to do that.

Note that PCANBasic API timestamps count seconds from system startup. To
convert these to epoch times, the uptime library is used. If it is not
available, the times are returned as number of seconds from system
startup. To install the uptime library, run ``pip install uptime``.
```

The file **`can.ini`** (see README) should look as follows:

```
[default]
interface = pcan
channel = PCAN_USBBUS1
```

Setting up python-can for Socketcan (Linux)

For Linux it is much simpler, as the Linux Kernel already has a native CAN driver, called "socketcan". To set the default, create a file **`~/.can`** with the following content:

```
[default]
interface = socketcan
channel = can0
```

To enable CAN, call:

```
sudo ip link set can0 up type can bitrate 1000000
```

See the documentation for more details.

Example Code

The following script connects to the CAN bus and waits for messages. When a message is received, values about status and motors are updated and printed in a human readable way. Note that most of the code is only needed to convert the data to the correct format. Accessing the CAN bus is very simple 🙂

Listen to CAN bus and print motor status

```
"""
Small working example on how to access CAN bus.
"""
from __future__ import print_function
import os
import struct
import can

def send_one(bus):
        """Send a single message on the CAN bus."""
        # enable motor 1
    msg = can.Message(arbitration_id=0x000,
                      data=[0, 0, 0, 1, 0, 0, 0, 2],
                      extended_id=False)
    try:
        bus.send(msg)
        print("Message sent on {}".format(bus.channel_info))
    except can.CanError:
        print("Message NOT sent")

def q_bytes_to_value(data):
        """Convert bytes recevied via CAN to float value."""
        assert len(data) == 4
        # convert bytes to signed int (note: bytes in data are in reversed order)
        qval = struct.unpack("<i", data[::-1])[0]
        # convert q-value to human-readable float
        fval = float(qval) / 2.0**24
        return fval

class MotorData:
        current = 0
        position = 0
        velocity = 0
        def set_current_pos(self, data):
                self.current = q_bytes_to_value(data[0:4])
                self.position = q_bytes_to_value(data[4:8])
```

```
        def set_velcity(self, data):
                self.velocity = q_bytes_to_value(data)
        def to_string(self):
                return "Iq: {:.3f},  Pos: {:.3f},  Speed: {:.3f}".format(
                                self.current, self.position, self.velocity)

class Status:
        system_enabled = 0
        mtr1_enabled = 0
        mtr1_ready = 0
        mtr2_enabled = 0
        mtr2_ready = 0
        mtr1_overheat = 0
        mtr2_overheat = 0
        system_error = 0
        def set_status(self, data):
                status_code = data[0]
                self.system_enabled = status_code & 1
                self.mtr1_enabled = status_code & (1 << 1)
                self.mtr1_ready = status_code & (1 << 2)
                self.mtr2_enabled = status_code & (1 << 3)
                self.mtr2_enabled = status_code & (1 << 4)
                self.mtr1_overheat = status_code & (1 << 5)
                self.mtr2_overheat = status_code & (1 << 6)
                self.system_error = status_code & (1 << 7)
        def to_string(self):
                str_sys = "SYS: "
                str_sys += "E" if self.system_enabled else "D"
                str_sys += "!!!" if self.system_error else ""
                str_mtr1 = "MTR1: "
                str_mtr1 += "E" if self.mtr1_enabled else "D"
                str_mtr1 += "R" if self.mtr1_ready else "A"
                str_mtr2 = "MTR2: "
                str_mtr2 += "E" if self.mtr2_enabled else "D"
                str_mtr2 += "R" if self.mtr2_ready else "A"
                return "{} | {} | {}".format(str_sys, str_mtr1, str_mtr2)

if __name__ == "__main__":
        bus = can.interface.Bus(bitrate=250000)
        mtr1 = MotorData()
        mtr2 = MotorData()
        status = Status()
        # wait for messages and update data
        for msg in bus:
                arb_id = msg.arbitration_id
                if arb_id == 0x010:
                        status.set_status(msg.data)
                elif arb_id == 0x021:
                        mtr1.set_current_pos(msg.data)
                elif arb_id == 0x022:
                        mtr2.set_current_pos(msg.data)
                elif arb_id == 0x031:
                        mtr1.set_velcity(msg.data)
                elif arb_id == 0x032:
                        mtr2.set_velcity(msg.data)
                #os.system('cls' if os.name == 'nt' else 'clear')
                print("\r{}   \tMTR1: {}   \tMTR2: {}".format(
                        status.to_string(), mtr1.to_string(), mtr2.to_string()),)
```

More Python Examples

There is a repository with code to encode/decode CAN messages and more sophisticated examples (e.g. foot position control) on GitAMD: https://git-amd.tuebingen.mpg.de/fwidmaier/python-blmc

See the wiki there for documentation.

References

- python-can Documentation: https://python-can.readthedocs.io/en/latest/index.html (the example code on the start page should work out of the box)
- Download from Bitbucket: https://bitbucket.org/hardbyte/python-can/downloads
- About the issue with PCAN: https://bitbucket.org/hardbyte/python-can/issues/74/python-can-fails-when-using-pcan-at-newest
- Python package for using the LaunchPad via CAN: https://git-amd.tuebingen.mpg.de/fwidmaier/python-blmc