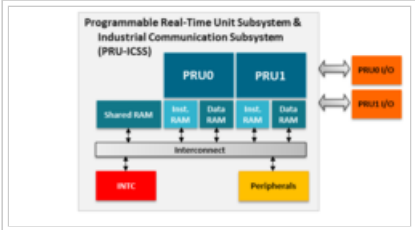
Introduction

PRUserial485 is a high-performance RS-485 serial interface developed for the embedded single board computer BeagleBone Black (BBB), an open-hardware platform based on Texas Instruments AM335x SoC. Running a Debian image, BBBs are the main distributed nodes for Sirius Control System.



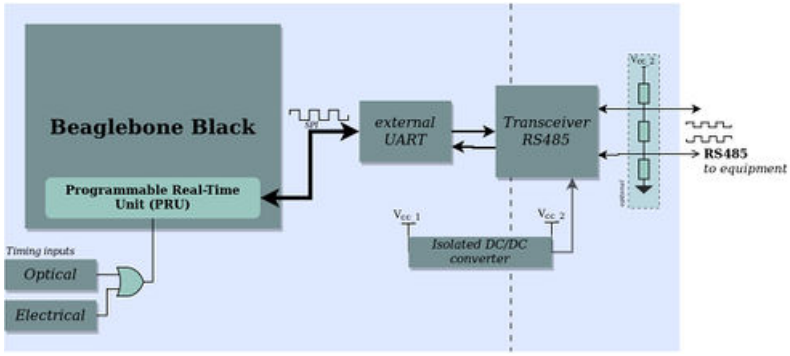
PRU-SS, from Texas Instruments

The PRUserial485, as its name may predict, has been designed to run at one of the two programmable real-time units (PRUs) available at the SoC. They are a dedicated RISC processor, which only executes a binary file at once, with a 200 MHz clock (5 ns instruction time).

Motivation and main features

This was the first and largest application for BeagleBone PRU. Reaching data rates up to 15 Mbps, it allows fast transfers of large data amount. Also, Sirius design requires equipment with great stability, reliability and synchronized operation.

For this purpose, the PRUserial485 has been designed mainly for Sirius Power Supplies, which are digitally controlled at 6 Mbps. This interface, which is very deterministic, also includes a timing input for triggering data sending during sync operation, such as booster ramping and soft orbit correction.



The steps below are mandatory to project development and understanding. They will be discussed in the next sections.

- Hardware interface
- BeagleBone pin configuration
- PRU firmware
- Carrier library
- Memory mapping

Synchronous operation mode: a particularity

"Sirius power supplies must perform synchronized adjusts during booster ramping, orbit correction, magnets cycling and migration mode. For booster energy ramping, for example, curves with 3920 points must be run at 2 Hz. For that purpose, a PRU serial mode was implemented to deliver serial messages with low jitter to power supplies' controllers. Concerning serial network, there are two main sync modes: after a triggering a timing pulse, PRU interface sends through RS-485:

- A broadcast command
- A command addressed to a specific device

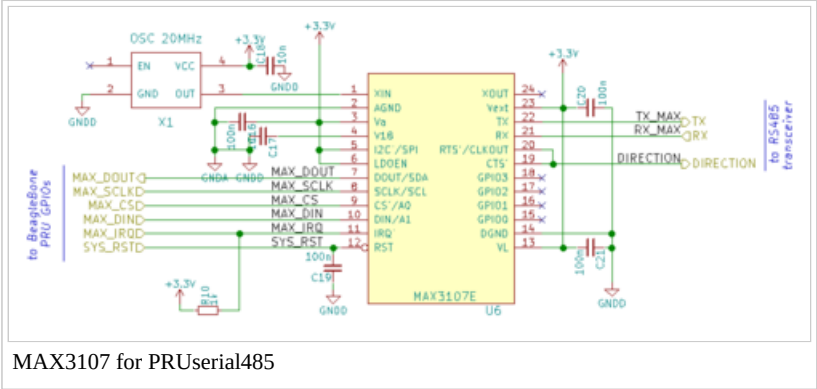
When sync mode is enabled, PRU is completely dedicated to receiving timing signals, polling a GPIO.

In the first model, after receiving a timing pulse, serial interface sends a broadcast message to all devices in the line. Final action depends on how the power supply is configured: it could be a setpoint implementation or start of a magnet cycling, for example. The second one sends a sequence of setpoints each time a timing pulse is received, following ramp curves that user has loaded previously. Ramp curves are stored in DDR memory and they are subdivided into four different groups. A group is composed of four curves of up to 6250 points (4-byte floating points) each.

In this case, it is possible to changes curves on-the-fly by loading them into a different block as well as changing pointer for next point of curve in execution. Also, two more parameters should be configured: execute curve once or multiple times and intercalate normal write/read messages between timing commands or only after the last curve point."

Hardware Requirements

The main Controls Group hardware interface designed for BeagleBone Black, SERIALxxCON, has all the peripherals that are needed to have the PRUserial485 interface working properly.



MAX3107 for PRUserial485

PRUserial485 depends on an external UART chip: MAX3107, from Maxim Integrated. Although it is able to interface with integrated UART subsystem available at BBB SoC, using an external UART has led to FIFO depth increasing and a wide range of programmable baud-rates, reaching up to 15 Mbps. The external UART is digitally controlled with PRU through SPI.

Another important external IC is the line driver. Controls Group uses NVE IL3685 as the RS-485 transceiver.

Programming Developments

This is the core of this project. It is needed to have, at least, two coding files: one for PRU firmware and other dedicated do ARM core (operating system). As a standalone system, PRU runs a binary file which must be pre-loaded and started by the ARM core.

Pin configuration

BeagleBone pins that are connected to P8 and P9 headers can be configured in several modes. Some of them are directly connected to pins that can be driven from PRUs and must be configured before use. For this application and considering SERIALxxCON hardware design and BBB kernel 4.14.x:

```
# PRUserial485 pins
config-pin P8_46 pruin          # SOFT SPI MISO
config-pin P8_39 pruin          # IRQ FROM MAX3107
config-pin P8_27 pruin          # SYNC TIMING INPUT
config-pin P8_45 pruout         # SOFT SPI CLK
config-pin P8_43 pruout         # SOFT SPI CS
config-pin P8_41 pruout         # SOFT SPI MOSI
```

In project folder, there is a bash script that configures pins automatically. It must be run before any application that uses PRUserial485.

Operation Modes

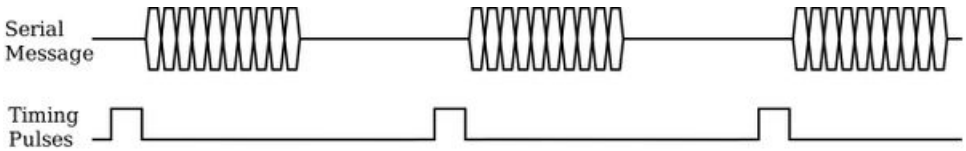
Conventional Mode

Once serial interface is started, it is needed to have it configured. As it is a full-duplex RS-485 standard communication, BeagleBone can have a role either as a **master** or as a **slave** in the network. This parameter, along with desired baudrate, is passed when it is started.

The master is the main device in the network. It is the master who will always request for information and get the reply. The slave will only send data into the serial network if it is requested to do so. Receiving any incoming message, it will only reply if the message is addressed to it.

For both cases, there are *write* and *read* commands to send/get data to/from data line.

Synchronized configuration



PRU Firmware

PRU firmware can be written either in C or assembly language. Texas Instruments provides compilers for both programming languages. In the case it is intended to have high performance and determinism, assembly language will be used in order to write PRU firmware. Few instructions are used, however, a good knowledge of the processor architecture is important.

PRU is the core which will be controlling the external UART (MAX3107).

Some important technical references about AM335x PRUs:

```
AM335x PRU-ICSS Reference Guide
PRU Assembly Language Tools - User's Guide
PRU Assembly Instruction User Guide
```

ARM/Linux Libraries

Texas Instruments also provides a library for dealing with PRUs in system level. It is called "libprussdrv" and can be easily installed through *apt-get install*. Using this library it is possible to load binary files into PRU, get/send signals to it, check/open memory mapping, etc.

Communication between PRU and Operating System is usually performed through memory access (either reading or writing to special registers). For that purpose, it has been developed a library for this project in C and a Python module (that comes from the C library) for interchanging information to/from PRU. Both are installed on the system, being available at the default directory (it is not necessary to carry along .h or .py files when coding).

Source Codes

Sources of PRU software (written in assembly language) and the corresponding system library for Linux (written in C) can be obtained from PRUserial485 Github repository.

Detailed Description

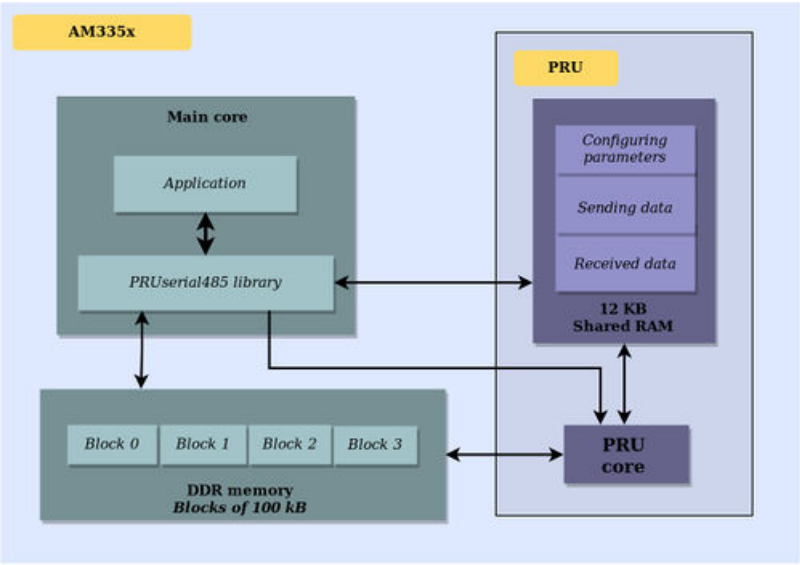
In this section, code development will be detailed. As discussed above, PRU and ARM share data through memory mapping. Specifically, there is a dedicated 12-kbyte shared RAM memory for this purpose and it will be used. Aditionally, it has been extended to DDR memory, in order to increase mapping size (developed for Power Supplies' synchronous operation through BBB).

AM335x Memory Mapping

Memory mapping is a very important feature for this application. It is through these memory spaces that data will be shared between PRU system and main Linux core.

First, and more directly, comes the shared RAM. A PRU-dedicated, it is reserved to application that use real-time cores. This will be used to exchange configuration, control and serial data bytes.

Also, there is the possibility of reserving a larger region of the external DDR memory and use it with PRU sub-systems. It comes to be useful when it is intended to store many long curve points and share them with PRU to perform synchronized adjustments. It will be discussed in the next sub-sections.



Shared RAM Memory

It is a 12kbyte RAM, embedded on the SoC, shared with main core and both PRUs. For PRUserial485 application, memory mapping follows the sequence below:

Shared RAM mapping for PRUserial485

Offset	Description
00	MAX3107 version (0x1A)
01	Data status (0x00: Data ready for reading / 0x55: Old data on buffer / 0xFF: Data ready for sending)
02	Serial baudrate (MAX3107 BRGCONFIG register)
03	Serial baudrate (MAX3107 DIVLSB register)
04	Serial baudrate (MAX3107 DIVMSB register)
05	Sync operation control (0x00: Stop / 0xFF: Start)
06..09	Timeout to wait for a reply after a writing cycle (x10 ns)
10..13	Curve execution: pointer for next curve point
15..18	Curve execution: absolute address of memory block at DDR memory (curve points storage)
20..23	Curve execution: total size, in bytes, of all four curves
24	Board hardware address (SERIALxxCON = 21)
25	Master or Slave operation ("M" or "S")
26..28	One serial byte length (ns)
29..31	Delay to wait between a sync and normal command (x10 ns)
32	Serial receiving timeout (MAX3107 RXTIMEOUT register)
50	Sync operation: command size
51..	Sync operation: command
80..83	Sync operation: receiving pulse counting
84	Sync operation: sync ok (0x00: not waiting for pulse / 0xFF: waiting for pulse triggering)
85	Sync operation: mode (0x51 Single curve sequence & Intercalated read messages / 0x5E Single curve sequence & Read messages at End of curve / 0xC1 Continuous curve sequence & Intercalated read messages / 0xCE Continuous curve sequence & Read messages at End of curve / 0x5B Single Sequence - Single Broadcast Function command)
100..103	Sending data: vector size
104..5999	Sending data: data
6000..6003	Incoming data: vector size
6004..11999	Incoming data: data

DDR Memory

While enabling PRU kernel module, an argument is needed to reserve a piece of DDR external memory (in this case, 0x200000 bytes will be reserved)

```
$ modprobe uio_pruss extram_pool_sz=0x200000
```

Once this is performed, a system file will be created, indicating starting address and size for reserved region. They are located at

```
/sys/class/uio/uio0/maps/map1/addr
/sys/class/uio/uio0/maps/map1/size
```

As a project definition, it has been set to have:

```
1. Four independent blocks (named 0 to 3)
2. Each block will store four curves, one for each power supply (total: 16 curves)
3. Each curve can have up to 6250 points (floating values, ie, 25000 bytes)
```

Here, dealing with up to four curves is justified that Power Supplies can be allocated in crate with up to four of them in the same crate (FBP model) and there is a BSMP command that adjusts all their currents at once.

Inside DDR memory, data is allocated according to the table below.

DDR mapping for PRUserial485

Curve Block	Offset (decimal)	Curve point	Curve Block	Offset (decimal)	Curve point
0	00000 .. 00003	curve0_0[0]	1	10000 .. 10003	curve0_1[0]
	00004 .. 00007	curve1_0[0]		10004 .. 10007	curve1_1[0]
	00008 .. 00011	curve2_0[0]		10008 .. 10011	curve2_1[0]
	00012 .. 00015	curve3_0[0]		10012 .. 10015	curve3_1[0]
	...	...		...	...
	09984 .. 09987	curve0_0[6249]		19984 .. 19987	curve0_1[6249]
	09988 .. 09991	curve1_0[6249]		19988 .. 19991	curve1_1[6249]
	09992 .. 09995	curve2_0[6249]		19992 .. 19995	curve2_1[6249]
	09996 .. 09999	curve3_0[6249]		19996 .. 19999	curve3_1[6249]
2	20000 .. 20003	curve0_2[0]	3	30000 .. 30003	curve0_3[0]
	20004 .. 20007	curve1_2[0]		30004 .. 30007	curve1_3[0]
	20008 .. 20011	curve2_2[0]		30008 .. 30011	curve2_3[0]
	20012 .. 20015	curve3_2[0]		30012 .. 30015	curve3_3[0]
	...	...		...	...
	29984 .. 29987	curve0_2[6249]		39984 .. 39987	curve0_3[6249]
	29988 .. 29991	curve1_2[6249]		39988 .. 39991	curve1_3[6249]
	29992 .. 29995	curve2_2[6249]		39992 .. 39995	curve2_3[6249]
	29996 .. 29999	curve3_2[6249]		39996 .. 39999	curve3_3[6249]

PRU Firmware

PRU is an assembly-based firmware for this application, once performance must be considered as well as jitter and determinism. This section discuss the code itself and strategies of implementation.

Files and compiling

Source files for PRUserial485:

```
PRUserial485.p
PRUserial485.hp
```

Texas Instruments provides a compiler (pasm) for PRU devices. As a standard, Controls Group stores its binary files into /usr/bin folder. For compiling code and generating a binary PRUserial485.bin:

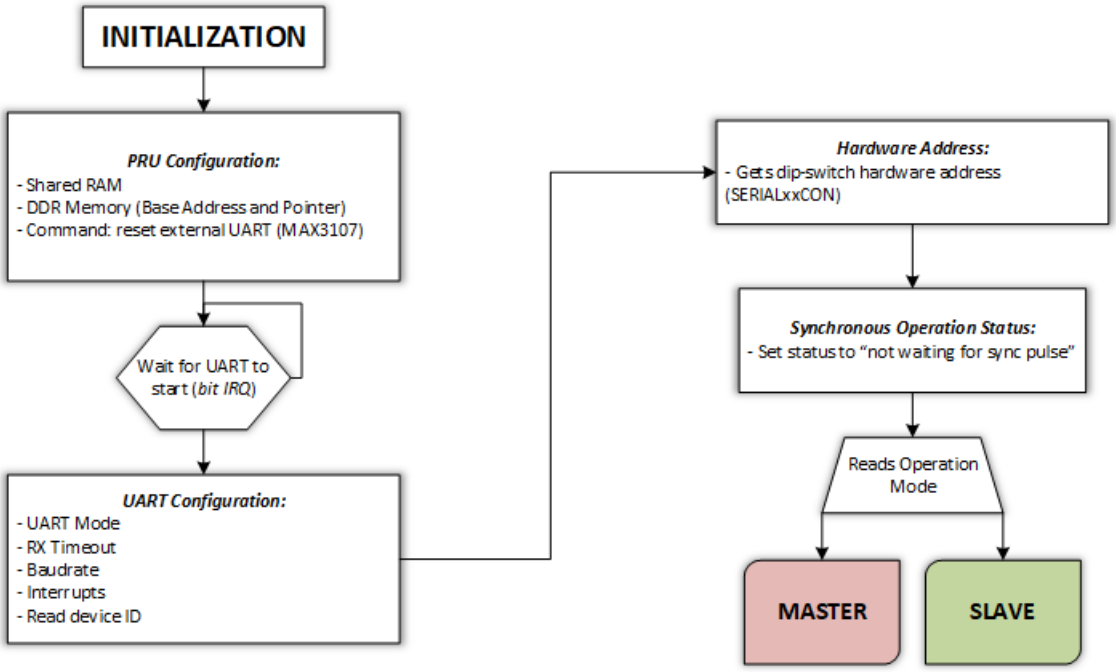
```
pasm -V3 -b PRUserial485.p
mv PRUserial485.bin /usr/bin
```

-V for PRU core version number. V3 for PRUSSv2 (latest)

-b for little endian binary file

Code Structure

Initialization



ARM/Linux Libraries

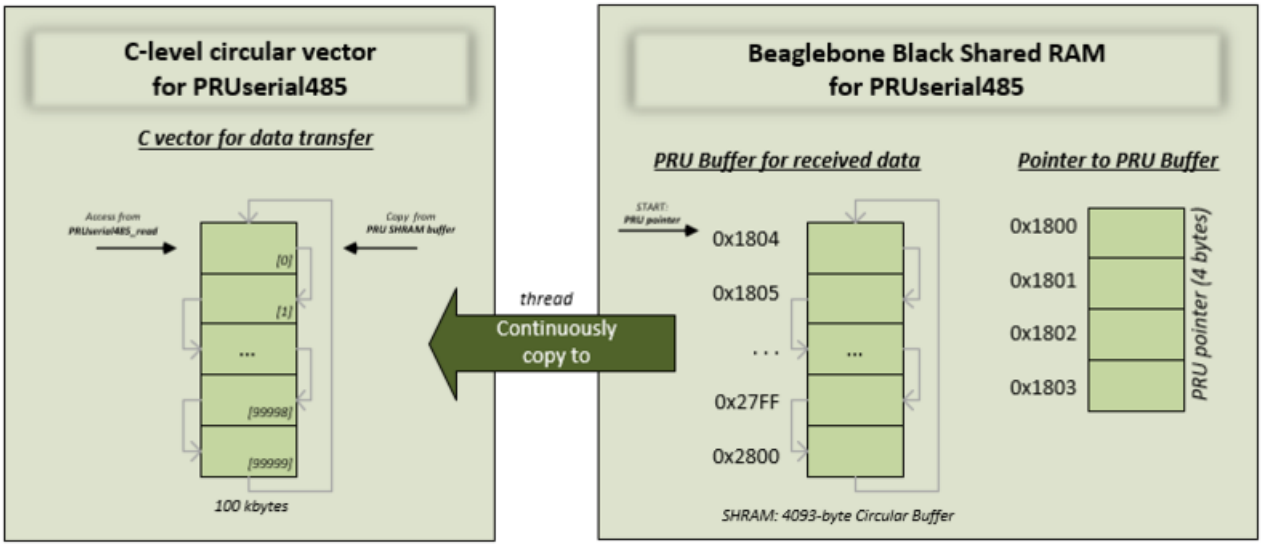
ARM/Linux library is coded in C language and a Python module is constructed using *Python.h* C library. This library is a bridge between userspace and PRU functionalities. It is used to pre-configure serial interface, to share data, select operation mode and verify PRU status.

Initialization

It is **mandatory** to initialize the interface before using it. For that purpose, there is a function available. PRUs and its interrupts are initialized, shared memory is mapped and some configuration (baudrate, master/slave mode, counting registers, etc) are set on memory before executing PRU binary.

As explained on PRU section, there is a circular buffer for incoming data. At ARM level, the circular buffer depth is pre-configured to 100 kB (it can be increased, but it is enough for regular applications, so far).

When invoking init function, an independent thread is launched (which is killed when closing the interface). The task performed by this thread is copying data from shared memory to a larger buffer on ARM environment as soon as it is available. This buffer is accessed by this thread an also by the function which gets PRU data (PRUserial485_read). A semaphore is implemented in this case.



The functions available can be divided into three main cathegories, which will be described further:

- **General purpose:** includes basic write and read functions and also special functions to open and close the serial port
- **Curves:** dedicated for loading and manipulating curves when operating in sync mode.
- **Sync operation:** commands designed to operate the interface when sync operation is required.

Compiling and installing the library

For compiling the library into your BeagleBone Black, clone the repository into it. There is a bash script that user can run to have all files correctly compiled and installed on the system (along with PRU firmware). Steps to install it are listed below:

```
# Compiling and installing C library
gcc -mfloat-abi=hard -Wall -fPIC -O2 -mtune=cortex-a8 -march=armv7-a -I/usr/include -c -o PRUserial485.o PRUserial485.c
ar -rv libPRUserial485.a PRUserial485.o
gcc -shared -Wl,-soname, -o libPRUserial485.so PRUserial485.o

install -m0755 libPRUserial485.a libPRUserial485.so /usr/lib
ldconfig -n /usr/lib/libPRUserial485.*
install -m0755 PRUserial485.h /usr/include

rm PRUserial485.o libPRUserial485.so libPRUserial485.a

# Installing Python module
python-sirius setup.py install
```

Library Functions and Overview

General Purpose Commands

Open serial interface

```
Python: int PRUserial485_open(int baudrate, byte mode)
C:      int init_start_PRU(int baudrate, char mode)

PARAMETERS
baudrate: RS485 serial desired baudrate. Available: 9600, 14400, 19200, 38400, 57600, 115200 bps and 6, 10, 12 Mbps
mode:     "M" for master and "S" for slave mode.
```

PRU initialization. Shared memory configuration and loading binaries into PRU. It is mandatory to open serial interface before invoking any other function.

Hardware physical address

```
Python: int PRUserial485_address()
C:      uint8_t hardware_address_serialPRU()

RETURN
Integer value (0 to 31)
```

Gets SERIALxxCON board address (hardware defined). For SERIALxxCON board, default value is 21.

Close serial interface

```
Python: int PRUserial485_close()
C:      void close_PRU()
```

Closes PRUs and memory mapping.

Send data through RS-485

```
Python: int PRUserial485_write(data, float timeout)
C:      int send_data_PRU(uint8_t *data, uint32_t *size, float timeout_ms)

PARAMETERS
data: Python bytes containing values to be transmitted through serial network / C array with bytes ti be transmitted.
timeout: Maximum waiting time to start getting an answer, in milliseconds (ms). Minimum: 15ns / Maximum: 64s. If 0, does not wait for res
tamanho: Number of bytes to be transmitted. Not needed in python module.

RETURN
Master mode: Returns only after response received (valid response, timeout or ignored)
Slave mode: Returns just after data completely sent.
```

Send data through RS485 network.

Receive data from serial

```
Python: bytes PRUserial485_read(uint32_t nbytes = 0)
C:      int recv_data_PRU(uint8_t *data, uint32_t *tamanho, uint32_t bytes2read)
```

```
PARAMETERS
nbytes/bytes2read: number of bytes to be read from the input buffer. Default value is 0 (entire buffer).
data: C array where incoming bytes will be saved to. Not needed in python module.
tamanho: Number of incoming bytes. Not needed in python module.

RETURN
Python: bytes corresponding to data received, according to nbytes.
C: 0
```

Receiving data through RS485 network

Flush input buffer

```
Python: int PRUserial485_read_flush()
C: int recv_flush()
```

Flush receive FIFO buffer.

Curves

Loading a curve

```
Python: int PRUserial485_curve(int block, [float_list curve1, float_list curve2, float_list curve3, float_list curve4])
C:      int loadCurve(float *curve1, float *curve2, float *curve3, float *curve4, uint32_t CurvePoints, uint8_t block)

PARAMETERS
curveX: Python float list/C vector containing curve points, up to 6250 points. Curves must all have same length.
block: Identification of block which will be loaded with curve points. (0 to 3)

RETURN
0 if curves are successful loaded.
```

Storing curves into memory. Each curve correspond to a power supply in the crate.

Selecting curve block to be performed

```
Python: void PRUserial485_set_curve_block(int block)
C:      void set_curve_block(uint8_t block)

PARAMETERS
block: Identification of block (0 to 3)
```

Selection of block which will be performed in next cycle. Default value is 0.

Get curve block that will be performed

```
Python: int PRUserial485_read_curve_block()
C:      uint8_t read_curve_block()

RETURN
Block identification (0 to 3)
```

Read block identification which will be performed in next cycle.

Select a point to be performed

```
Python: void PRUserial485_set_curve_pointer(int next_point)
C:      void set_curve_pointer(uint32_t new_pointer)

PARAMETER
next_point/new_pointer: index of next point (0 to (len(curve)-1))
```

Selection of point of curve that will be performed after the next sync pulse.

Read which point will be performed

```
Python: int PRUserial485_read_curve_pointer()
C:      int uint32_t read_curve_pointer()
```

```
RETURN
Index of next point (0 to (len(curve)-1))
```

Read curve index (point) which will be sent in next sync pulse.

Sync Operation

PRUserial485_sync_start(int sync_mode, float delay, int sync_address)

```
Python: void PRUserial485_sync_start(int sync_mode, float delay, int sync_address)
C:      void set_sync_start_PRU(uint8_t sync_mode, uint32_t delay_us, uint8_t sync_address)

PARAMETERS
sync_mode:  operation category of synchronous mode:
            - 0x51 Single curve sequence & Intercalated read messages
            - 0x5E Single curve sequence & Read messages at End of curve
            - 0xC1 Continuous curve sequence & Intercalated read messages
            - 0xCE Continuous curve sequence & Read messages at End of curve
            - 0x5B Single Sequence - Single Broadcast Function command
delay/delay_us: time between end of sync serial message and start of a normal message, when sending normal commands after sync pulses.
sync_address: PS Controller address to which setpoints will be addressed to. Parameter only needed if sync_mode != 0x5B
```

Synchronous mode operation.

Stop sync mode and return to normal operation

```
Python: void PRUserial485_sync_stop()
C:      void set_sync_stop_PRU()
```

Stops sync operation mode.

Verify sync status

```
Python: bool PRUserial485_sync_status()
C:      int sync_status()

RETURN
True/1: waiting for sync pulse
False/0: not waiting for sync pulse
```

Verifies whether PRU is waiting for a sync pulse or not

Reading pulse count register

```
Python: int PRUserial485_read_pulse_count_sync()
C:      uint32_t read_pulse_count_sync()

RETURN
counting value (0 to 4294967295)
```

Read number of sync pulses already received.

Clear pulse counting register

```
Python: int PRUserial485_clear_pulse_count_sync()
C:      int clear_pulse_count_sync()

RETURN
0 if succeeded.
```

Clears pulse counting registers. Action is only performed if sync mode is disabled.

Intrinsic timing constraints

Software delays are real and in the case of high performance applications, they must be considered. For library version 1.3.3, some of them were quantified, as shown below:

Sync Mode - PRU level

- Lattency of sync message regarding sync pulse

    Broadcast command: ***1.02 µs***
    Curve setpoint command: ***2.12 µs***

- Sync message jitter: **13.45 ns**
- Returning to waiting for next pulse, after sending a

    Sync message: ***3.1 µs***
    Normal message, without waiting for its answer: ***4 µs***
    Normal message, receive the answer immediately (no response delay) and store its 64 bytes: ***48 µs***

Sync Mode - Processor level

- Load curve into memory: **29 ms**
- Start sync mode: **14 µs**
- Close sync mode and recover normal operation: **517 µs**

Normal Mode - Processor level

- Initialize PRUserial485: **1.4 ms**
- Complete message sending, without waiting for an answer

    5-byte message: ***50 µs***
    50-byte message: ***165 µs***
    100-byte message: ***250 µs***

- Complete message sending, receiving the answer immediately (no response delay) and store its 64 bytes:

    5-byte message: ***120 µs***
    50-byte message: ***215 µs***
    100-byte message: ***305 µs***