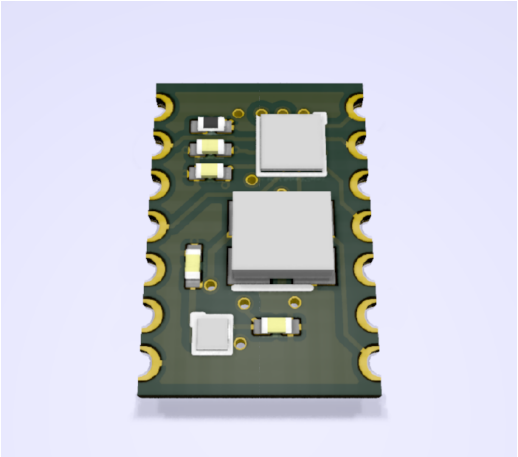
CD485 Data Manual

Duke Fong

June 10, 2017



Contents

1 Functional description

1.1 Overview

CD485 (or CD-BUS) is a communication protocol based on RS485, but it also represents the hardware implementations.

CD485 protocol was designed by Duke Fong in 2009 for simple, multi-master and high-speed communication in mind.

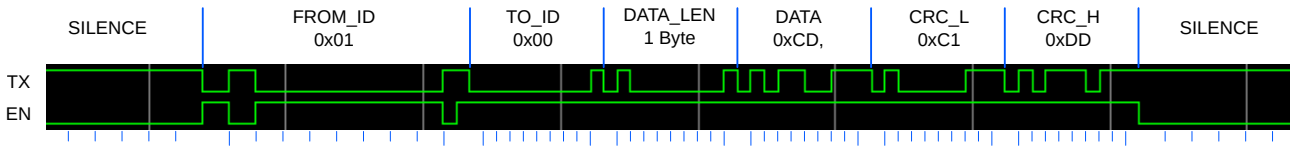
1.2 Highlights

Currently hardware implements:

- Support multiple master on CD485 bus, arbitration by node ID address
- Pack user data up to 253 bytes per packet for communication
- 8 buffer pages for RX purpose, 2 buffer pages for TX purpose, each page is 256 bytes
- 16 bit hardware CRC generation and verification
- Baud rate from 412 bps to 9 Mbps (support 10 Mbps by replace the oscillator; and there is another device which support up to 36Mbps)
- Separate baud rate setting for arbitration byte and follow data
- Backward-compatible with traditional RS485 bus
- Support SPI and I2C peripheral interface
- Easy configuration and operation

2 CD485 protocol

Timing example of CD485 bus:



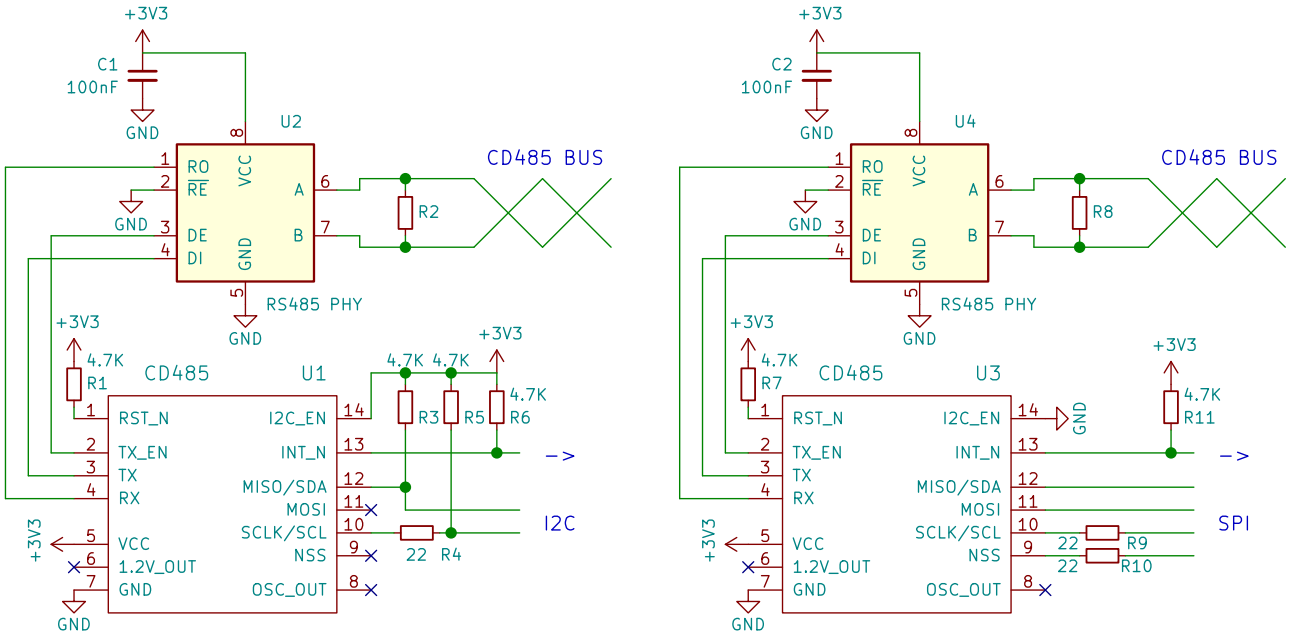
Field name	Length (bytes)	Purpose
SILENCE	0~25.5 Default: 2 (20 bits)	The separator between packets Wait for the end of any packet on the bus and bus keep logic 1 for SILENCE bits of time, bus enter IDLE mode.

FROM_ID	1	Sender ID Only allow sending after bus kept in IDLE mode for a period of length (10 bits by default). TX_EN pin inactive for all logic 1 during this field, allow the sender read back bus state to check if there are any other node start sending at same time, if so, the lower priority node immediately stops sending and deferred sending, or enable TX_EN at the end of last check. Priority level caculate by formula: $255 - bit\_reversal(\text{FROM\_ID})$ We read back bus state at middle of logic 1 bits during this field, the baud rate for this field should normally less than 1 Mbps, because of the delay exist between TX and RX, we may read the previous bit we sent if the baud rate set too high.
TO_ID	1	Receiver ID, 255 for broadcast.
DATA_LEN	1	User data length, range: 0~253 bytes, each buffer page is 256 bytes, the first 3 bytes are used by FROM_ID, TO_ID and DATA_LEN.
DATA	0~253	User data
CRC_L	1	Low 8 bits of CRC, Use the same CRC standard as Modbus.
CRC_H	1	High 8 bits of CRC

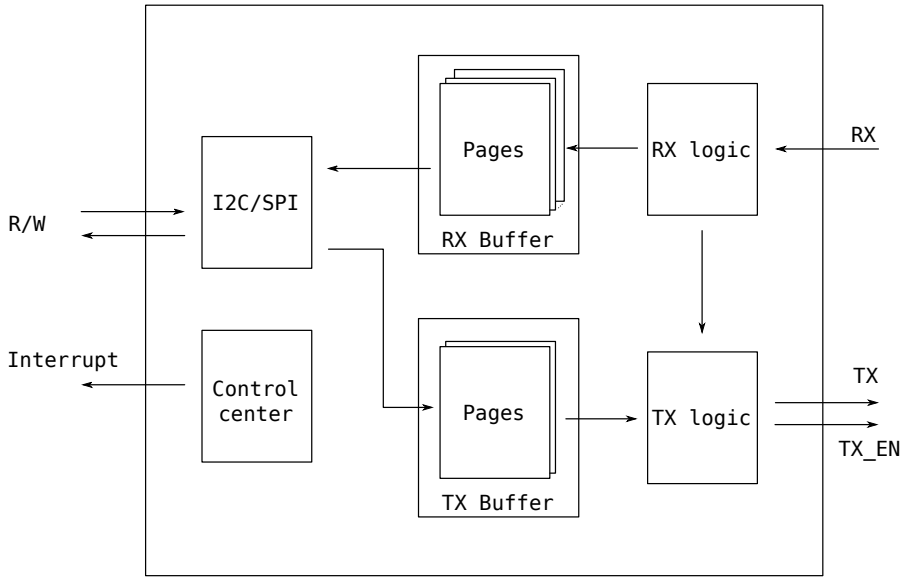
CD485 protocol only defines the packet format, does not specify the user data format; Only supports unicast and broadcast, does not support multicast; Only provide hardware arbitration, automatic retransmission after conflict, packet handshake and error handling are handled by user at upper layer.

3 Hardware

3.1 Circuit reference



3.2 Internal block



3.3 Pin definition

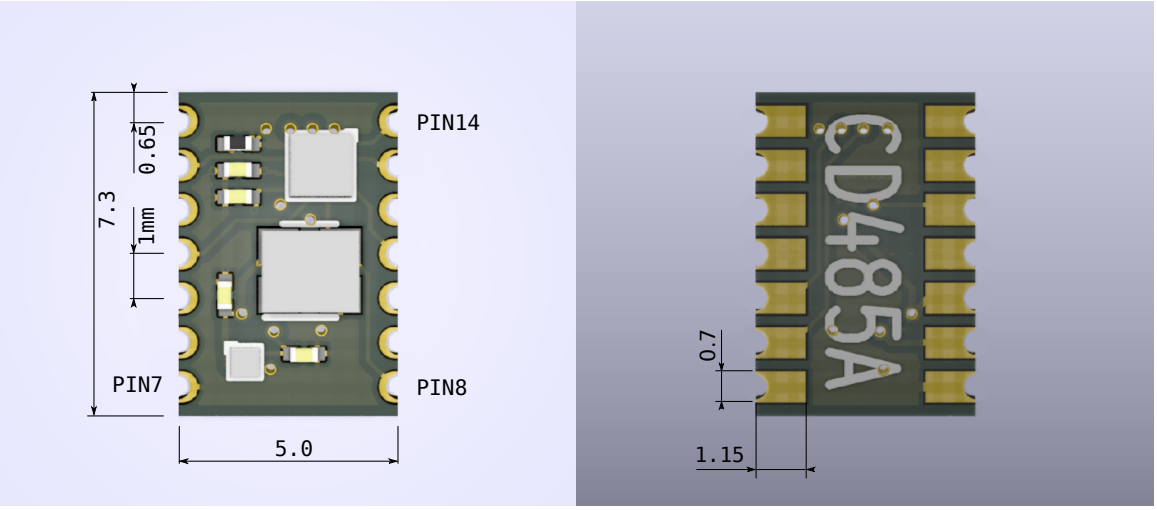
No	Name	I/O	Pull	Description
1	RST_N	I	-	RST_N, should pull up by external resister
2	TX_EN	O	D	Transmit enable pin to RS485 PHY (internal 2 KΩ pull-down resister)
3	TX	O	U	Transmit pin to RS485 PHY
4	RX	I	-	Receive pin from RS485 PHY
5	VCC			Supply voltage: 3.3V ±5%, ≤5mA
6	1.2V_OUT			On module 1.2V LDO ouput
7	GND			Ground
8	OSC_OUT	O		On module 27MHz oscillator output
9	NSS	I	U	SPI chip select (do not drive low during power on or reset)
10	SCLK/SCL	I	U	SPI/I2C clock
11	MOSI	I	-	SPI MOSI
12	MISO/SDA	I/IO	U	SPI MOSI / I2C SDA
13	INT_N	O	-	Interrupt pin, open-drain output
14	I2C_SEL	I	-	High or open select I2C mode, low select SPI mode

All pull-up is around 50 KΩ, and only pulled after reset.

Notice: The TX pin ouput low, NSS ouput high and SCLK/SCL output clock pulse for a short period during power on and reset, do not drive those pin during this period, add two small resistor for NSS and SCLK/SCL

between the module and CPU is suggested.

3.4 Mechanical specifications



4 Register reference

Addr	Name	R/W	description
0x00	VERSION	R	Hardware version, value: 0xC1.
0x01	SETTING	RW	Bits:

bit0	OUTPUT_EN If not set, TX and TX_EN ouput Hi-Z (high impedance).
bit1	NO_ARBITRATE Disable auto arbitration and always active TX_EN pin during sending.
bit2	USER_CRC Disable hardware CRC generation and verification. If set, user should write two addition CRC bytes followed by data, and read two more bytes for CRC after data, the data length reduced to 251 bytes in this situation.
bit3	NO_DROP If set, not drop packet which set RX_ERROR flag. Determine whether the packet is correct or not in current RX page by RX_PAGE_FLAG.
bit[5:4]	TX_EN_ADVANCE (only used if NO_ARBITRATE is set). Advance TX_EN for bits of time before TX sending (with 1 system clock period in addition).

Default value: xx01x000 (x: not care, write by 0).

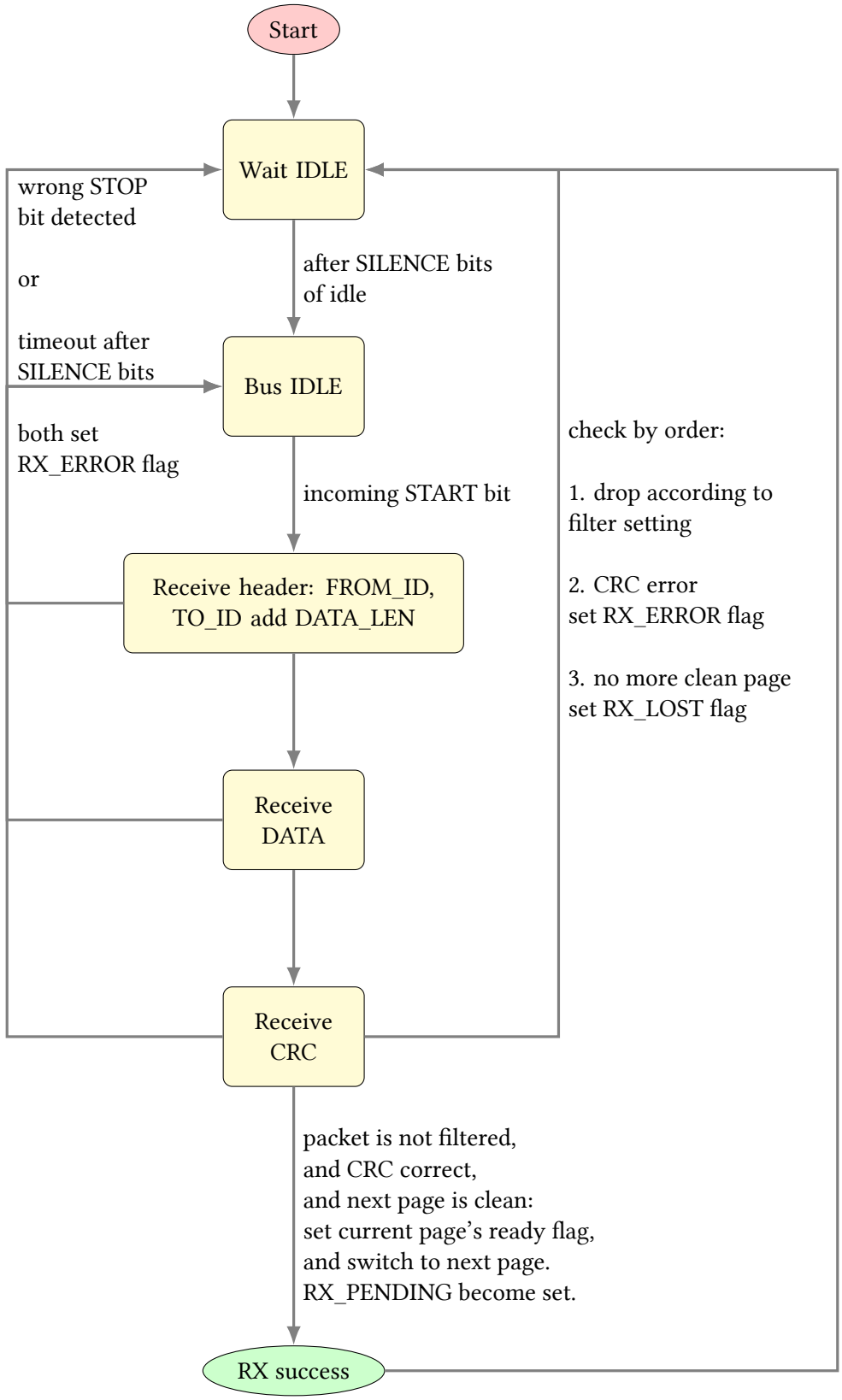
<table>
<tr><td>0x02</td><td>SILENCE_LEN</td><td>RW</td><td>Bus enter IDLE when bus keep logic 1 for SILENCE bits of time after end of any packet, default: 20 (bits).</td></tr>
<tr><td>0x03</td><td>TX_DELAY</td><td>RW</td><td>Allow TX after bus kept in IDLE mode for this period of length, default: 10 (bits).
You could lower the value for higher priority node, retain at least 1 bit to make sure all node have enough time to detect the bus's IDLE state.</td></tr>
<tr><td>0x04</td><td>SELF_ID</td><td>RW</td><td>Only used for RX packet filtering: (match from top to bottom)
<table><tr><th>FROM_ID</th><th>TO_ID</th><th>SELF_ID</th><th>Receive or drop</th></tr><tr><td>not care</td><td>not care</td><td>255</td><td>receive (sniffer mode)</td></tr><tr><td>= SELF_ID</td><td>not care</td><td>!= 255</td><td>drop (avoid loopback)</td></tr><tr><td>!= SELF_ID</td><td>255</td><td>not care</td><td>receive (broadcast)</td></tr><tr><td>!= SELF_ID</td><td>!= 255</td><td>= TO_ID</td><td>receive (P2P)</td></tr><tr><td>not care</td><td>!= 255</td><td>!= TO_ID</td><td>drop</td></tr></table>Default: 255.</td></tr>
<tr><td>0x05</td><td>PERIOD_LS_L</td><td>RW</td><td>Low byte of PERIOD_LS (LS: short for Low Speed)
Bond rate corresponding divisor factor for SILENCE, TX_DELAY and FROM_ID (for EN_ADVANCE also).
Formula: $factor = sysclock \div bond\_rate - 1$
E.g. the system clock is 27 MHz, set factor to 233 to get bond rate close to 115200 bps (default: 233).</td></tr>
<tr><td>0x06</td><td>PERIOD_LS_H</td><td>RW</td><td>High byte of PERIOD_LS, 16 bits in total (default: 0).</td></tr>
<tr><td>0x07</td><td>PERIOD_HS_L</td><td>RW</td><td>Low byte of PERIOD_HS (HS: short for High Speed, default: 233)
Bond rate corresponding divisor factor for TO_ID, DATA_LEN, DATA and CRC_L/H.</td></tr>
<tr><td>0x08</td><td>PERIOD_HS_H</td><td>RW</td><td>High byte of PERIOD_HS, 16 bits in total (default: 0).</td></tr>
</table>

0x09	INT_FLAG	R	Interrupt flag: bit0 BUS_IDLE Indicate whether the bus is in IDLE. bit1 RX_PENDING Indicate whether any page of the RX buffer is ready for read Clear current page's ready flag by write 1 to RX_CTRL [CLR_RX_PENDING] bit. bit2 RX_LOST Auto set when incoming packet is correct, but droped due to no more page is clean for next RX. Clear by write 1 to RX_CTRL[CLR_RX_LOST] bit. bit3 RX_ERROR Auto set when incoming packet is incorrect: stop bit error, timeout or CRC error. Clear by write 1 to RX_CTRL[CLR_RX_ERROR] bit. bit4 TX_BUF_CLEAN Indicate whether each page of the TX buffer is clean. bit5 TX_CD Auto set when collision detected during TX, Clear by write 1 to TX_CTRL[CLR_TX_CD] bit. This bit is for debug purpose. bit6 TX_ERROR After collision detected, the hardware re-send the packet when bus idle for specified time, if collision continue for further 3 times, the transmition will be canceled, and this flag will be set. Clear by write 1 to TX_CTRL[CLR_TX_ERROR] bit.
0x0A	INT_MASK	RW	Interrupt mask INT_N output low if INT_FLAG & INT_MASK != 0, ouput Hi-Z otherwise (default: 0x00).
0x0B	RX	R	Read data from internal RX buffer page, address pointer auto increase There are 8 buffer pages for RX purpose with each 256 bytes. When hardware side receive a wanted packet: if next page is clean for next receive, set current page's ready flag and switch to the next page; else drop the packet and set RX_LOST bit. On user side, the RX_PENDING bit indicated that there are some pages ready for read, write 1 to CLR_RX_PENDING bit clear current page's ready flag and switch to next page. write 1 to RST_RX bit clear all page's ready flag by reset both RX buffer and RX logic. RX_PENDING bit is clean if all page's ready flag are clean.

<table>
<tr><td>0x0C</td><td>TX</td><td>W</td><td>Write data to internal TX buffer page, address pointer auto increase
There are 2 buffer pages for TX purpose with each 256 bytes.
On user side, when finish writing packet to current page, user should wait for TX_BUF_CLEAN bit be set, then trigger START_TX bit to set current page's ready flag and move to next page (nothing happen if TX_BUF_CLEAN bit is clean).
The hardware side gonna start TX if page's ready flag was set, clear the flag and waiting for next page when complete reading.
TX_BUF_CLEAN bit is set if all page's ready flag are clean.</td></tr>
<tr><td>0x0D</td><td>RX_CTRL</td><td>W</td><td>RX control:<table><tr><td>bit0</td><td>RST_RX_POINTER
Write 1 to reset the read pointer of current RX page.</td></tr><tr><td>bit1</td><td>CLR_RX_PENDING (auto select bit0)</td></tr><tr><td>bit2</td><td>CLR_RX_LOST</td></tr><tr><td>bit3</td><td>CLR_RX_ERROR</td></tr><tr><td>bit4</td><td>RST_RX (auto select bit0, 2, 3)</td></tr></table></td></tr>
<tr><td>0x0E</td><td>TX_CTRL</td><td>W</td><td>TX control:<table><tr><td>bit0</td><td>RST_TX_POINTER
Write 1 to reset the write pointer of current TX page.</td></tr><tr><td>bit1</td><td>START_TX (auto select bit0)</td></tr><tr><td>bit2</td><td>SET_TX_BUF_CLEAN_MASK
Set INT_MASK[TX_BUF_CLEAN] bit.
You can set this bit while set START_TX, then the TX_BUF_CLEAN interrupt will occur after the transmission has been completed.</td></tr><tr><td>bit3</td><td>CLR_TX_CD</td></tr><tr><td>bit4</td><td>CLR_TX_ERROR</td></tr></table></td></tr>
<tr><td>0x0F</td><td>RX_ADDR</td><td>RW</td><td>Set and get the read pointer of current RX page.</td></tr>
<tr><td>0x10</td><td>RX_PAGE_FLAG</td><td>R</td><td>(only used if NO_DROP is set).
Value zero indicate the packet in current RX page is correct;
Non-zero indicate the pointer of last received byte of the disturbed packet, include CRC.</td></tr>
</table>

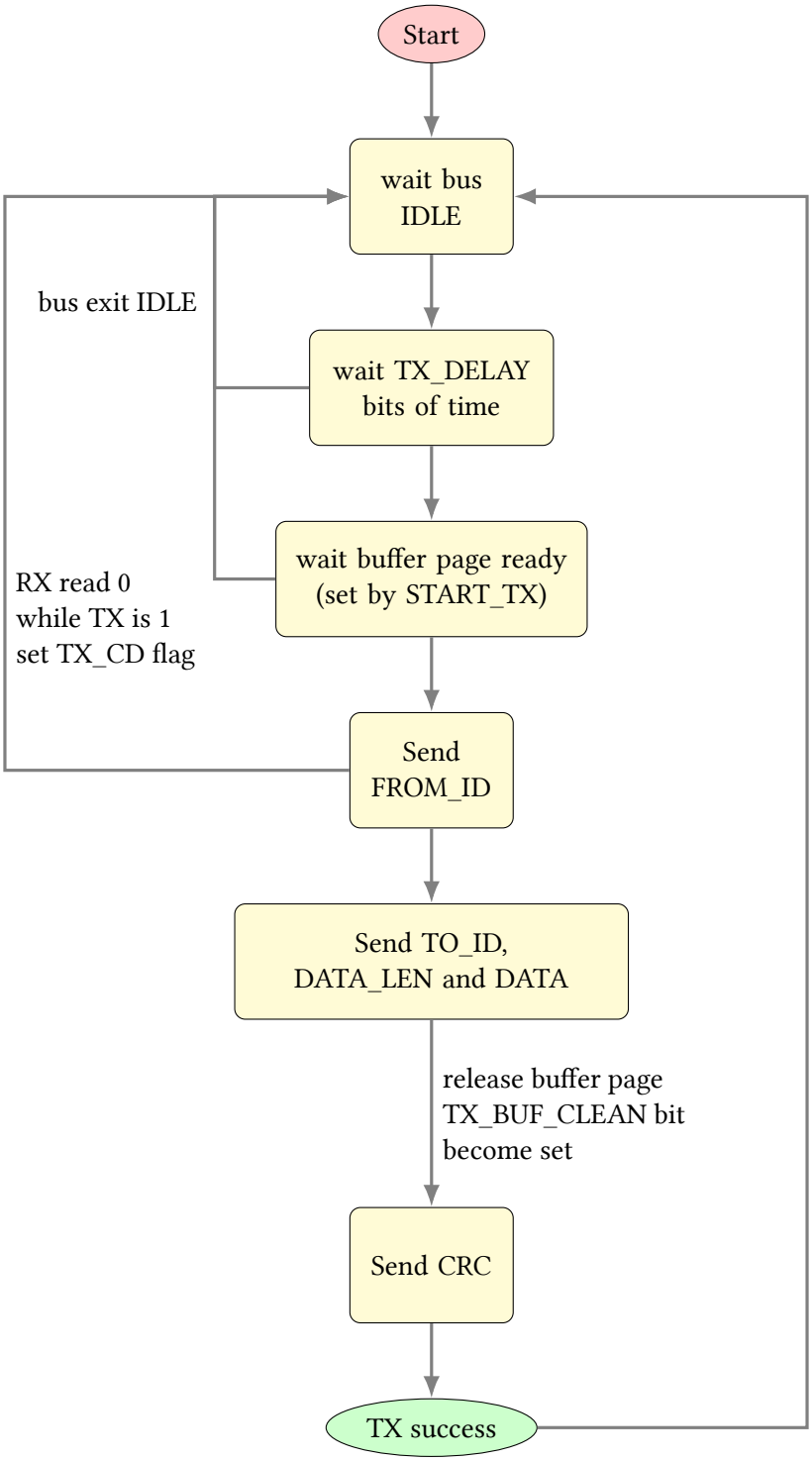
5 Workflow

5.1 RX



Not set RX_ERROR flag if current packet is gonna filtered or less than two bytes received.

5.2 TX



6 Peripheral interface

Both SPI and I2C frequency should better less than $sysclock \div 10$.

We usually read or write only 1 byte except reg RX and TX.

6.1 SPI

Read and write:

```
start (NSS = 0)
Write reg address with bit7:  0: read, 1: write
Read or write arbitrary length of data
stop (NSS = 1)
```

6.2 I2C

Write address: 0xc0
Read address: 0xc1

Write:

```
start
write the write address
write 1 byte reg address
write arbitrary length of data
stop
```

Read:

```
start
write the write address
write 1 byte reg address
restart (or stop + start)
write the read address
read arbitrary length of data, ACK all bytes except last byte
stop
```

7 Operate demonstration

7.1 Init

```
// select system clock, enable OUTPUT
CD485_write(REG_SETTING, F27M | OUTPUT_EN);

// set SELF_ID
CD485_write(REG_SELF_ID, 0xcd);

// set bondrate, PERIOD_XX_H default 0
CD485_write(REG_PERIOD_LS_L, 35); // 750000 bps
CD485_write(REG_PERIOD_HS_L, 2);  // 9 Mbps
```

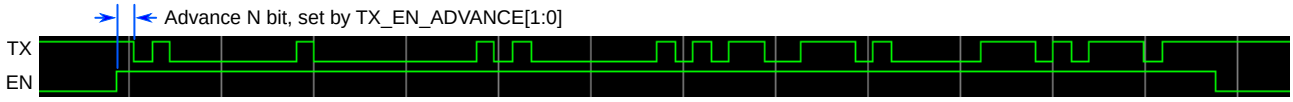
```
// clean RX buffer
CD485_write(REG_RX_CTRL, RST_RX);

// enable interrupt
CD485_write(REG_INT_MASK, RX_ERROR | RX_LOST | RX_PENDING);
```

7.1.1 Compatible and conventional mode

Set same value to reg PERIOD_LS and PERIOD_HS for compatible mode.

Set NO_ARBITRATE bit further for conventional mode:



7.2 TX

```
header_buf[0] = 0xcd; // FROM_ID
header_buf[1] = 0x02; // TO_ID
header_buf[2] = 12;   // DATA_LEN

CD485_write_chunk(REG_TX, header_buf, 3);             // write HEADER
CD485_write_chunk(REG_TX, data_buf, header_buf[2]);   // write DATA

while (CD485_read(REG_INT_FLAG) & TX_BUF_CLEAN == 0); // make sure TX_BUF_CLEAN is set
// sent packet, and enable TX_BUF_CLEAN interrupt
CD485_write(REG_TX_CTRL, SET_TX_BUF_CLEAN_MASK | START_TX);

// write next packet
// send next packet when the TX_BUF_CLEAN interrupt occur

// if no further packet need send, disable TX_BUF_CLEAN interrupt:
CD485_write(REG_INT_MASK, CD485_read(REG_INT_MASK) & ~TX_BUF_CLEAN);
```

7.3 RX

```
// when RX_PENDING interrupt occur:

CD485_read_chunk(REG_RX, header_buf, 3);           // read HEADER
CD485_read_chunk(REG_RX, data_buf, header_buf[2]); // read DATA

CD485_write(REG_RX_CTRL, CLR_RX_PENDING);          // release page
```

8 Copyright statement

CD485 (or CD-BUS) is a fairly open protocol, hardware implementation is relatively simple, in addition to chip manufacturers need to pay a small amount of royalties, the rest of anyone can use this protocol and its variants for free, only need to retain the original copyright information in the product manual.

Contact: duke@dukelec.com