SUB-20
User Manual

Table of Contents

1 Document Revision History

- 2.17

1. Fixed Power Distribution Picture

- 2.16

1. sub_reset
2. Updated Error Codes

- 2.15

1. EEPROM Read/Write Functions

- 2.14

1. SPI_CLK_XXXXX
2. SUP-20-Dxxx picture

- 2.13

1. Added Power Distribution topic

- 2.12

1. High Speed I2C Master
2. CFP MSA Support

- 2.11

1 Edge Detector

- 2.10

1. I2C_DISABLE

- 2.9

1. sub_spi_transfer_ess

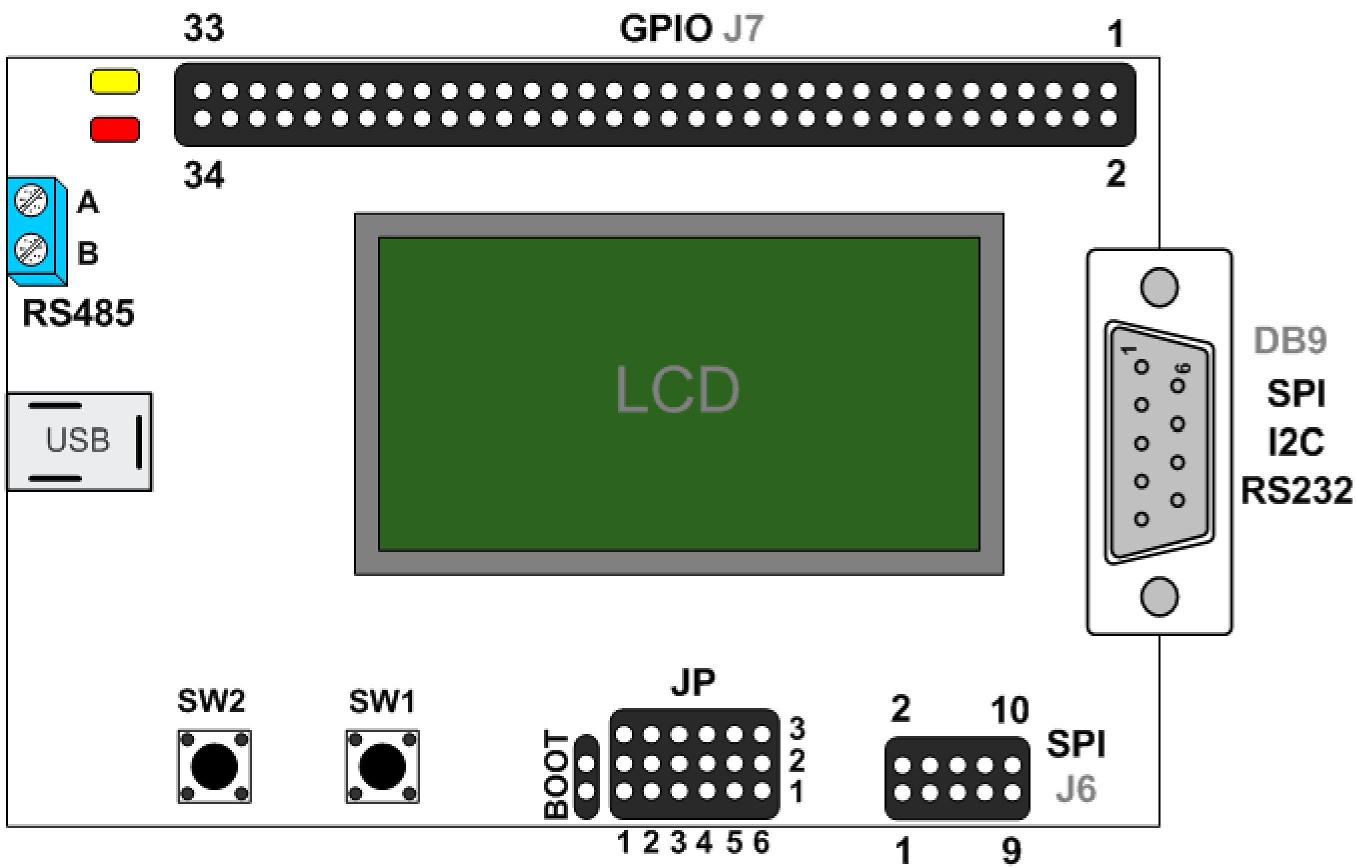
- 2.8

1. GPIOB
2. FIFO_READ_FULL
3. MDIO1

- 2.7

1. Bit-bang I2C Masters
2. Updated GPIO Header for I2Cx, SPIx, UARTx

2 SUB-20 Layout

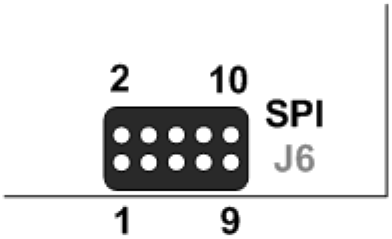


2.1 GPIO Header

Header Pin	GPIO	Alternative Function	Notes
1	GPIO8	I2C SCL	Pulled up 1.5KOhm DB9.8
2	GPIO9	I2C SDA	Pulled up 1.5KOhm DB9.6
3	GPIO10	RS232 RXD_TTL	For Configurations with RS232 or RS485
4	GPIO11	RS232 TXD_TTL	
5	GPIO12		
6	GPIO13		
7	GPIO14	MDC	
8	GPIO15	MDIO	
9	GPIO24	PWM_0, SCLx0, MISOx0[1]	
10	GPIO25	PWM_1, SDAx0, MOSIx0[1]	
11	GPIO26	PWM_2, SCLx1, SCKx0[1]	
12	GPIO27	PWM_3, SDAx1, SSx0[1]	
13	GPIO28	PWM_4, SCLx2, MISOx1[1], UARTx0_RX[1]	
14	GPIO29	PWM_5, SDAx2, MOSIx1[1], UARTx0_TX[1]	
15	GPIO30	PWM_6, SCLx3, SCKx1[1], UARTx1_RX[1]	
16	GPIO31	PWM_7, SDAx3, SSx1[1], UARTx1_TX[1]	
17	GPIO0	LCD_D0	For Configurations with LCD
18	GPIO1	LCD_D1	
19	GPIO2	LCD_D2	
20	GPIO3	LCD_D3	
21	GPIO4	IR_TX	For Configurations with IR
22	GPIO5		
23	GPIO6	LCD_En	For Configurations with LCD
24	GPIO7	LCD_RS	
25	GPIO23	ADC7	
26	GPIO22	ADC6	
27	GPIO21	ADC5	
28	GPIO20	ADC4	
29	GPIO19	ADC3	
30	GPIO18	ADC2	
31	GPIO17	ADC1	
32	GPIO16	ADC0	
33	VCC		
34	GND		

[1] - Under development

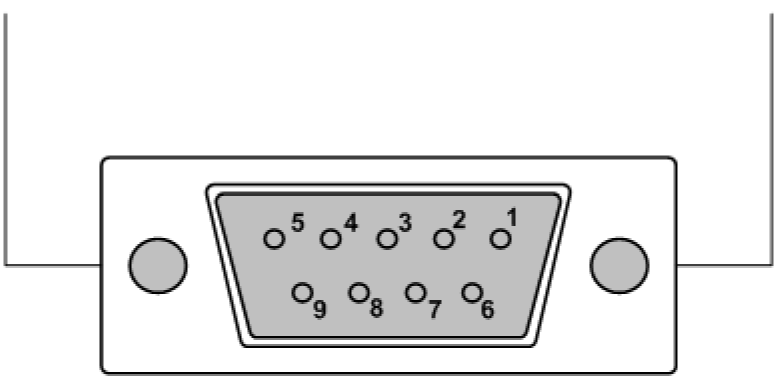
2.2 SPI Header



Header Pin	SPI Function	Fast PWM	GPIOB	MDIO1	Notes
1	MOSI - Master Out Slave In		GPIOB1	MDC1	Optional Level Converter*
2	SS1		GPIOB0		GPIO Voltage**
3	MISO - Master In Slave Out		GPIOB3		Optional Level Converter*
4	SS2	FPWM_0	GPIOB2		GPIO Voltage**
5	SS0		GPIOB5		Output,Optional Level Converter*
6	SS3	FPWM_1	GPIOB4		GPIO Voltage**
7	SCK - Master Clock		GPIOB7	MDIO1	Optional Level Converter*
8	SS4	FPWM_2	GPIOB6		GPIO Voltage**
9	SPI_EXT_PWR - External SPI Power		EXT_PWR		
10	GND		GND		

Level Converter* - Adjustable operating voltage in configurations with Level Converters (see Jumpers and Ordering Information).
GPIO Voltage** - Operates at the same voltage as GPIO according to SUB-20 configuration (see Ordering Information).

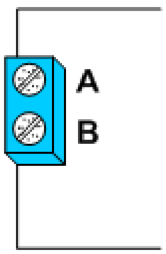
2.3 DB9 Connector



Pin	Function	Configuration
1	SPI SS0	
2	I2C_EXT_PWR	
	RS232 PC_RX	Serial2
3	SPI_EXT_PWR	Lxxx

	RS232 PC_TX	Serial2
4	SPI SCK	
5	GND	
6	I2C SDA	
7	SPI MISO	
8	I2C SCL	
9	SPI MOSI	

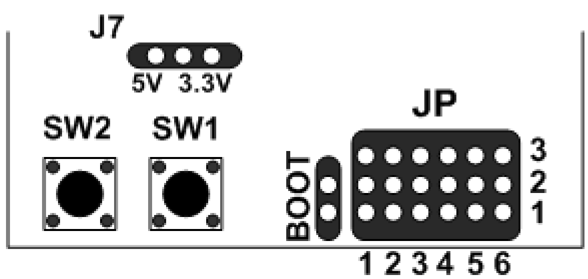
2.4 RS485 Connector



RS485 Connector is installed only in Serial4 configuration.

Pin	Function
A	inverting or '-' pin
B	non-inverting or '+' pin

2.5 Jumpers, SW1,SW2

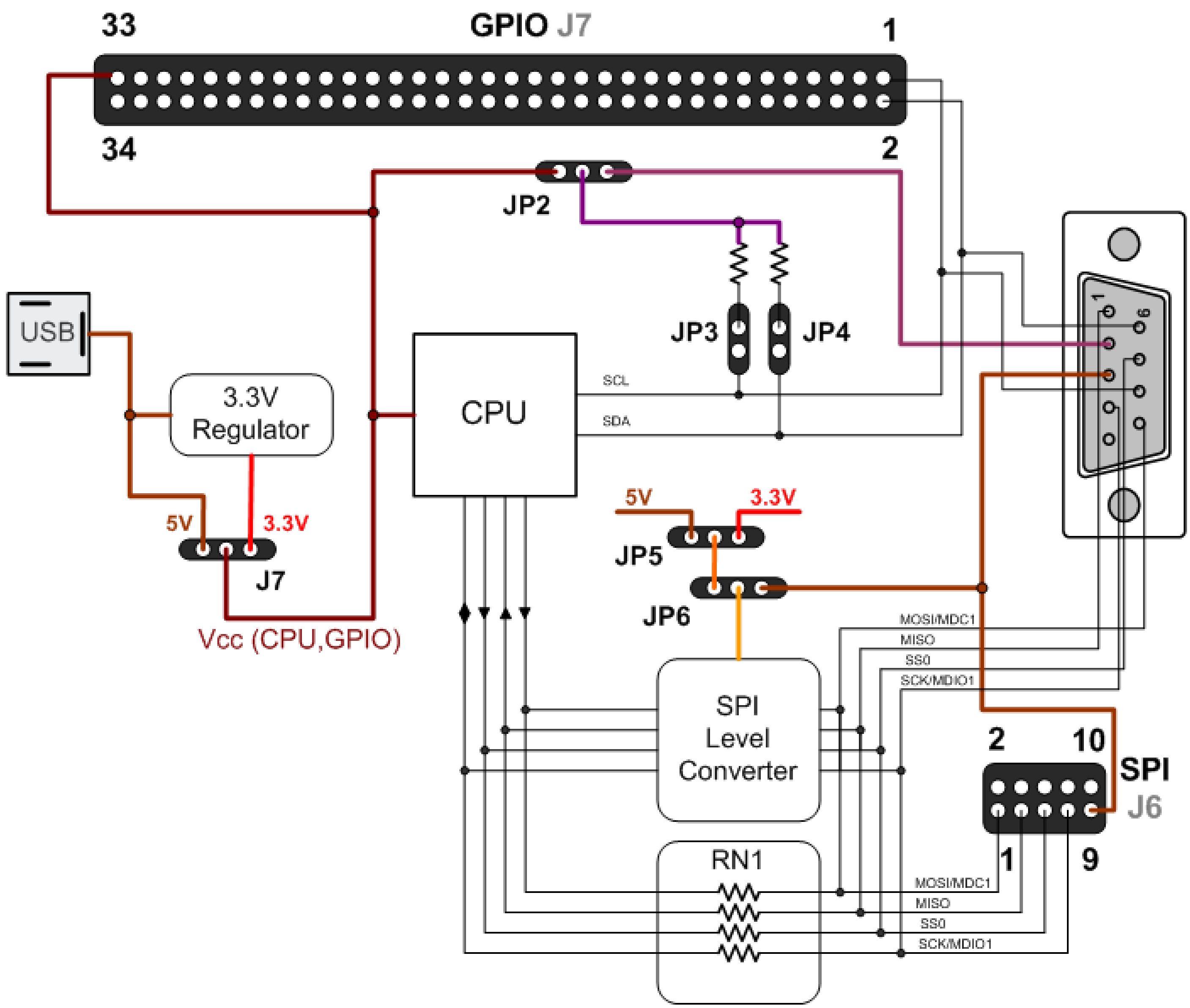


JP1	I2C Pull-ups Voltage when JP2 1-2 closed	1-2	3.3V	
		2-3	5V	Should not be used with 3.3V configurations
JP2	I2C Pull-ups Voltage	1-2	Internal	
		2-3	External DB9.2	Can not be used in Serial configurations
JP3	SCL Pull-up	1-2	On	
		2-3	Off	
JP4	SDA Pull-up	1-2	On	
		2-3	Off	
JP5	SPI Voltage when JP6 1-2 closed	1-2	3.3V	
		2-3	5V	
JP6	SPI Voltage	1-2	Internal	
		2-3	External J6.9 or DB9.3	Only for Lxxx configurations. DB9.3 Can not be used as SPI Voltage in Serial configurations
J7	GPIO Voltage	left	5V	
		rigth	3.3V	
BOOT, SW1	Force Bootloader on Power Reset	Closed or Pressed		
SW2	Force Monitor on Power Reset	Pressed		Only for Serial2 configurations

For additional information about SUB-20 configurations please see Ordering Information.

2.6 Power Distribution

The diagram below shows SUB-20 power distribution and voltage levels.



CPU and GPIO Voltage

CPU and GPIO voltage is defined by J7. It can be 5V from the USB or 3.3V from the internal regulator. In customized SUB-20 models 3.3V regulator can be exchanged with 2.7V .. 5V regulator.
It is possible to source up to 70mA from GPIO33 to power external target.

I2C Voltage

I2C voltage is defined by I2C pull-ups voltage. If pull-ups are enabled with JP3, JP4 I2C voltage is controlled by JP2. It can be internal 5V or 3.3V or external taken from DB9.2. External I2C voltage is not available in SUB-20-R25 (Serial2) model.

SPI/MDIO1 Voltage

SPI Level Converter and RN1 resistors network are mutually exclusive. SPI Level Converter is available in SUB-20-Lxxx models.

- If SPI Level Converter installed SPI/MDIO1 voltage is defined by JP5 and JP6. It can be 5V or 3.3V if JP6 is configured for internal voltage or externally referenced if JP6 is configured for external voltage. External voltage can be taken from SPI.9 or DB9.3 (in non Serial2 model).External voltage reference can be in range 1.6V .. 5V for SUB-20-Lxxx models or 1.2V .. 3.3V for SUB-20-Lxxx-1.2 models. Models with SPI Level Converter can not be used in SPI Slave mode.

- If SPI Level converter is not installed RN1 is used to rout SPI/MDIO1 signals to SPI header and DB9 connector. SPI/MDIO1 voltage in this case is defined by CPU Voltage.

3 SUB-20 API

API Functions List

sub_find_devices
sub_open
sub_get_serial_number
sub_get_product_id
sub_get_version

sub_eep_read
sub_eep_write

sub_i2c_freq
sub_i2c_config
sub_i2c_start
sub_i2c_stop
sub_i2c_scan
sub_i2c_read
sub_i2c_write

sub_bb_i2c_config
sub_bb_i2c_scan
sub_bb_i2c_read
sub_bb_i2c_write

sub_spi_config
sub_spi_transfer

sub_gpio_config
sub_gpio_read
sub_gpio_write

sub_gpiob_config
sub_gpiob_read
sub_gpiob_write

sub_edge_config
sub_edge_read

sub_fpwm_config
sub_fpwm_set

sub_adc_config
sub_adc_read
sub_adc_single

sub_lcd_write

sub_rs_set_config
sub_rs_get_config
sub_rs_timing
sub_rs_xfer

sub_fifo_config
sub_fifo_read
sub_fifo_write

3.1 Device Initialization

Functions

sub_find_devices
sub_open
sub_get_serial_number
sub_get_product_id

3.1.1 sub_find_devices

Synopsis

```
sub_device sub_find_devices( sub_device first )
```

Function scans USB devices currently connected to the host looking for SUB-20 device(s). If parameter ***first*** is NULL function will initiate new search, otherwise it will continue to search from the place it finished during last call.

Return value

Function returns next found SUB-20 device descriptor or NULL if no more devices were found. Returned value can be used as parameter for sub_open. Device descriptor is not a device handle required by API calls. Handle is returned by sub_open.

Example

```
sub_device dev=0;
while( dev = sub_find_devices(dev) )
{
   /* Check device serial number */
}
```

3.1.2 sub_open

Synopsis

```
sub_handle sub_open( sub_device dev )
```

Open SUB-20 device. If parameter **dev** is NULL function will try to open first available SUB-20 device. In this case it will call internally sub_find_devices(0). Otherwise function will try to open SUB-20 device referenced by **dev**. If your application intended to work with single SUB-20 device you can always call sub_open with **dev** =NULL.

Return value

On success function returns non zero handler that should be used in all subsequent calls to SUB-20 API functions. In case of error function returns NULL and set sub_errno

Example

```
    handle = sub_open( 0 );
    if( !handle )
    {
```

```
        printf("sub_open: %s\n", sub_strerror(sub_errno));
        return -1;
    }
```

3.1.3 sub_get_serial_number

Synopsis

```
int sub_get_serial_number( sub_handle hndl, char *buf, int sz)
```

Get serial number string descriptor

Parameters

- *buf - buffer to store descriptor
- sz - buffer size

Return value

On success function returns string descriptor size. Otherwise negative number.

Example

```
if( sub_get_serial_number(hndl, buf, sizeof(buf)) >= 0 )
    printf( "Serial Number : %s\n", buf );
else
{
    /* Error */
}
```

3.1.4 sub_get_product_id

Synopsis

```
int sub_get_product_id( sub_handle hndl, char *buf, int sz);
```

Get product ID string descriptor

Parameters

- *buf - buffer to store descriptor
- sz - buffer size

Return value

On success function returns string descriptor size. Otherwise negative number.

Example

```
if( sub_get_product_id(hndl, buf, sizeof(buf)) >= 0 )
    printf( "Product ID : %s\n", buf );
else
{
    /* Error */
}
```

3.1.5 sub_get_version

Synopsis

```
const struct sub_version* sub_get_version( sub_handle hndl );
```

Get version of shared (DLL) or static library, driver, SUB-20 device and bootloader.

Return value

Function fills internal structure with gathered versions and returns pointer to this structure.

Example

```
const struct sub_version* sub_ver;
sub_ver = sub_get_version( hndl );
```

3.1.6 sub_reset

Synopsis

```
int sub_reset( sub_handle hndl );
```

Reset SUB-20 device.

Return value

On success function returns 0. Otherwise error code.

3.2 Internal EEPROM Functions

Beginning from the firmware version 0.3.4 and library version 0.1.12.26, SUB-20 provides access to it's internal EEPROM (non volatile memory). EEPROM memory array size is 1KB, **endurance: 100,000 Write/Erase Cycles**. EEPROM Memory array can be freely used by application without any format or interdependency restrictions.
Access to EEPROM is gained by two simple functions:
sub_eep_read
sub_eep_write

3.2.1 sub_eep_read

Synopsis

```
int sub_eep_read( sub_handle hndl, int addr, char* buf, int sz )
```

Read **sz** bytes from internal EEPROM starting from address **addr**. Read data will be stored in **buf**.

Parameters

- addr - read start address
- buf - buffer to store read data
- sz - read size, max 64 bytes.

Return value

On success function returns 0. Otherwise error code.

3.2.2 sub_eep_write

Synopsis

```
int sub_eep_write( sub_handle hndl, int addr, char* buf, int sz )
```

Write **sz** bytes into internal EEPROM starting from address **addr**. Write data is taken from **buf**.

Parameters

- addr - write start address
- buf - buffer for data to be written
- sz - write size, max 64 bytes.

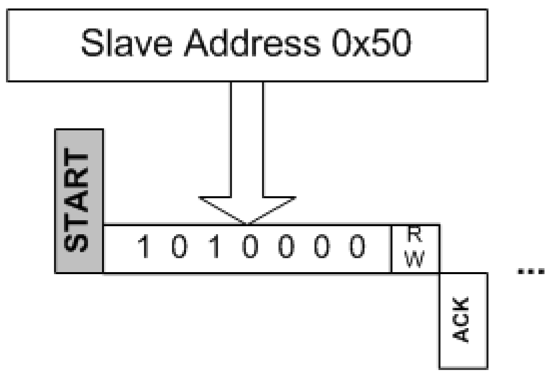
Return value

On success function returns 0. Otherwise error code.

3.3 I2C functions

Slave Address Convention

Slave address parameter referenced in I2C related functions is 7 bit number representing I2C slave device address without R/W bit. For example with slave address parameter equal 0x50 slave address stage of the I2C transaction will look like:



Acknowledge Polling

I2C EEPROM slave devices may enter internal memory write cycle. Once the internally-timed write cycle has started EEPROM inputs are disabled, acknowledge polling can be initiated. This involves sending a start condition followed by the device slave address and R/W bit. Only if the internal write cycle has completed will the EEPROM respond with an ACK, allowing the read or write sequence to continue. Acknowledge polling should be performed on application level. SUB-20 knows nothing about nature of I2C slave device it is connected to.

EEPROM Page Write

I2C EEPROMs are usually capable of no more then 64-byte page writes. The internal EEPROM data address lower 6 bits are internally incremented following the receipt of each data byte. The higher data address bits are not incremented, retaining the memory page row location. When the data address, internally generated, reaches the page boundary, the following byte is placed at the beginning of the same page. If more than 64 data bytes are transmitted to the EEPROM, the data address will “roll over” and previous data will be overwritten. The address “roll over” during write is from the last byte of the current page to the first byte of the same page.

High Speed Support

Beginning from FW version 0.3.2 and library version 0.1.12.24, SUB-20 provides High Speed (HS) I2C Master capabilities compliant to "I2C Bus Specification Rev 3.0". HS I2C Master functionality is available via sub_i2c_hs_rw API call. Timing characteristics can be found here HS I2C Master AC Characteristics

Slave Mode

SUB-20 can work in both I2C master and I2C slave modes. Detailed description of I2C slave mode can be found here: SUB-20 as I2C Slave.

Functions

3.3.1 sub_i2c_freq

Synopsis

```
int sub_i2c_freq( sub_handle hndl, int* freq )
```

Set and get SUB-20 I2C master clock frequency. If ***freq** is non zero I2C clock frequency will be set to value equal or close to ***freq**. SUB-20 has I2C frequency generator that can generate clock frequencies in range form 444,444 KHz down to 489 Hz. Function will calculate close possible frequency from the available range. If function succeed ***freq** will reflect the resulting I2C clock frequency. Some of available I2C clock frequencies are listed below

444444 Hz (Maximal)
421052 Hz
400000 Hz
...
100000 Hz
...
491 Hz
489 Hz (Minimal)

Parameters

- *freq - Desired frequency or zero. On return will be filled with resulting frequency

Return value

On success function returns 0. Otherwise error code.

3.3.2 sub_i2c_config

Synopsis

```
int sub_i2c_config( sub_handle hndl, int sa, int flags )
```

Configure SUB-20 I2C module.

Parameters

- sa - slave address for SUB-20 in I2C slave mode. See I2C Slave Address Convention and I2C Slave
- flags - set of below flags

I2C_GCE	Enable general call address (0x00) detection in I2C slave mode
I2C_DISABLE	Disable I2C module. (Any following I2C transaction will enable I2C module) . I2C_DISABLE is required to get full control over GPIO8,GPIO9 I/Os. While I2C module is enabled GPIO8,GPIO9 can only be used as inputs.

Return Value

On success function returns 0. Otherwise error code.

3.3.3 sub_i2c_start

Synopsis

```
int sub_i2c_start( sub_handle hndl )
```

Generate I2C start condition.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes, I2C Status

3.3.4 sub_i2c_stop

Synopsis

```
int sub_i2c_stop( sub_handle hndl )
```

Generate I2C stop condition.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes, I2C Status

3.3.5 sub_i2c_scan

Synopsis

```
int sub_i2c_scan( sub_handle hndl, int* slave_cnt, char* slave_buf )
```

Scan I2C bus looking for connected slave devices.

Parameters

- *slave_cnt - Buffer to store number of found slave devices
- slave_buf - Buffer to store found slave device addresses

Return value

On success function returns 0. Otherwise error code.

See also

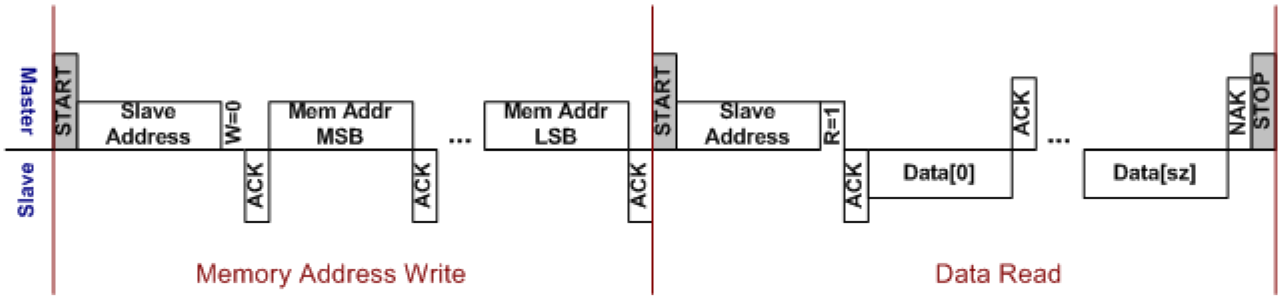
Error Codes, I2C Status

3.3.6 sub_i2c_read

Synopsis

```
int sub_i2c_read( sub_handle hndl, int sa, int ma, int ma_sz,
                                                  char* buf, int sz )
```

Perform complete I2C master read transaction with optional memory address write. Transaction will have following format:



If **ma_sz** is zero "Memory Address Write" stage will be skipped.
Function has no limitation of the read data size - **sz** parameter. However internal organization of the I2C slave device being read should be considered.

Parameters

- sa - Slave Address
- ma - Memory Address. Will be shifted out in "Memory Address Write" stage MSB first
- ma_sz - Memory Address size bytes
- buf - Buffer to store read data
- sz - Read data size bytes.

Return value

On success function returns 0. Otherwise error code.

See also

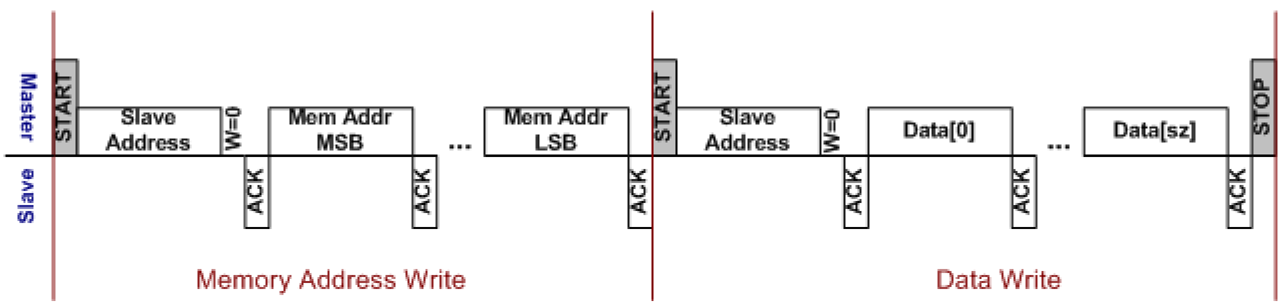
Error Codes, I2C Status

3.3.7 sub_i2c_write

Synopsis

```
int sub_i2c_write( sub_handle hndl, int sa, int ma, int ma_sz,
                                                 char* buf, int sz )
```

Perform complete I2C master write transaction with optional memory address write. Transaction will have following format:



If **ma_sz** is zero "Memory Address Write" stage will be skipped.
Function has no limitation of the write data size - **sz** parameter. However internal organization of the I2C slave device being written should be considered especially Acknowledge Polling and EEPROM Page Write.

Parameters

- sa - Slave Address
- ma - Memory Address. Will be shifted out in "Memory Address Write" stage MSB first
- ma_sz - Memory Address size bytes
- buf - Buffer for data to be written
- sz - Write data size bytes

Return value

On success function returns 0. Otherwise error code.

See also

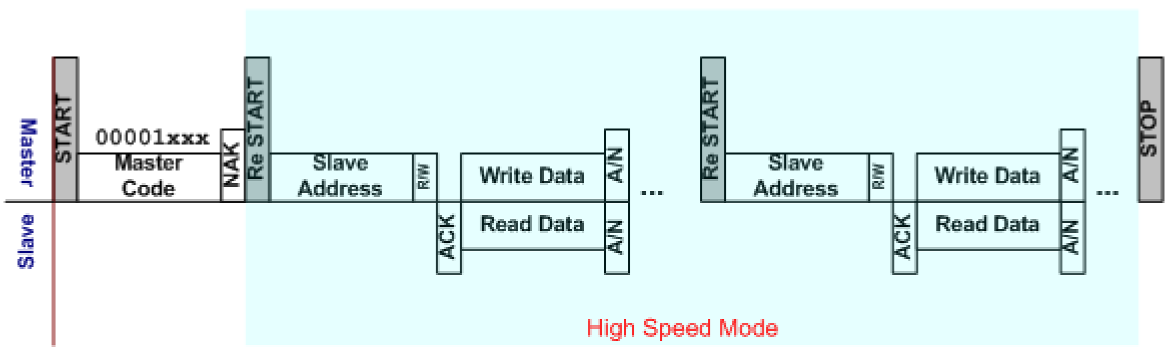
Error Codes, I2C Status

3.3.8 sub_i2c_hs_rw

Synopsis

```
int sub_i2c_hs_rw( sub_handle hndl, int mcode, int count,
                                    struct sub_i2c_hs_xfer* px );
```

Perform High Speed (HS) I2C Master read/write transaction(s) with prior master code transmit according to I2C-bus specification. The whole session has the following format:



At the beginning of the session both master and slave are in Fast Speed or Standard I2C mode. Master issues START condition and transmits Master Code 00001xxx, where xxx is under SW control via **mcode** parameter. (Master Code 00001000 should not be used as it is reserved by I2C-bus specification for future extensions). Master Code will not be acknowledged - NAK will be generated. At the end of NAK I2C-bus is in HS mode. Now master can perform HS read/write transactions by issuing REPEATED START condition followed by Slave Address with R/W bit and data byte(s) being read/written according to R/W bit. To exit HS mode master issues STOP condition. A number of HS transactions can be performed within single HS mode session (it is recommended by I2C-bus specification).

Parameters

- mcode - Master Code lower 3 bits
- count - number of HS mode read/write transactions to perform
- px - pointer to array of **count** transaction descriptors of type `struct sub_i2c_hs_xfer`

```
struct sub_i2c_hs_xfer
{
    int     sa;
    int     r_w;
    int     sz;
    int     act_sz;
    char    data[64];
    int     status;
```

```
};
```

where:

- sa - Slave Address
- r_w - 0-write, 1-read
- sz - size in bytes of data to read/write
- act_sz - actual read/write size. This parameter will be filled by function
- data - 64 bytes data buffer. For read transaction function will attempt to read **sz** bytes into buffer, for write transaction function will attempt to write **sz** bytes from buffer. Maximal transaction size is 64 bytes
- status - I2C transaction result (see I2C Status).

Return value

On success function returns 0. Otherwise error code. The "I2C Error" return value means that there was a failure at the master code transmit stage. Results of the HS transactions are reported in **status** field of the corresponding transaction descriptor.

Example

```
/*
* Master Code 00001001
* Write 4 bytes to slave 0x44
* Write 2 bytes to slave 0x50
* Read 8 bytes from slave 0x44
*/
struct sub_i2c_hs_xfer xfer[3] =
{
    { 0x44, 0, 4, 0, "\x10\x11\x12\x13", 0 },
    { 0x50, 0, 2, 0, "\x20\x30", 0 },
    { 0x44, 1, 8, 0, "", 0 }
};

rc = sub_i2c_hs_rw( hndl, 0x01, 3, xfer, stat );
```

3.3.9 I2C Status

The status of last I2C operation successful or not successful is stored in **sub_i2c_status** global variable defined in *libsub.h* as

```
extern int   sub_i2c_status;
```

Following statuses are available:

0x00	No errors
0x08	START condition transmitted
0x10	Repeated START condition transmitted
0x18	SLA+W transmitted; ACK received
0x20	SLA+W transmitted; NAK received
0x28	Data byte transmitted; ACK received
0x30	Data byte transmitted; NAK received
0x38	Arbitration lost in SLA; or NAK received
0x40	SLA+R transmitted; ACK received
0x48	SLA+R transmitted; NAK received
0x50	Data received; ACK has been returned
0x58	Data received; NAK has been returned
0xE0	Arbitration lost
0xE1	Arbitration lost in START
0xE2	Arbitration lost in STOP
0xE3	Arbitration lost in read ACK
0xE4	Arbitration lost in read NAK
0xE5	Arbitration lost in write
0xF8	Unknown error

0xFF Illegal START or STOP condition

3.4 Bit-bang I2C Masters

Beginning from the FW version 0.2.6 and library version 0.1.12.16, SUB-20 provides additional four independent I2C master channels on GPIO24..31 compatible with I2C specification 3.0.

Header Pin	GPIO	Alternative I2C Master Function
9	GPIO24	SCLx0
10	GPIO25	SDAx0
11	GPIO26	SCLx1
12	GPIO27	SDAx1
13	GPIO28	SCLx2
14	GPIO29	SDAx2
15	GPIO30	SCLx3
16	GPIO31	SDAx3

Each channel can operate in Fast Plus (1000KHz), Fast (400KHz) and Standard (100KHz) modes. In Fast and Standard modes I2C masters support clock stretching.

SDAx, SCLx Pull-ups

Considering the fact that bit-bang I2C channels share the GPIO pins with other SUB-20 modules, there are no internal (mounted on SUB-20 board) pull-ups on these channels. **Pull-ups must be provided on target board**.

Clock Stretching

In Fast and Standard modes bit-bang I2C masters support clock stretching. Clock stretching pauses a transaction by holding the SCL line LOW. The transaction cannot continue until the line is released HIGH again. Slaves can hold the SCL line LOW to force the
master into a wait state until the slave is ready.

Functions

sub_bb_i2c_config
sub_bb_i2c_scan
sub_bb_i2c_read
sub_bb_i2c_write

3.4.1 sub_bb_i2c_config

Synopsis

```
int sub_bb_i2c_config( sub_handle hndl, int mode, int stretch_ms )
```

Configure SUB-20 Bit-bang I2C Masters module. Configuration is common for all Bit-bang I2C channels.

Parameters

- mode - Bit-bang I2C Master mode. One of the following flags:

BB_I2C_FAST_P	Fast Plus mode 1000KHz
BB_I2C_FAST	Fast mode 400KHz
BB_I2C_STD	Standard mode 100KHz

- stretch_ms - Clock stretching timeout up to 4194ms. Clock stretching feature is available in Fast and Standard modes. If SCL line is hold LOW by I2C slave device longer than **stretch_ms** timeout, Bit-bang I2C function will fail and I2C Status will be set to 0xE0.

Return Value

On success function returns 0. Otherwise error code.

Compatibility

FW version 0.2.6 or grater
Library version 0.1.12.16 or grater

3.4.2 sub_bb_i2c_scan

Synopsis

```
int sub_bb_i2c_scan( sub_handle hndl, int channel, int* slave_cnt,
                                                    char* slave_buf )
```

Scan I2C bus looking for connected slave devices.

Parameters

- channel - Bit-bang I2C Master channel 0..3
- *slave_cnt - Buffer to store number of found slave devices
- slave_buf - Buffer to store found slave device addresses

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes, I2C Status

Compatibility

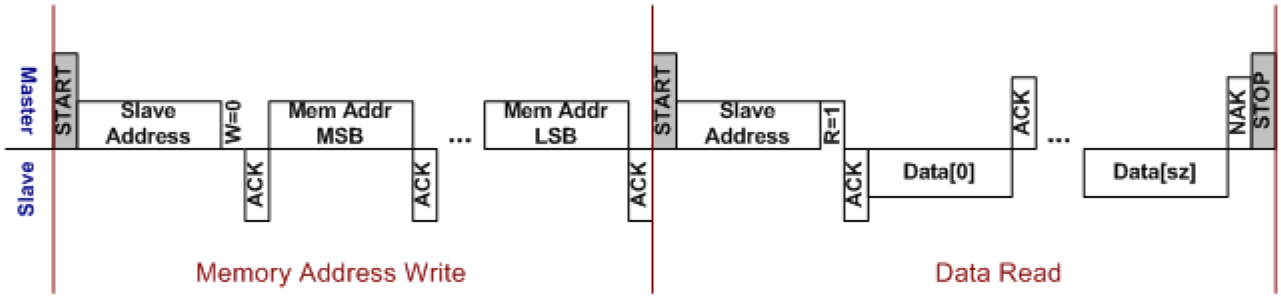
FW version 0.2.6 or grater
Library version 0.1.12.16 or grater

3.4.3 sub_bb_i2c_read

Synopsis

```
int sub_bb_i2c_read( sub_handle hndl, int channel, int sa, int ma, int ma_sz,
                                                    char* buf, int sz )
```

Perform complete I2C master read transaction with optional memory address write. Transaction will have following format:



If **ma_sz** is zero "Memory Address Write" stage will be skipped.
Function has no limitation of the read data size - **sz** parameter. However internal organization of the I2C slave device being read should be considered.

Parameters

- channel - Bit-bang I2C Master channel 0..3
- sa - Slave Address
- ma - Memory Address. Will be shifted out in "Memory Address Write" stage MSB first
- ma_sz - Memory Address size bytes
- buf - Buffer to store read data
- sz - Read data size bytes.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes, I2C Status

Compatibility

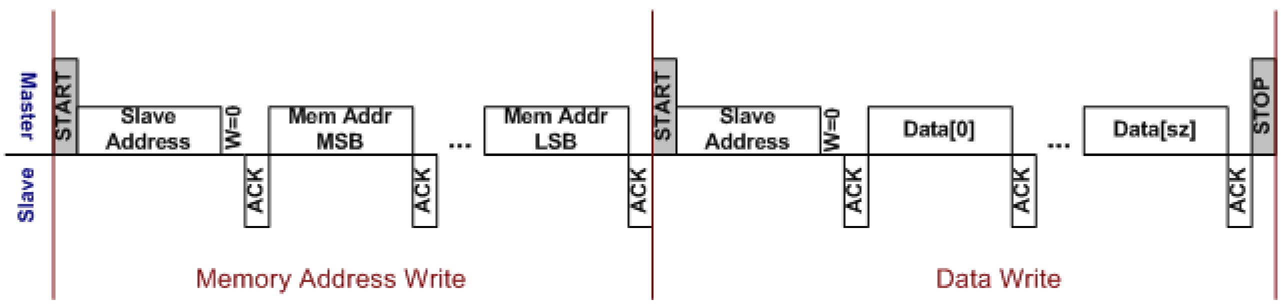
FW version	0.2.6 or grater
Library version	0.1.12.16 or grater

3.4.4 sub_bb_i2c_write

Synopsis

```
int sub_bb_i2c_write( sub_handle hndl, int channel, int sa, int ma, int ma_sz,
                                                       char* buf, int sz )
```

Perform complete I2C master write transaction with optional memory address write. Transaction will have following format:



If **ma_sz** is zero "Memory Address Write" stage will be skipped.
Function has no limitation of the write data size - **sz** parameter. However internal organization of the I2C slave device being written should be considered especially Acknowledge Polling and EEPROM Page Write.

Parameters

- channel - Bit-bang I2C Master channel 0..3
- sa - Slave Address
- ma - Memory Address. Will be shifted out in "Memory Address Write" stage MSB first
- ma_sz - Memory Address size bytes
- buf - Buffer for data to be written
- sz - Write data size bytes

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes, I2C Status

Compatibility

FW version 0.2.6 or grater
Library version 0.1.12.16 or grater

3.5 SPI Functions

SPI Pins Configuration

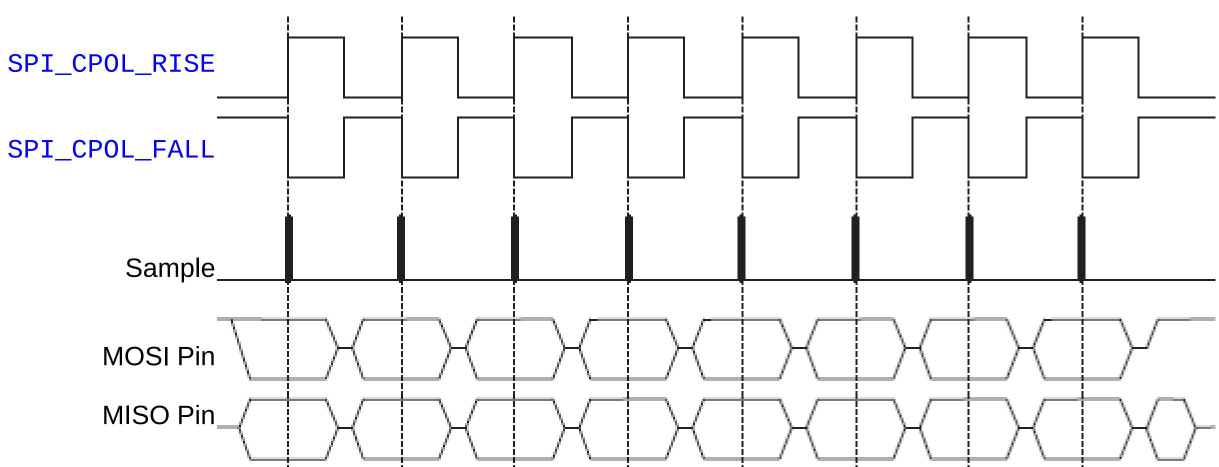
After SPI is enabled SUB-20 SPI module will take control over SPI pins.

Pin	Function	Master mode	Slave mode	Disabled
SCK	SPI Master Clock	Output	Input	HiZ
MOSI	Master Out Slave In	Output	Input	
MISO	Master In Slave Out	Input	Output**	
SS0	Slave Select 0	Output Hi (Configurable by sub_spi_transfer)	Input	
SS1 SS2 SS3 SS4	Slave Select 1..4	HiZ (Configurable by sub_spi_transfer)	Not in use	

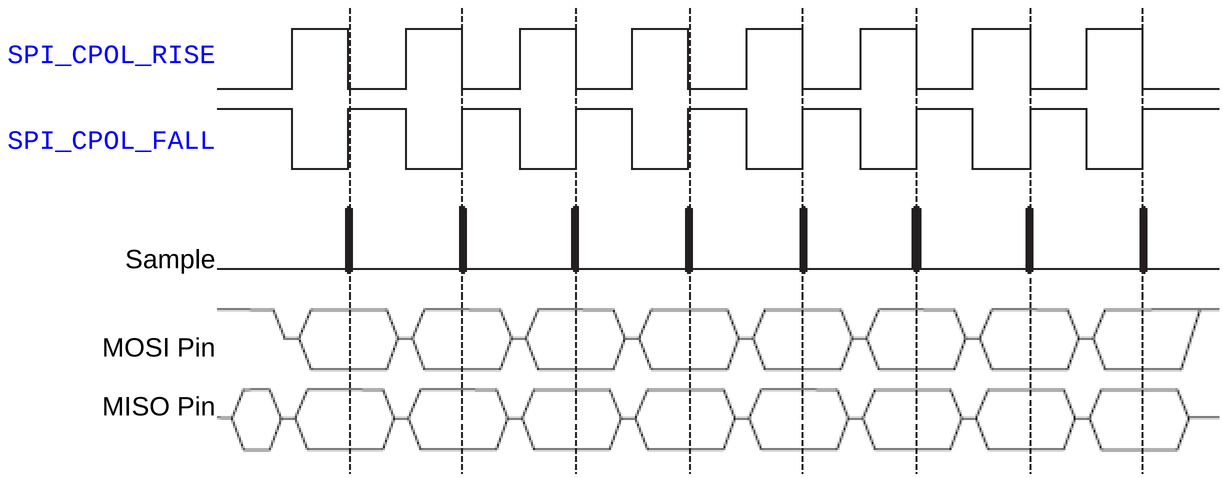
*If SPI is disabled all SPI pins are in HiZ.
** In SPI slave mode MISO is output only when SS0 is low

SPI SCK Polarity and Phase explanations

SPI Transfer format with SPI_SMPL_SETUP



SPI Transfer format with SPI_SETUP_SMPL



Functions

sub_spi_config
sub_spi_transfer
sub_spi_transfer_ess
sub_sdio_transfer

3.5.1 sub_spi_config

Synopsis

```
int sub_spi_config( sub_handle hndl, int cfg_set, int* cfg_get )
```

Configure SUB-20 SPI module or read current configuration. If ***cfg_get** is NULL function will configure SPI according to the **cfg_set** parameter. Otherwise it will read current SPI configuration into ***cfg_get**

Parameters

- cfg_set - Desired SPI configuration. This parameter is effective only if ***cfg_get** is NULL. **cfg_set** should be assembled as a combination of below flags

SPI_ENABLE	Enable SUB-20 SPI module.
SPI_SLAVE	SPI module is in Slave mode. sub_spi_transfer can work only when SPI module is in Master mode
SPI_CPOL_RISE	SCK is low when idle. See SPI Polarity and Phase explanations
SPI_CPOL_FALL	SCK is high when idle. See SPI Polarity and Phase explanations
SPI_SMPL_SETUP	Sample data on leading SCK edge, setup on trailing. See SPI Polarity and Phase
SPI_SETUP_SMPL	Setup data on leading SCK edge, sample on trailing. See SPI Polarity and Phase

Below flags are relevant only for SPI master mode (SPI_SLAVE not set)

SPI_LSB_FIRST	Transmit LSB first
SPI_MSB_FIRST	Transmit MSB first

SPI_CLK_8MHZ	SPI SCK frequency 8 MHz
SPI_CLK_4MHZ	SPI SCK frequency 4 MHz
SPI_CLK_2MHZ	SPI SCK frequency 2 MHz
SPI_CLK_1MHZ	SPI SCK frequency 1 MHz
SPI_CLK_500KHZ	SPI SCK frequency 500 KHz
SPI_CLK_250KHZ	SPI SCK frequency 250 KHz
SPI_CLK_125KHZ	SPI SCK frequency 125 KHz

Beginning from FW version 0.3.3 and library version 0.1.12.25 lower SPI frequencies (down to 4KHz) are supported for SPI Master transfer with sub_spi_transfer function. Special `SPI_CLK_HZ(FHZ)` macro defined in libsub.h, where **FHZ** parameter denotes SPI frequency in Hz. Supported frequencies range is 100.000Hz .. 4000Hz. Exact frequency may differ from requested due to integer rounding in division.

SPI_CLK_HZ(FHZ)	SPI SCK frequency 100.000Hz .. 4000Hz

- *cfg_get - Pointer to store current configuration read from SUB-20 device or NULL

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Read current SPI configuration */
sub_spi_config( hndl, 0, &spi_config );

/* Configure SPI */
sub_spi_config( hndl,
SPI_ENABLE|SPI_CPOL_RISE|SPI_SMPL_SETUP|SPI_MSB_FIRST|SPI_CLK_4MHZ, 0 );

/* Disable SPI */
sub_spi_config( hndl, 0, 0 );
```

Compatibility

SPI_CLK_HZ(FHZ)	FW version	0.3.3 or grater
	Library version	0.1.12.25 or grater

3.5.2 sub_spi_transfer

Synopsis

```
int sub_spi_transfer( sub_handle hndl, char* out_buf, char* in_buf,
                                                   int sz, int ss_config )
```

Perform SPI master transaction. Depending on **out_buf** and **in_buf** parameters transaction can be either read (**out_buf**==0), write (**in_buf**==0) or read-write (both **in_buf** and **out_buf** are non zero).

Parameters

- out_buf - Output data buffer or NULL. If NULL there will be no write transaction and MOSI pin will stay unchanged.
- in_buf - Input buffer to store read data or NULL. If NULL there will be no read transaction and data on MISO pin will be ignored.
- sz - Transaction size
- ss_config - Determines selection and operation of SS pin. **ss_config** value must be created with macro **SS_CONF**(SS_N,SS_MODE) ,

where **SS_N** is SS pin number and **SS_MODE** is one of the following flags:

SS_H	SS goes high and stays high during and after transaction
SS_HL	SS goes high and stays high during first byte transfer, after that it goes low
SS_HHL	SS goes high and stays high during first 2 bytes transfer, after that it goes low
SS_HHHL	SS goes high and stays high during first 3 bytes transfer, after that it goes low
SS_HHHHL	SS goes high and stays high during first 4 bytes transfer, after that it goes low
SS_HI	SS goes high and stays high during entire transfer, after that it goes low
SS_L	SS goes low and stays low during and after transaction
SS_LH	SS goes low and stays low during first byte transfer, after that it goes high
SS_LLH	SS goes low and stays low during first 2 bytes transfer, after that it goes high
SS_LLLH	SS goes low and stays low during first 3 bytes transfer, after that it goes high
SS_LLLLH	SS goes low and stays low during first 4 bytes transfer, after that it goes high
SS_LO	SS goes low and stays low during entire transfer, after that it goes high
SS_HiZ	SS stays in HiZ mode (Except SS0)

If **ss_config** is zero there will be no SS activity (changes).

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Write 10 bytes. Use SS0 low */
rc = sub_spi_transfer( hndl, out, 0, 10, SS_CONF(0,SS_LO) );

/* Transfer 10 bytes out and 10 bytes in. Use SS2 high */
rc = sub_spi_transfer( hndl, out, in, 10, SS_CONF(2,SS_HI) );

/* Read 10 bytes. No SS */
rc = sub_spi_transfer( hndl, 0, in, 10, 0 );
```

3.5.3 sub_spi_transfer_ess

Synopsis

```
int sub_spi_transfer_ess( sub_handle hndl, char* out_buf, char* in_buf,
                          int sz, char* ess_str )
```

Perform SPI master transaction with Extended Slave Select control (ESS). Depending on **out_buf** and **in_buf** parameters transaction can be either read (**out_buf**==0), write (**in_buf**==0) or read-write (both **in_buf** and **out_buf** are non zero).

Parameters

- out_buf - Output data buffer or NULL. If NULL there will be no write transaction and MOSI pin will stay unchanged.
- in_buf - Input buffer to store read data or NULL. If NULL there will be no read transaction and data on MISO pin will be ignored.
- sz - Transaction size
- ess_str - string defining SS selection and activity during SPI transfer. Characters of the **ess_str** have following meaning

0	Select SS0	
1	Select SS1	
2	Select SS2	
3	Select SS3	
4	Select SS4	
H	SS High	Byte Transfer

L	SS Low	Byte Transfer
^	SS High	No Byte Transfer
v	SS Low	No Byte Transfer
-	Small delay without SS changes	No Byte Transfer

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Write 5 bytes with SS0 Low. Make SS0 High to Low pulse after each byte */
rc = sub_spi_transfer( hndl, out, 0, 5, "0L^-vL^-vL^-vL^-vL^-v" );

/* Transfer 10 bytes with SS High. Use SS1 for first 5 bytes and SS2 for last */
rc = sub_spi_transfer( hndl, out, in, 10, "1HHHHHv-2HHHHHv" );

/* Read 10 bytes. SS0 High for first 2 bytes and low after that */
rc = sub_spi_transfer( hndl, 0, in, 10, "0HHv" );
```

Compatibility

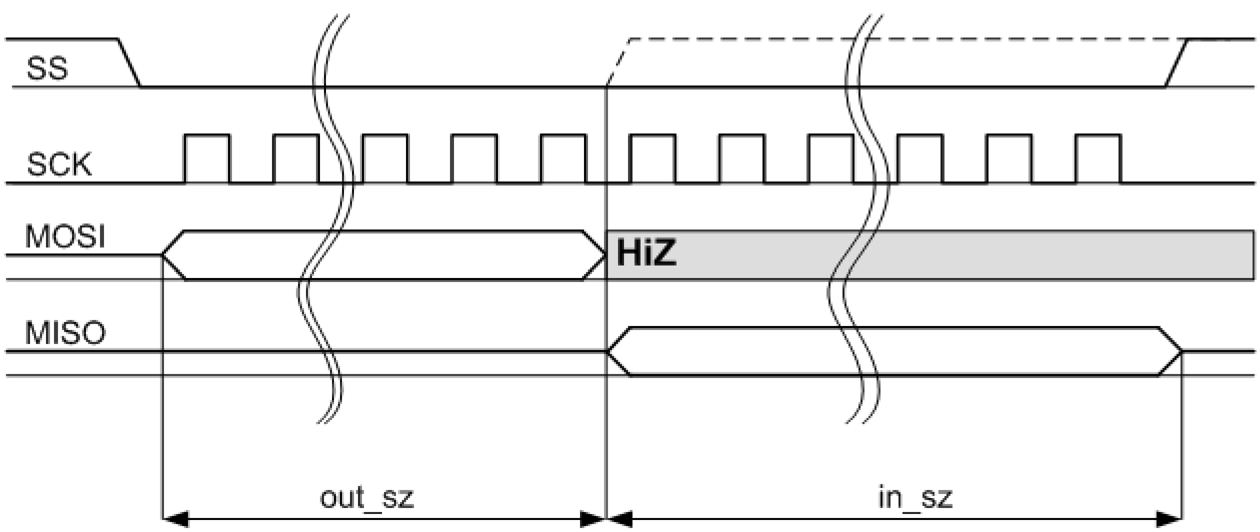
FW version	0.2.9 or grater
Library version	0.1.12.19 or grater

3.5.4 sub_sdio_transfer

Synopsis

```
int sub_sdio_transfer( sub_handle hndl, char* out_buf, char* in_buf,
                                        int out_sz, int in_sz, int ss_config )
```

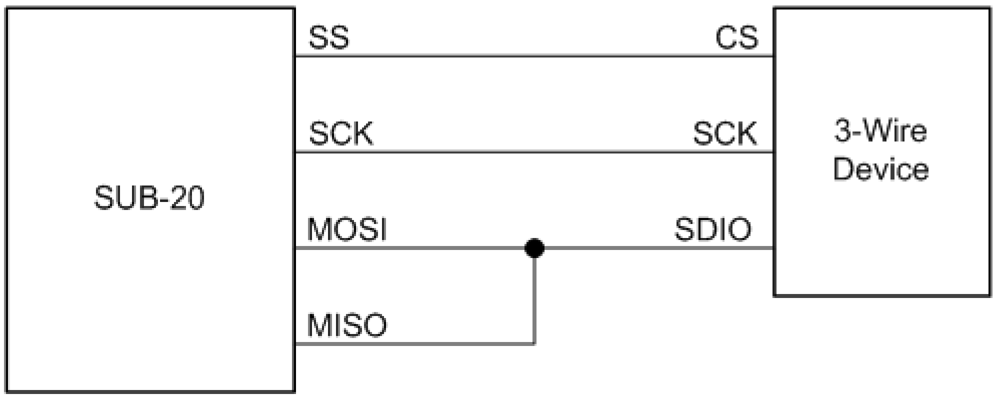
Perform 3-Wire compatible SPI master transaction like one shown below.



1. SS changes according to **ss_config** parameter.
2. **out_sz** bytes are transmitted via MOSI. MOSI will be in output state during transfer.
3. MOSI state is changed to HiZ after last bit transmitted. If required SS changes.
4. **in_sz** bytes received from MISO input. MOSI left in HiZ state.
5. SS changes according to **ss_config** parameter

SPI Clock - SCK phase and polarity can be configured with sub_spi_config.

Above implementation of SPI transaction allows using of the SUB-20 as 3-Wire SPI master. In this case connection to 3-Wire device should be as following:



Parameters

- out_buf - Output data buffer.
- in_buf - Input buffer to store read data.
- out_sz - Number of bytes to transmit in range 0..60.
- in_sz - Number of bytes to receive in range 0..60.
- ss_config - Determines selection and operation of SS pin. **ss_config** value must be created with macro **SS_CONF**(SS_N,SS_MODE) ,

where **SS_N** is SS pin number and **SS_MODE** is one of the following flags:

SS_HL	SS goes high and stays high during transmit stage, after that it goes low
SS_HI	SS goes high and stays high during entire transaction (transmit and receive), after that it goes low
SS_LH	SS goes low and stays low during transmit stage, after that it goes high
SS_LO	SS goes low and stays low during entire transaction (transmit and receive), after that it goes high
SS_HiZ	SS stays in HiZ mode (Except SS0)

If **ss_config** is zero there will be no SS activity (changes).

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Transmit one byte and receive 3 bytes */
/* SS1 goes low during transaction */
sub_sdio_transfer( hndl, out_buf, in_buf, 1, 3, SS_CONF(1,SS_LO) );
```

Compatibility

FW version	0.1.9 or grater
Library version	0.1.12.11 or grater

3.6 MDIO Functions

MDIO is a Management Data Input/Output Interface defined in IEEE 802.3 Clause 22 and extended in Clause 45. It is two signal based interface between Station Management (SUB-20 in our case) and a Physical Layer device (PHY). Where a PHY, or grouping of PHY's, is an individually manageable entity, known as an MDIO Manageable Device (MMD).

Signals

- MDC - management data clock. MDC is sourced by SUB-20 to the PHY as the timing reference for transfer of information on the MDIO signal.
- MDIO - management data input/output. MDIO is a bidirectional signal between PHY and the SUB-20 It is used to transfer control information and status between the PHY and the SUB-20. Control information is driven by the SUB-20 synchronously with respect to MDC and is sampled synchronously by the PHY. Status information is driven by the PHY synchronously with respect to MDC and is sampled synchronously by the SUB-20.

Frame Format

SUB-20 supports both MDIO frame formats defined in IEEE 802.3 Clause 22 and Clause 45.

- Clause 22 MDIO frame format

	PREAMBLE	ST	OP	PHYAD	REGAD	TA	DATA
READ	111...111	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD
WRITE	111...111	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD

- Clause 45 MDIO frame format

	PREAMBLE	ST	OP	PRTAD	DEVAD	TA	ADDRESS / DATA
Addres	111...111	00	00	PPPPP	EEEEE	10	AAAAAAAAAAAAAAAA
Write	111...111	00	01	PPPPP	EEEEE	10	DDDDDDDDDDDDDDDD
Read	111...111	00	11	PPPPP	EEEEE	Z0	DDDDDDDDDDDDDDDD
Post-read-increment-address	111...111	00	10	PPPPP	EEEEE	Z0	DDDDDDDDDDDDDDDD

Channels

Beginning from FW version 0.2.8 and library version 0.1.12.18, additional MDIO1 channel is available on SPI header (see SPI Header). To access MDIO1 channel use sub_mdio_xfer_ex function with **channel**=1.

CFP MSA Support

Beginnig from FW version 0.3.2 and library version 0.1.12.24, SUB-20 supports CFP MSA compatible transactions at 4MHz MDC frequency. See sub_mdio_xfer_ex.

Functions

sub_mdio22
sub_mdio45
sub_mdio_xfer, sub_mdio_xfer_ex

3.6.1 sub_mdio22

Synopsis

```
int sub_mdio22( sub_handle hndl, int op, int phyad, int regad,
                                             int data, int* content )
```

Generate IEEE 802.3 Clause 22 MDIO READ or WRITE frame. Frame format is shown here: Clause 22 MDIO frame.

Parameters

- op - operation code

SUB_MDIO22_READ	READ operation
SUB_MDIO22_WRITE	WRITE operation

- phyad - 5 bit PHY address
- regad - 5 bit register address
- data - 16 bit data for WRITE operation
- *content - 16 bit register content placeholder for READ operation

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Read register 0x12 in PHY 1 */
rc = sub_mdio22( hndl, SUB_MDIO22_READ, 0x01, 0x12, 0, &content );
/* Write register 0x5 in PHY 2 */
rc = sub_mdio22( hndl, SUB_MDIO22_WRITE, 0x02, 0x05, 0x55AA, 0 );
```

Compatibility

FW version	0.2.1 or grater
Library version	0.1.12.12 or grater

3.6.2 sub_mdio45

Synopsis

```
int sub_mdio45( sub_handle hndl, int op, int prtad, int devad,
                                             int data, int* content )
```

Generate IEEE 802.3 Clause 45 MDIO frame. Frame format is shown here: Clause 45 MDIO frame.

Parameters

- op - operation code

SUB_MDIO45_ADDR	ADDRESS operation
SUB_MDIO45_WRITE	WRITE operation
SUB_MDIO45_PRIA	POST-READ-INCREMENT-ADDRESS operation
SUB_MDIO45_READ	READ operation

- prtad - 5 bit port address
- devad - 5 bit device address
- data - 16 bit address or data for ADDRESS or WRITE operation

- *content - 16 bit register content placeholder for READ or POST-READ-INCREMENT-ADDRESS operation

Return value

On success function returns 0. Otherwise error code.

Compatibility

FW version	0.2.1 or grater
Library version	0.1.12.12 or grater

3.6.3 sub_mdio_xfer, sub_mdio_xfer_ex

Synopsis

```
int   sub_mdio_xfer( sub_handle hndl, int count, union sub_mdio_frame* mdios )
int   sub_mdio_xfer_ex( sub_handle hndl, int channel, int count,
                                       union sub_mdio_frame* mdios )
```

Generate a sequence of independent MDIO frames. Frames in sequence can be Clause 22 or Clause 45 format with different operations and addresses.

Parameters

- channel - mdio channel (see MDIO Channels). Optional SUB_CFP_MSA flag applied to **channel** (`channel|SUB_CFP_MSA`) will force SUB-20 to generate CFP MSA compatible transaction(s) at 4MHz MDC frequency.
- count - number of frames to generate (currently up to 15).
- *mdios - array of **count** sub_mdio_frame unions

```
union sub_mdio_frame
{
   struct
   {
      int      op;
      int      phyad;
      int      regad;
      int      data;
   }clause22;
   struct
   {
      int      op;
      int      prtad;
      int      devad;
      int      data;
   }clause45;
};
```

Frame operation is defined by **op** field value.

clause22.op can be

SUB_MDIO22_READ	READ operation
SUB_MDIO22_WRITE	WRITE operation

clause45.op can be

SUB_MDIO45_ADDR	ADDRESS operation
SUB_MDIO45_WRITE	WRITE operation
SUB_MDIO45_PRIA	POST-READ-INCREMENT-ADDRESS operation

SUB_MDIO45_READ READ operation

For READ and POST-READ-INCREMENT-ADDRESS operations **clause22.data** or **clause45.data** will be filled with data read from PHY.

Return value

On success function returns 0. Otherwise error code.

Example

```
union sub_mdio_frame mdios[2];

mdios[0].clause22.op = SUB_MDIO22_READ;
mdios[0].clause22.phyad = 0x01;
mdios[0].clause22.regad = 0x12;

mdios[1].clause45.op = SUB_MDIO45_ADDR;
mdios[1].clause45.prtad = 0x04;
mdios[1].clause45.devad = 0x02;
mdios[1].clause45.data = 0x55AA;

rc = sub_mdio_xfer( hndl, 2, mdios );
```

Compatibility

sub_mdio_xfer

FW version	0.2.1 or grater
Library version	0.1.12.12 or grater

sub_mdio_xfer_ex

FW version	0.2.8 or grater
Library version	0.1.12.18 or grater

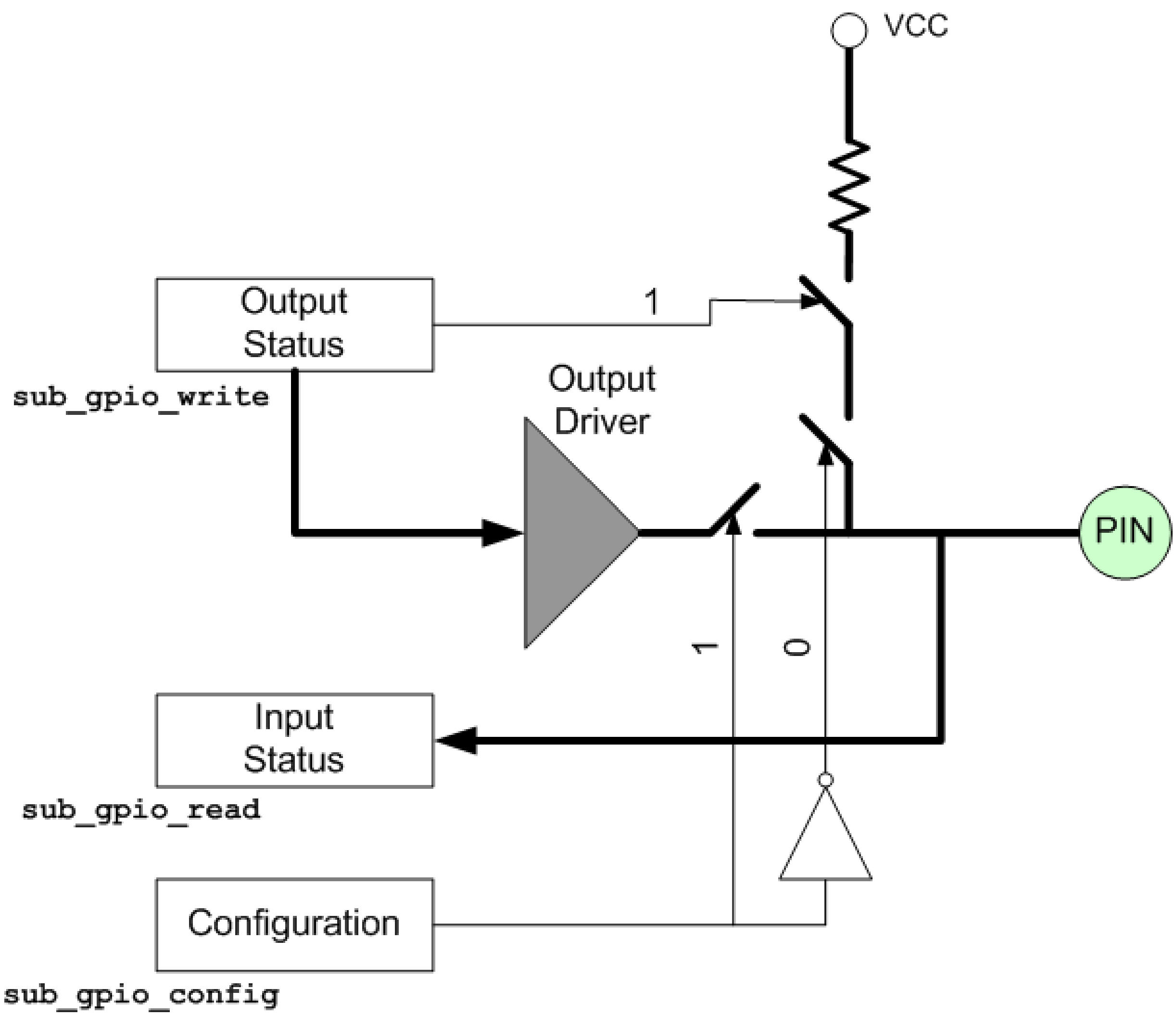
SUB_CFP_MSA

FW version	0.3.2 or grater
Library version	0.1.12.24 or grater

3.7 GPIO Functions

GPIO Functional Description

SUB-20 GPIO can be in input or output state. GPIO state is defined by configuration bit. In output state GPIO will drive high or low level depending on output status. In input state GPIO can be pulled high by internal weak pull-up resistor.



GPIO Pin configuration

Configuration	State	Output Status	
		"0"	"1"
"0"	Input state	HiZ	pull-up
"1"	Output state	LOW	HIGH

GPIOB

GPIOB are located on SPI header (see SPI Header). GPIOB module shares I/O pins with SPI module. If SPI is enabled MISO,MOSI,SCK and SS0 pins are partially controlled by SPI module, although they can be accessed by GPIOB functions with restrictions.
Special care should be taken for SUB-Lxxx models where SPI Level Converters are installed. In this case **GPIOB pins direction should respect direction of the corresponding Level Converter** to prevent short circuit on this pins.

Functions

sub_gpio_config
sub_gpio_read
sub_gpio_write
sub_gpiob_config

sub_gpiob_read
sub_gpiob_write

3.7.1 sub_gpio_config

Synopsis

```
int   sub_gpio_config( sub_handle hndl, int set, int* get, int mask )
```

Configure GPIO state (direction) as input or output.

Parameters

- set - Bits 0..31 of **set** parameter correspond to GPIO0..GPIO31 configuration bits. If GPIOn configuration bit is "1" then GPIOn direction is output, otherwise it is input.
- *get - Pointer to store current GPIO configuration read from SUB-20.
- mask - Bit in **set** parameter will take effect only if corresponding **mask** bit is "1". With **mask**=0 function will only read current GPIO configuration.

Return value

On success function returns 0. Otherwise error code.

Example

```
/* gpio0..6 - input, gpio7 - output */
rc = sub_gpio_config( hndl, 0x00000080, &config, 0x000000FF );

/* read gpio configuration */
rc = sub_gpio_config( hndl, 0, &config, 0 );
```

See also

Error Codes

3.7.2 sub_gpio_read

Synopsis

```
int   sub_gpio_read( sub_handle hndl, int* get )
```

Read GPIO input status. Function reads logic value "1"-high, "0"-low directly from GPIO pin.

Parameters

- *get - Pointer to store received GPIO input status. Bits 0..31 of ***get** correspond to GPIO0..GPIO31 input statuses.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes

3.7.3 sub_gpio_write

Synopsis

```
int   sub_gpio_write( sub_handle hndl, int set, int* get, int mask )
```

Set GPIO output status. For GPIO in output state function will set output driver to drive "1"-high, "0"-low. For

GPIO in input state function will "1"-enable, "0"-disable weak pull-up on corresponding GPIO pin.

Parameters

- set - Bits 0..31 of **set** parameter correspond to GPIO0..GPIO31 output statuses.
- *get - Pointer to store current GPIO output status read from SUB-20.
- mask - Bit in **set** parameter will take effect only if corresponding **mask** bit is "1". With **mask**=0 function will only read current GPIO output status.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes

3.7.4 sub_gpiob_config

Synopsis

```
int   sub_gpiob_config( sub_handle hndl, int set, int* get, int mask )
```

Configure GPIOB state (direction) as input or output.

Parameters

- set - Bits 0..7 of **set** parameter correspond to GPIOB0..GPIOB7 configuration bits. If GPIOBn configuration bit is "1" then GPIOBn direction is output, otherwise it is input.
- *get - Pointer to store current GPIOB configuration read from SUB-20.
- mask - Bit in **set** parameter will take effect only if corresponding **mask** bit is "1". With **mask**=0 function will only read current GPIOB configuration.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes

Compatibility

FW version	0.2.7 or grater
Library version	0.1.12.17 or grater

3.7.5 sub_gpiob_read

Synopsis

```
int   sub_gpiob_read( sub_handle hndl, int* get )
```

Read GPIOB input status. Function reads logic value "1"-high, "0"-low directly from GPIOB pin.

Parameters

- *get - Pointer to store received GPIOB input status. Bits 0..7 of ***get** correspond to GPIOB0..GPIOB7 input statuses.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes

Compatibility

FW version 0.2.7 or grater
Library version 0.1.12.17 or grater

3.7.6 sub_gpiob_write

Synopsis

```
int   sub_gpiob_write( sub_handle hndl, int set, int* get, int mask )
```

Set GPIOB output status. For GPIOB in output state function will set output driver to drive "1"-high, "0"-low. For GPIO in input state function will "1"-enable, "0"-disable weak pull-up on corresponding GPIOB pin.

Parameters

- set - Bits 0..7 of **set** parameter correspond to GPIOB0..GPIOB7 output statuses.
- *get - Pointer to store current GPIOB output status read from SUB-20.
- mask - Bit in **set** parameter will take effect only if corresponding **mask** bit is "1". With **mask**=0 function will only read current GPIOB output status.

Return value

On success function returns 0. Otherwise error code.

See also

Error Codes

Compatibility

FW version 0.2.7 or grater
Library version 0.1.12.17 or grater

3.7.7 Edge Detector

Edge Detector description

SUB-20 Edge detector module (EDGE) is capable to detect falling, rising or "any change" edge on the GPIO pins listed below.

GPIO Header Pin	GPIO	Edge Number
1	GPIO8	EDG0
2	GPIO9	EDG1
3	GPIO10	EDG2
4	GPIO11	EDG3
13	GPIO28	EDG4
14	GPIO29	EDG5
15	GPIO30	EDG6
16	GPIO31	EDG7

Pulses on EDG7:0 pins wider than the minimum pulse width of 50ns will trigger edge detection. If enabled, the edge detection will trigger even if the corresponding GPIO are configured as outputs.

Functions

sub_edge_config
sub_edge_read

3.7.7.1 sub_edge_config

Synopsis

```
int    sub_edge_config( sub_handle hndl, int set, int* get )
```

Configure EDGE module.

Parameters

- set - **set** will be used only if **get** == 0. EDGE lines EDG7..EDG0 are configured by two bits Ex1,Ex0 in **set** where x is 7..0. **set** bits have following interpretation:

MSB

31..16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
-	EDG7		EDG6		EDG5		EDG4		EDG3		EDG2		EDG1		EDG0	
-	E71	E70	E61	E60	E51	E50	E41	E40	E31	E30	E21	E20	E11	E10	E01	E00

Ex1	Ex0	Edge Detection
0	0	No
0	1	Any change
1	0	Falling edge
1	1	Rising edge

- *get - Pointer to store current edge configuration read from SUB-20. If get is not zero function will read edge configuration otherwise it will set edge configuration

Return value

On success function returns 0. Otherwise error code.

Example

```
rc = sub_edge_config( hndl, 0, &edge_config );   /* Read Edge configuration */

/* EDG0 = 01B - Any change */
/* EDG1 = 01B - Any change */
/* EDG2 - 11B - Risinge edge */
/* EDG7..EDG3 not detected */
rc = sub_edge_config( hndl, 0x0035, 0 );   /* set = 0011.0101B */
```

See also

Error Codes

3.7.7.2 sub_edge_read

Synopsis

```
int    sub_edge_read( sub_handle hndl, int* get, int* edge )
```

Read GPIO input status and EDGE status. For GPIO Function reads logic value "1"-high, "0"-low directly from GPIO pin. For EDGE function reads "1" - edge detected, "0" - edge not detected.

Parameters

- *get - Pointer to store received GPIO input status. Bits 0..31 of ***get** correspond to GPIO0..GPIO31 input statuses.
- *edge - Pointer to store EDGE status. Bits 0..7 of ***edge** correspond to EDG0..EDG7.

Return value

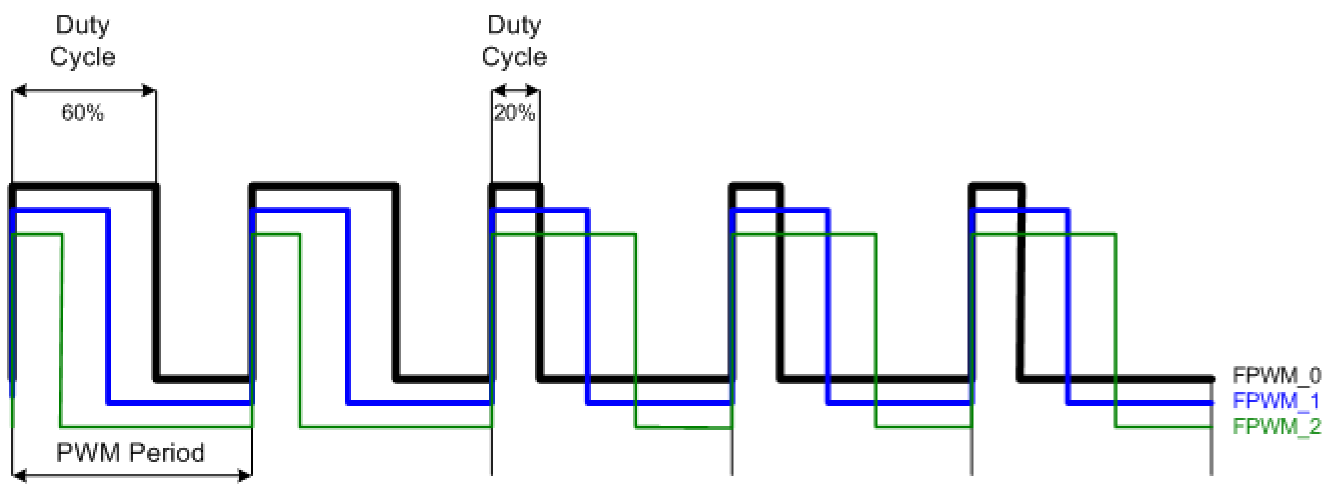
On success function returns 0. Otherwise error code.

See also

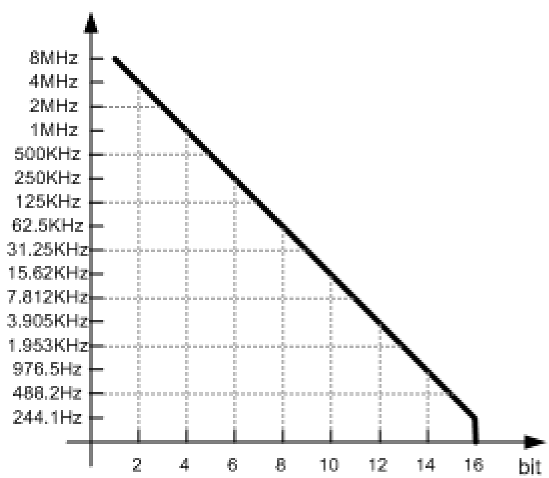
Error Codes

3.7.8 Fast PWM

Fast PWM outputs are available on SPI header pins 4,6,8 (see SPI Header). Outputs are referenced as FPWM_0, FPWM_1 and FPWM_2. All three outputs share the same PWM generation module in SUB-20 and have the same PWM frequency but separate duty cycle configuration.



Fast PWM frequency range is 8Mhz .. 0.238Hz. Duty cycle precision is 2..16 bits depending on the PWM frequency.



Functions

sub_fpwm_config
sub_fpwm_set

3.7.8.1 sub_fpwm_config

Synopsis

```
int sub_fpwm_config( sub_handle hndl, double freq_hz, int flags )
```

Configure fast PWM module.

Parameters

- freq_hz - Desired fast PWM frequency in Hz. Frequency can be in a range 8MHz .. 0.238Hz
- flags - Set of configuration flags listed below

FPWM_ENABLE	General fast PWM enable. If this flag is not set fast PWM module will be disabled and FPWM outputs will go to HiZ.
FPWM_EN0	Enable FPWM_0 output. Otherwise FPWM_0 will stay low.
FPWM_EN1	Enable FPWM_1 output. Otherwise FPWM_1 will stay low.
FPWM_EN2	Enable FPWM_2 output. Otherwise FPWM_2 will stay low.

Return value

On success function returns 0. Otherwise error code.

Example

```
/*
 Enable fast PWM module with FPWM_0 and FPWM_2 outputs.
 PWM frequency 10.6Hz
*/
sub_fpwm_config( hndl, 10.6, FPWM_ENABLE|FPWM_EN0FPWM_EN2 );
```

See also

sub_fpwm_set
Error Codes

3.7.8.2 sub_fpwm_set

Synopsis

```
int sub_fpwm_set( sub_handle hndl, int pwm_n, double duty )
```

Configure specific fast PWM output.

Parameters

- pwm_n - number of FPWM output to configure. Can be 0,1,2
- duty - desired duty cycle % in range 0..100. Can be not integer.

Return value

On success function returns 0. Otherwise error code.

Example

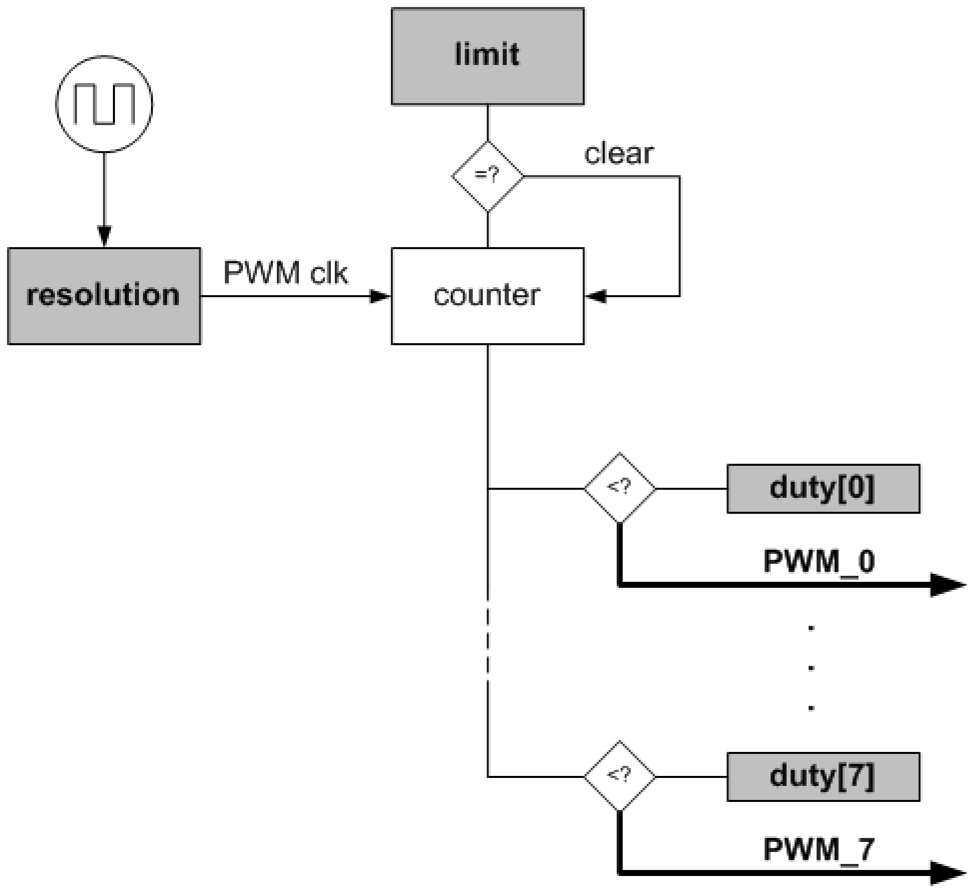
```
sub_fpwm_set( hndl, 0, 20 );
sub_fpwm_set( hndl, 1, 30.5 );
sub_fpwm_set( hndl, 2, 12.25 );
```

See also

sub_fpwm_config
Error Codes

3.7.9 PWM

PWM outputs are available on GPIO24..GPIO31 pins (see GPIO Header). Outputs are referenced as PWM_0 .. PWM_7.
PWM generation module shown below has configurable source clock, counter limit and separate comparator for each PWM output.



PWM module resolution and limit are configured with sub_pwm_config. Duty cycle of each PWM output can be set with sub_pwm_set.
PWM module output signal controls output driver of the corresponding GPIO (see GPIO Functions). **To get high/low transition of the GPIO pin it should be configured to output state with** sub_gpio_config. Otherwise PWM module will enable/disable internal pull-up.

Functions

sub_fpwm_config
sub_pwm_set

3.7.9.1 sub_pwm_config

Synopsis

```
int sub_pwm_config( sub_handle hndl, int res_us, int limit )
```

Configure PWM module.

Parameters

- res_us - PWM module clock resolution in µs. Resolution range is 20µs - 16384µs.
- limit - PWM module counter limit in range 0-255. If **limit** is 0 PWM module will be turned off.

PWM frequency

Resolution and limit define PWM module frequency

```
Fpwm = 1000000/(res_us * limit) Hz
```

PWM frequency range is

```
Fpwm_min = 1000000/(20 * 2) = 25000 Hz = 25KHz
Fpwm_max = 1000000/(16384 * 255) = 0.238 Hz
```

Return value

On success function returns 0. Otherwise error code.

See also

sub_pwm_set
Error Codes

Compatibility

FW version	0.1.8 or grater
Library version	0.1.12.10 or grater

3.7.9.2 sub_pwm_set

Synopsis

```
int sub_pwm_set( sub_handle hndl, int pwm_n, int duty )
```

Configure specific PWM output.

Parameters

- pwm_n - number of PWM output to configure. Can be 0..7.
- duty - Duty cycle in range 0..255

Duty cycle

- If duty cycle is 0, PWM output will be constantly low
- If duty cycle is grater or equal to the limit set by sub_pwm_config, PWM output will be constantly high

Effective duty cycle in percent can be calculated as

```
DUTY% = duty/limit * 100%
```

Duty cycle resolution in bits depends on limit. It can be in range of 1 bit for limit=2 up to 8 bit for limit=255

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Set PWM resolution=10ms, limit=100, frequency=1Hz */
sub_pwm_config( hndl, 10000, 100 );

/* Set PWM_0 pin (GPIO24) to output state */
sub_gpio_config( hndl, 0x01000000, &config, 0x01000000 );

/* Output 50% duty cycle on PWM_0 pin */
```

```
sub_pwm_set( hndl, 0, 50 );
```

See also

sub_pwm_config
Error Codes

Compatibility

FW version	0.1.8 or grater
Library version	0.1.12.10 or grater

3.8 Analog to Digital Converter - ADC

SUB-20 has 8 single ended and a 16 differential analog input combinations with x1 x10 or x200 input gain amplifier. Analog inputs are referenced as ADC0..ADC7 see GPIO Header. The SUB-20 ADC module features a 10-bit successive approximation ADC. The ADC contains a Sample and Hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion.

Functions

sub_adc_config
sub_adc_single
sub_adc_read

3.8.1 sub_adc_config

Synopsis

```
int sub_adc_config( sub_handle hndl, int flags )
```

Configure SUB-20 ADC module.

Parameters

- flags - set of below flags

ADC_ENABLE	Enable ADC module. If this flag is not set ADC module will be disabled
ADC_REF_VCC	Use VCC as ADC reference
ADC_REF_2_56	Use internal 2.56V reference

Return value

On success function returns 0. Otherwise error code.

3.8.2 sub_adc_single

See sub_adc_read.

3.8.3 sub_adc_read

Synopsis

```
int sub_adc_single( sub_handle hndl, int* data, int mux );
int sub_adc_read( sub_handle hndl, int* data, int* mux, int reads );
```

Read single or multiple ADC conversion result(s)

Parameters

- data - buffer to store conversion result(s)
- mux - ADC input channel multiplexer control code(s). See table below.
- reads - number of results to read

ADC mux code	Single Ended	Differential Positive	Differential Negative	Gain
ADC_S0	ADC0	N.A.		
ADC_S1	ADC1			
ADC_S2	ADC2			
ADC_S3	ADC3			
ADC_S4	ADC4			
ADC_S5	ADC5			
ADC_S6	ADC6			
ADC_S7	ADC7			
ADC_D10_10X	N.A.	ADC1	ADC0	10
ADC_D10_200X		ADC1	ADC0	200
ADC_D32_10X		ADC3	ADC2	10
ADC_D32_200X		ADC3	ADC2	200
ADC_D01		ADC0	ADC1	N.A.
ADC_D21		ADC2	ADC1	
ADC_D31		ADC3	ADC1	
ADC_D41		ADC4	ADC1	
ADC_D51		ADC5	ADC1	
ADC_D61		ADC6	ADC1	
ADC_D71		ADC7	ADC1	
ADC_D02		ADC0	ADC2	
ADC_D12		ADC2	ADC2	
ADC_D32		ADC3	ADC2	
ADC_D42		ADC4	ADC2	
ADC_D52		ADC5	ADC2	
ADC_1_1V	Internal 1.1V	N.A.		
ADC_GND	Analog GND			

ADC Result

For single ended conversion:

$$ADC = \frac{V_{IN} \cdot 1023}{V_{REF}}$$

For differential:

$$ADC = \frac{(V_{POS} - V_{NEG}) \cdot GAIN \cdot 512}{V_{REF}}$$

ADC is a result of conversion. Note that the result of differential conversion is signed and can be negative.

Return value

On success function returns 0. Otherwise error code.

Example

```
int adc, adc_buf[3], adc_mux[3];
```

```
sub_adc_single( fd, &adc, ADC_S0 );   /* Read ADC on ADC0 input */

adc_mux[0] = ADC_S0;
adc_mux[1] = ADC_S2;
adc_mux[2] = ADC_S3;
sub_adc_read( fd, adc_buf, adc_mux, 3 ); /* Read ADC on ADC0,2,3 inputs */
```

3.9 LCD Functions

Functions

- sub_lcd_write

3.9.1 sub_lcd_write

Synopsis

```
int sub_lcd_write( sub_handle hndl, char* str );
```

This function will work only on SUB-20 configurations with LCD.
Characters from ***str** are written to the LCD beginning from the current LCD position. Special characters and sequences listed below are used to format LCD output and control current position.

ANSI C notation	Hex value	Description	Example
\f	0x0C	Clear LCD and go to first position	"\fHello"
\r	0x0D	Go to first position	"\r0123"
\n	0x0A	Go to next string	"\rHello\nWorld"
\eXY	0x1B X Y	Go to position X,Y	"\e00Hello\e01World"

Every string written to LCD will be space padded till the end of the string.

Parameters

- *str - LCD string

Return value

On success function returns 0. If LCD is not supported function will return "Feature not supported".

Example

Code	LCD
`sub_lcd_write( hndl, "\fHello\nWorld" );`	Hello World
`sub_lcd_write( hndl, "\f\e20Hello\e21World" );`	Hello World
`sub_lcd_write( hndl, "\r1\n2" );`	1 2
`sub_lcd_write( hndl, "\fT:\nR:" );` `sub_lcd_write( hndl, "\e20abc\e21def" );`	T:abc R:def

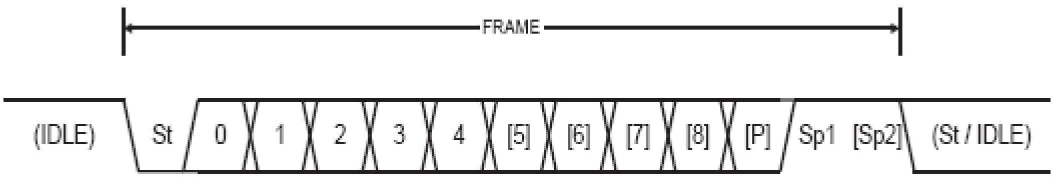
Compatibility

- Required firmware version > 0.0.4

- Required DLL version > 0.1.12.2

3.10 RS232 RS485 Functions

RS232 RS485 Frame format



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.

Functions

sub_rs_set_config
sub_rs_get_config
sub_rs_timing
sub_rx_xfer

3.10.1 sub_rs_set_config

Synopsis

```
int sub_rs_set_config( sub_handle hndl, int config, int baud)
```

Configure SUB-20 UART (Universal Asynchronous Receiver Transmitter).

Parameters

- config - UART configuration. Config should be assembled as a combination of the following flags

RS_RX_ENABLE	Enable UART receiver
RS_TX_ENABLE	Enable UART transmitter
RS_CHAR_5 RS_CHAR_6 RS_CHAR_7 RS_CHAR_8 RS_CHAR_9	Data Bits
RS_PARITY_NONE RS_PARITY_EVEN RS_PARITY_ODD	Parity
RS_STOP_1 RS_STOP_2	Stop Bits

To disable UART provide configuration without RS_RX_ENABLE and RS_TX_ENABLE.

- baud - Desired baudrate. Maximum baudrate is 2Mbps. The actual baudrate may slightly differ from the desired as actual baudrate is an integer quotient from dividing the 16MHz reference clock.

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Set 9660 bps, 8 data bits, no parity, 1 stop bit */
sub_rs_set_config( hndl,
RS_RX_ENABLE|RS_TX_ENABLE|RS_CHAR_8|RS_PARITY_NONE|RS_STOP_1, 9600 );
```

3.10.2 sub_rs_get_config

Synopsis

```
int sub_rs_get_config( sub_handle hndl, int* config, int* baud)
```

Read current SUB-20 UART configuration.

Parameters

- *config - will be filled with UART configuration see sub_rs_set_config for details
- *baud - will be filled with actual UART baudrate

Return value

On success function returns 0. Otherwise error code.

3.10.3 sub_rs_timing

Synopsis

```
int sub_rs_timing( sub_handle hndl, int flags, int tx_space_us,
                                       int rx_msg_us, int rx_byte_us )
```

Configure UART transfer timing and order of transmit and receive operations. Actual transfer is initiated by sub_rx_xfer function.

Parameters

- flags - one or none of the below flags:

RS_RX_BEFORE_TX	Receive message and after that transmit message
RS_RX_AFTER_TX	Transmit message and after that receive message
none	Receive and transmit simultaneously

- tx_space_us - delay in µs between subsequent byte transmit
- rx_msg_us - message reception timeout in µs
- rx_byte_us - byte to byte reception timeout in µs

Delay and timeouts precision is ± 64 µs. They should not exceed 4.000.000µs = 4s.

Following table explains relation between **rx_msg_us** and **rx_byte_us** parameters

rx_msg_us=0	rx_byte_us=0	Message reception will last no more then 4s
rx_msg_us=0	rx_byte_us>0	First byte reception will last no more then 4s. Every next byte should be received in rx_byte_us time
rx_msg_us>0	rx_byte_us=0	Message reception will last no more then rx_msg_us time

rx_msg_us>0	rx_byte_us>0	First byte reception will last no more then rx_msg_us time. Every next byte should be received in rx_byte_us time

Return value

On success function returns 0. Otherwise error code.

Example

```
/* Request message transmit and after that receipt */
/* No space between transmitted bytes */
/* Message should be received in 1s */
sub_rs_timing( hndl, RS_RX_AFTER_TX, 0, 1000000, 0 );
```

3.10.4 sub_rs_xfer

Synopsis

```
int sub_rs_xfer( sub_handle hndl, char* tx_buf, int tx_sz,
                                  char* rx_buf, int rx_sz )
```

Transmit and/or receive message(s) via UART configured with sub_rs_set_config and in accordance with transfer timing set with sub_rs_timing.

Parameters

- tx_buf - buffer with data to be transmitted
- tx_sz - number of bytes to transmit (can be 0 if transmit not required)
- rx_buf - buffer to store received data
- rx_sz - maximal number of bytes to receive (can be 0 if reception is not required)

rx_sz and **tx_sz** should not be grater then 62 bytes.
Function will terminate after following conditions fulfilled: **tx_sz** bytes transmitted and either **rx_sz** bytes received or one of the timeouts occurs (see sub_rs_timing)

For 9bit transfer data in **tx_buf** and **rx_buf** has following format

byte 0	byte 1	...	byte n	byte n+1
bit ..8 LSB	bits 7..0	...	bit ..8 LSB	bits 7..0

Even bytes (0,2,4,....) contain 8'th bit in LSB. Odd bytes (1,3,5,...) contain bits 7..0. Data is shifted out beginning from bit 0 (see RS232 RS485 Frame format). Parameters **tx_sz** and **rx_sz** should correspond to total number of bytes in **tx_buf** and **rx_buf**.

Return value

On success function returns non negative value that denotes number of received bytes. It can be less then **rx_sz** if timeout occurs.
Otherwise -1 will be returned and sub_errno will contain error code.

Example

```
/* Set timing */
sub_rs_timing( hndl, RS_RX_AFTER_TX, 0, 1000000, 200000 );

/*
 Transmit 3 bytes and try to receive up to 5 bytes
 in 1s with 200ms byte to byte timeout
*/
```

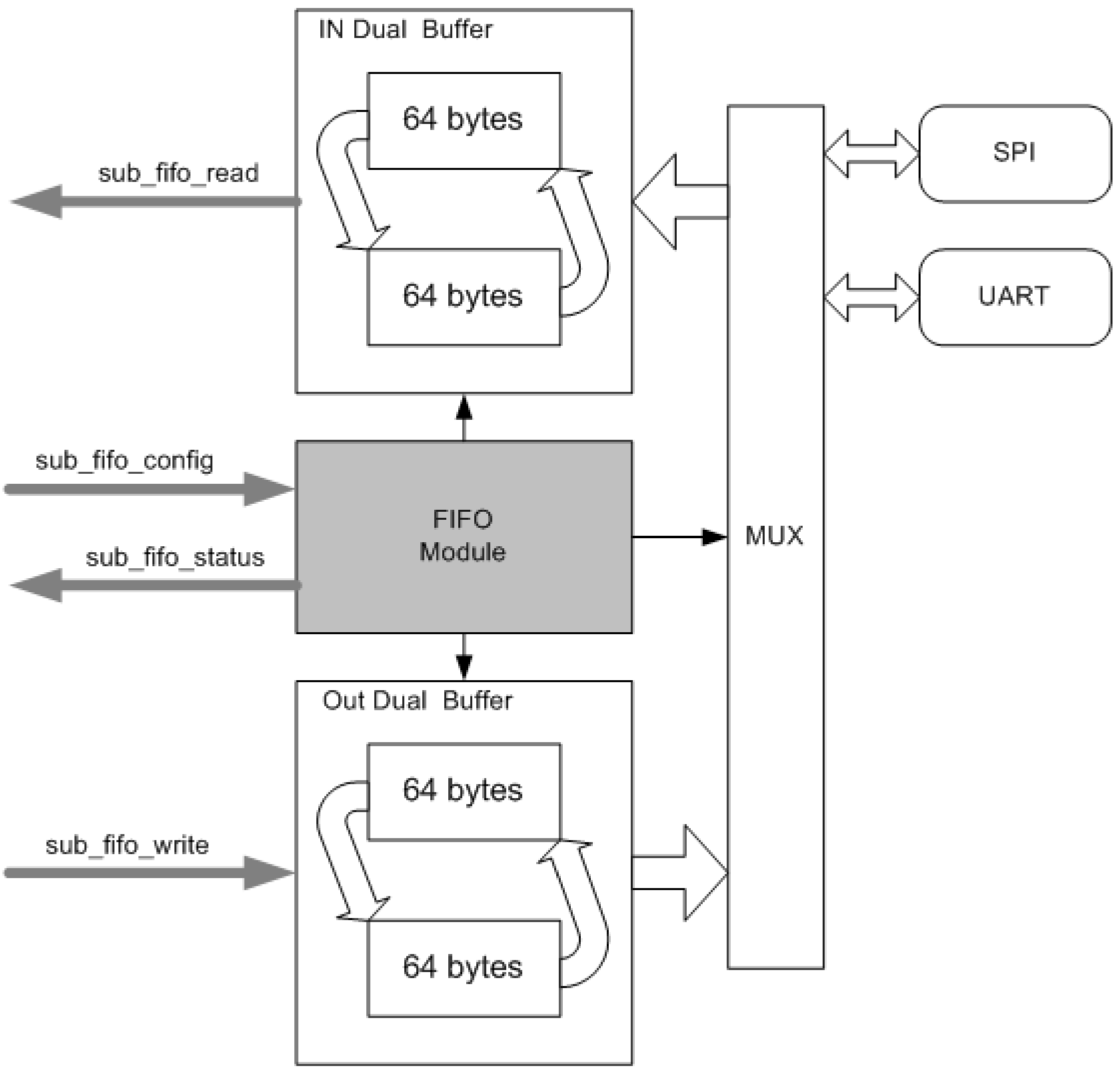
```
sub_rs_xfer( hndl, "at\r", 3, rx_buf, 5 );
```

Compatibility

FW version 9-bit transfer support - 0.2.2 or grater

3.11 FIFO, Streaming, Slave Modes

FIFO is used for implementation of streaming data transfer and slave modes functionality such as SPI slave, I2C slave. Below is simplified block diagram of the SUB-20 FIFO module.



3.11.1 FIFO Functions

sub_fifo_config
sub_fifo_read
sub_fifo_write

3.11.1.1 sub_fifo_config

Synopsis

```
int sub_fifo_config( sub_handle hndl, int config )
```

Configure FIFO.

Parameters

- config - set of FIFO configuration flags listed below

FIFO_SELECT_SPI	Connect SPI module to FIFO
FIFO_SELECT_I2C	Connect I2C slave module to FIFO
FIFO_SELECT_UART	Connect UART to FIFO
FIFO_CLEAR	Clear IN and OUT FIFO buffers
FIFO_READ_FULL	Read only full 64 byte FIFO buffers

Return value

On success function returns 0. Otherwise error code.

Compatibility

FW version	FIFO_CLEAR - 0.1.3 or grater FIFO_SELECT_UART - 0.2.0 or grater
Library version	0.1.12.10 or grater

3.11.1.2 sub_fifo_write

Synopsis

```
int sub_fifo_write( sub_handle hndl, char* buf, int sz, int to_ms )
```

Function attempts to transfer **sz** bytes from buffer into OUT FIFO in no more then **to_ms** time.

Parameters

- buf - source buffer
- sz - buffer size
- to_ms - timeout in milliseconds

Return Value

On success a non negative number of actually written bytes is returned. It can be less then or equal to **sz**. In case of error negative error code (defined in errno.h) will be returned and if applicable sub_errno will be set to the corresponding value. Possible error codes are

`-ENOENT (-2)`	USB failure
`-EIO (-5)`	USB failure
`-ENOMEM (-12)`	Memory failure
`-EINVAL (-22)`	Invalid parameter
`-EFBIG (-27)`	Buffer overflow
`-ETIMEDOUT (-116)`	Timeout

3.11.1.3 sub_fifo_read

Synopsis

```
int sub_fifo_read( sub_handle hndl, char* buf, int sz, int to_ms )
```

Function attempts to read **sz** bytes from IN FIFO into buffer in no more then **to_ms** time.

Parameters

- buf - destination buffer

- sz - buffer size. Considering internal FIFO structure it is highly recommended to use only 64 divisible **sz** values and corresponding buffer size. It will guarantee better performance and no buffer overflow error (-EFBIG).
- to_ms - timeout in milliseconds

Return Value

On success a non negative number of actually read bytes is returned. It can be less then or equal to **sz**. In case of error negative error code (defined in errno.h) will be returned and if applicable sub_errno will be set to the corresponding value. Possible error codes are

-ENOENT (-2)	USB failure
-EIO (-5)	USB failure
-ENOMEM (-12)	Memory failure
-EINVAL (-22)	Invalid parameter
-EFBIG (-27)	Buffer overflow. To prevent this error use 64 divisible **sz**
-ETIMEDOUT (-116)	Timeout

Known Issue

If IN FIFO contains exactly 64 or 128 bytes, attempt to read more then 64 or 128 bytes correspondingly with sub_fifo_read will cause to timeout (-116). As a workaround always call sub_fifo_read with sz=64.

3.11.2 SPI Slave

Configuration

To use SUB-20 in SPI slave mode following should be done:

- Configure SPI module with sub_spi_config. Make sure to set SPI_SLAVE flag.
- Connect SPI module to FIFO with sub_fifo_config. FIFO_SELECT_SPI must be set.
- If required, write data to FIFO with sub_fifo_write. This data will be read by external SPI master.
- If required, read data from FIFO with sub_fifo_read. This will be data written by external SPI master.

Example

Below is an example to exchange 12 bytes with SPI master:

```
sub_spi_config( hndl, SPI_ENABLE|SPI_SLAVE|SPI_CPOL_RISE|SPI_SMPL_SETUP, 0 );
sub_fifo_config( hndl, FIFO_SELECT_SPI );
sub_fifo_write( hndl, "Hello Master", 12, 100 );
read_sz=0;
while( read_sz < 12 )
{
   rc = sub_fifo_read( hndl, in_buff, 64, 10000 ); /* wait 10 sec */
   if( rc < 0 )
      return rc;   /* error */
   read_sz += rc;
}
```

Slave Select

SUB-20 SPI Slave module use SS0 pin as slave select. SS0 must be pulled low by external SPI master to enable SUB-20 SPI module functionality. When SS0 is held low, the SPI is activated, and MISO becomes an output. All other pins are inputs. When SS is driven high, all pins are inputs, and the SPI is passive.

3.11.3 I2C Slave

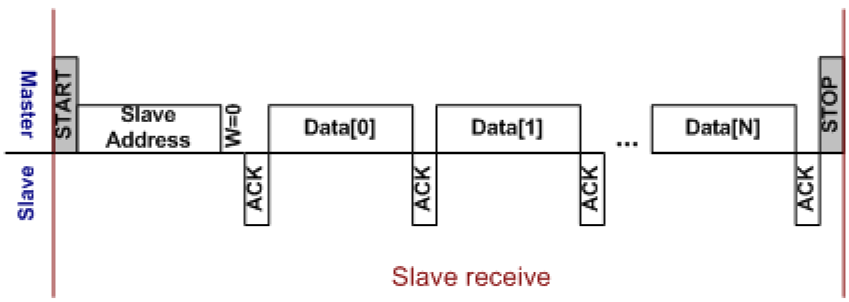
Configuration

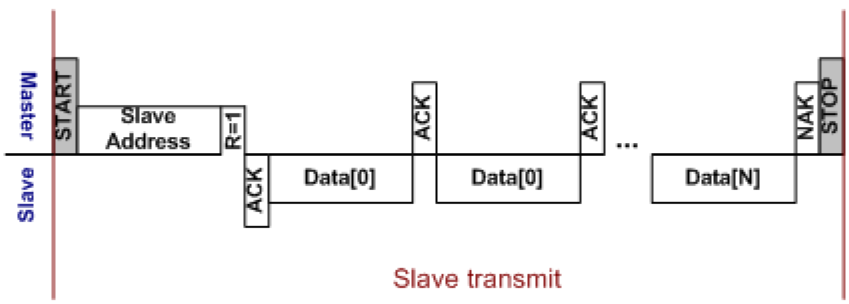
To use SUB-20 in I2C slave mode following should be done:

- Configure I2C Slave module with sub_i2c_config. Set desired I2C slave address and optional flag(s).
- Connect I2C Slave module to FIFO with sub_fifo_config. Set FIFO_SELECT_I2C flag.
- If required write data to FIFO with sub_fifo_write. This data will be read by external I2C master.
- If required read data from FIFO with sub_fifo_read. This will be data written by external I2C master.

Transactions

SUB-20 I2C slave module acts as zero address I2C slave device. Below are diagrams of receive and transmit transactions.





3.12 Error Codes

API functions may return error code and/or set global variable sub_errno to provide information about completion of requested operation. The string describing error code can be received with sub_strerror function call.

3.12.1 sub_errno

Global variable **sub_errno** contains status of last SUB-20 API call. Variable **sub_errno** defined in *libsub.h.*
Following error codes are available:

0	OK
1	SUB Device not found
2	Can't open SUB device
3	Can't set configuration

4	Can't claim interface
5	Failed to setup async transaction
6	Failed to submit async transaction
7	Bulk write failed
8	Bulk read failed
9	Bulk read incomplete
10	Out buffer overflow
11	I2C error
12	Wrong tag code in response
13	Wrong tag size in response
14	Wrong parameters
15	SPI Disabled
16	Feature not supported
17	Timeout
18	Init Failed
19	Bad Handle
20	InfraRed Failure

3.12.2 sub_strerror

Synopsis

```
char* sub_strerror( int errnum )
```

Return string describing error number. This function is similar to strerror().

Parameters

- errnum - error number

Return Value

Function returns pointer to string with error description. If error number is unknown function will return pointer to string "Unrecognized error <error_number>".

4 Electrical Characteristics

4.1 Absolute Maximum Ratings

Operating Temperature	-20°C to +65°C
Voltage on any Pin with respect to Ground	-0.5V to VCC+0.5V
Maximum Operating Voltage	+6V
DC Current per I/O Pin	40mA
DC Current V_{CC} and GND Pins	200mA

4.2 DC Characteristics

Symbol	Parameter	Condition	Min	Typ	Max	Units
V_{IL}	Input Low Voltage	V_{CC} = 3.3V - 5.5V	0.5		0.2V_{CC}	V
V_{IH}	Input High Voltage	V_{CC} = 3.3V - 5.5V	0.6V_{CC}		V_{CC} + 0.5	V
V_{OL}	Output Low Voltage	I_{OL} = 10mA, V_{CC} = 5V I_{OL} = 5mA, V_{CC} = 3.3V		0.3 0.2	0.7 0.5	V
V_{OH}	Output High Voltage	I_{OH} = -20mA, V_{CC} = 5V I_{OH} = -10mA, V_{CC} = 3.3V	4.2 2.3	4.5 2.6		V
I_{IL}	Input Leakage Current I/O Pin	V_{CC} = 5.5V, pin low			1	µA
I_{IH}	Input Leakage Current I/O Pin	V_{CC} = 5.5V, pin high			1	µA
I_{CC}	Power Consumption Current	V_{CC} = 5V		19	30	mA

Although each I/O port can sink or source more than the test conditions under steady state conditions (non-transient), the following must be observed:
1. The sum of all I_{OL}, for GPIO4-GPIO7 should not exceed 100 mA.
2. The sum of all I_{OL}, for GPIO0-GPIO3, GPIO8-GPIO15 should not exceed 100 mA.
3. The sum of all I_{OL}, for GPIO24-GPIO31 should not exceed 100 mA.
4. The sum of all I_{OL}, for GPIO16-GPIO23 should not exceed 100 mA.
5. If I_{OL} exceeds the test condition, V_{OL} may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test condition.
6. The sum of all I_{OH}, for GPIO4-GPIO7 should not exceed 100 mA.
7. The sum of all I_{OH}, for GPIO0-GPIO3, GPIO8-GPIO15 should not exceed 100 mA.
8. The sum of all I_{OH}, for GPIO24-GPIO31 should not exceed 100 mA.
9. The sum of all I_{OH}, for GPIO16-GPIO23 should not exceed 100 mA.

4.3 AC Characteristics

Symbol	Parameter	Condition	Min	Typ	Max	Units
SPI						
F_{SPI}	SPI Master frequency		0.125		8	MHz
F_{SPI_SLAVE}	SPI Slave frequency				2	MHz
MDIO						
F_{MDC}	MDC frequency			1		MHz
High Speed (HS) I2C Master						
F_{SCLH}	HS SCL frequency				3.125	MHz
t_{HLOW}	HS SCL LOW period			187.5		ns
t_{HHIGH}	HS SCL HIGH period			125		ns

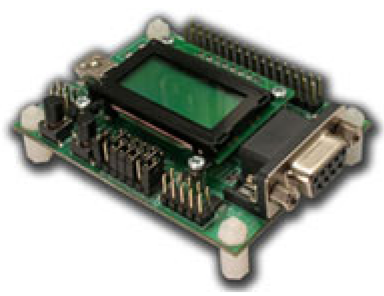
5 Ordering Information

Ordering Code	I2C	SPI Master	SPI Slave	GPIO	LCD	RS232	RS485	SW1/2	IR	Notes
SUB-20-B **Basic**	V	V	V	V						
SUB-20-EB **Basic Enclosure**	V	V	V							Heavy duty enclosure with DB9 Connector
SUB-20-Dxxx	V	V	V	V	V	x				Heavy duty enclosure with DB9 and DB26
SUB-20-V **Visual**	V	V	V	V	V					
SUB-20-R25 **Serial2**	V	V	V	V	V	V		V		
SUB-20-R45 **Serial4**	V	V	V	V	V		V	V		
SUB-20-I5 **Infra5**	V	V	V	V	V			V	V	
SUB-20-Lxxx	V	V		V	x	x	x	x	x	SPI Level Converters
SUB-20-Cxxx	Custom Configuration									

For SUB-20-Lxxx, 'xxx' - can be any of the above configuration with addition of SPI Level Converters and without SPI Slave option. For example SUB-20-LV is SUB-20-V board with SPI Level Converters.

SUB-20-B
SUB-20-LB

SUB-20-V **SUB-20-R25**
SUB-20-LV **SUB-20-LR25**



SUB-20-R45
SUB-20-LR45

SUB-20-EB
SUB-20-LEB

SUB-20-DV **SUB-20-DR25**
SUB-20-LDV **SUB-20-LDR25**