# Table of Contents

## Part I  Document Revision History

## Part II  SUB-20 Layout

1. GPIO Header ................................................................. 7
2. SPI Header ............................................................... 8
3. DB9 Connector ........................................................... 8
4. RS485 Connector ...................................................... 9
5. Jumpers, SW1, SW2 .................................................. 9
6. Power Distribution .................................................... 10

## Part III  SUB-20 API

1. Device Initialization .................................................. 14
   - sub_find_devices .................................................. 14
   - sub_open ............................................................... 14
   - sub_get_serial_number .......................................... 15
   - sub_get_product_id .............................................. 15
   - sub_get_version ................................................... 16
   - sub_reset ............................................................. 16
2. Internal EEPROM Functions ...................................... 16
   - sub_eep_read ......................................................... 16
   - sub_eep_write ........................................................ 17
3. I2C functions .......................................................... 17
   - sub_i2c_freq .......................................................... 18
   - sub_i2c_config ...................................................... 18
   - sub_i2c_start ......................................................... 19
   - sub_i2c_stop .......................................................... 19
   - sub_i2c_scan .......................................................... 19
   - sub_i2c_read .......................................................... 20
   - sub_i2c_write .......................................................... 20
   - sub_i2c_transfer .................................................... 20
   - sub_i2c_hs_rw ....................................................... 21
   - I2C Status ............................................................ 22
4. Bit-bang I2C Masters ................................................. 24
   - sub_bb_i2c_config .................................................. 24
   - sub_bb_i2c_scan .................................................... 25
   - sub_bb_i2c_read ..................................................... 25
   - sub_bb_i2c_write ................................................... 26
5. SPI Functions .......................................................... 27
   - sub_spi_config ....................................................... 28
   - sub_spi_transfer ..................................................... 30
   - sub_spi_transfer_ess ............................................. 31
   - sub_sdio_transfer .................................................. 31

© 2018 xdimax.com
Part IV Electrical Characteristics

1 Absolute Maximum Ratings ............................................................ 60
2 DC Characteristics ................................................................. 60
3 AC Characteristics ................................................................. 60
Part V  Ordering Information
1 Document Revision History

- 2.18
  1. sub_i2c_transfer
  2. GPIO Watchdog

- 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**

![SUB-20 Layout Diagram](image)
## 2.1 GPIO Header

<table>
<thead>
<tr>
<th>Header Pin</th>
<th>GPIO</th>
<th>Alternative Function</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GPIO8</td>
<td>I2C_SCL</td>
<td>Pulled up 1.5KOhm DB9.8</td>
</tr>
<tr>
<td>2</td>
<td>GPIO9</td>
<td>I2C_SDA</td>
<td>Pulled up 1.5KOhm DB9.6</td>
</tr>
<tr>
<td>3</td>
<td>GPIO10</td>
<td>RS232_RXD_TTL</td>
<td>For Configurations with RS232 or RS485</td>
</tr>
<tr>
<td>4</td>
<td>GPIO11</td>
<td>RS232_TXD_TTL</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GPIO12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>GPIO13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>GPIO14</td>
<td>MDC</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>GPIO15</td>
<td>MDIO</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>GPIO24</td>
<td>PWM_0, SCLx0, MISOx0¹</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>GPIO25</td>
<td>PWM_1, SDAx0, MOSIx0¹</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>GPIO26</td>
<td>PWM_2, SCLx1, SCKx0¹</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>GPIO27</td>
<td>PWM_3, SDAx1, SSx0¹</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>GPIO28</td>
<td>PWM_4, SCLx2, MISOx1¹, UARTx0_RX¹</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>GPIO29</td>
<td>PWM_5, SDAx2, MOSIx1¹, UARTx0_TX¹</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>GPIO30</td>
<td>PWM_6, SCLx3, SCKx1¹, UARTx1_RX¹</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>GPIO31</td>
<td>PWM_7, SDAx3, SSx1¹, UARTx1_TX¹</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>GPIO0</td>
<td>LCD_D0</td>
<td>For Configurations with LCD</td>
</tr>
<tr>
<td>18</td>
<td>GPIO1</td>
<td>LCD_D1</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>GPIO2</td>
<td>LCD_D2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>GPIO3</td>
<td>LCD_D3</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>GPIO4</td>
<td>IR_TX</td>
<td>For Configurations with IR</td>
</tr>
<tr>
<td>22</td>
<td>GPIO5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>GPIO6</td>
<td>LCD_En</td>
<td>For Configurations with LCD</td>
</tr>
<tr>
<td>24</td>
<td>GPIO7</td>
<td>LCD_RS</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>GPIO23</td>
<td>ADC7</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>GPIO22</td>
<td>ADC6</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>GPIO21</td>
<td>ADC5</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>GPIO20</td>
<td>ADC4</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>GPIO19</td>
<td>ADC3</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>GPIO18</td>
<td>ADC2</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>GPIO17</td>
<td>ADC1</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>GPIO16</td>
<td>ADC0</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>GPIO</td>
<td>VCC</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td>GND</td>
<td></td>
</tr>
</tbody>
</table>

¹ - Under development
2.2 SPI Header

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

Level Converter* - Adjustable operating voltage in configurations with Level Converters (see Jumper and Ordering Information).

GPIO Voltage** - Operates at the same voltage as GPIO according to SUB-20 configuration (see Ordering Information).

2.3 DB9 Connector

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SPI SS0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I2C_EXT_PWR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RS232 PC_RX</td>
<td>Serial2</td>
</tr>
<tr>
<td>3</td>
<td>SPI_EXT_PWR</td>
<td>Lxxx</td>
</tr>
<tr>
<td></td>
<td>RS232 PC_TX</td>
<td>Serial2</td>
</tr>
<tr>
<td>Pin</td>
<td>Function</td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SPI SCK</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>GND</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>I2C SDA</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SPI MISO</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>I2C SCL</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>SPI MOSI</td>
<td></td>
</tr>
</tbody>
</table>

### 2.4 RS485 Connector

RS485 Connector is installed only in Serial4 configuration.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>inverting or '-' pin</td>
</tr>
<tr>
<td>B</td>
<td>non-inverting or '+' pin</td>
</tr>
</tbody>
</table>

### 2.5 Jumpers, SW1, SW2
### Table: SUB-20 Configuration

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

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_i2c_transfer
sub_i2c_hs_rw

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_gpio_wdt_set
sub_gpio_wdt_get

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
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**

```c
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:

![Slave Address 0x50 Diagram]

**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**

- `sub_i2c_freq`
- `sub_i2c_config`
- `sub_i2c_start`
- `sub_i2c_stop`
- `sub_i2c_read`
- `sub_i2c_write`
- `sub_i2c_transfer`
- `sub_i2c_hs_rw`

### 3.3.1  sub_i2c_freq

#### Synopsis

```c
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

```c
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

```c
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:

![I2C Transaction Diagram](image)

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

```c
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 \texttt{ma\_sz} is zero "Memory Address Write" stage will be skipped. Function has no limitation of the write data size - \texttt{sz} parameter. However internal organization of the I2C slave device being written should be considered especially \texttt{Acknowledge Polling} and \texttt{EEPROM Page Write}.

**Parameters**

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

**Return value**

On success function returns 0. Otherwise error code.

**See also**

Error Codes, I2C Status

### 3.3.8 \texttt{sub\_i2c\_transfer}

**Synopsis**

\begin{verbatim}
int sub_i2c_transfer( sub_handle hndl, int sa,
                        char* wr_buf, int wr_buf_sz, char* rd_buf, int rd_buf_sz )
\end{verbatim}

Perform I2C master write transaction followed by master read transaction. Transactions will have following format:

Function has no limitation of the read data size - \texttt{rd\_buf\_sz} parameter. However internal organization of the I2C slave device being read should be considered.

**Parameters**

- \texttt{sa} - \texttt{Slave Address}
```
· wr_buf - Buffer for data to be written
· wr_buf_sz - Write data size bytes
· rd_buf - Buffer for data to be read
· rd_buf_sz - Read data size bytes

Return value
On success function returns 0. Otherwise error code.

See also
Error Codes, I2C Status

Compatibility
FW version 0.4.4 or grater
Library version 0.1.12.32 or grater

3.3.9 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;
}
```c
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**

```c
/*
 * 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 );
```

### 3.3.10 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

```c
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
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.

<table>
<thead>
<tr>
<th>Header Pin</th>
<th>GPIO</th>
<th>Alternative I2C Master Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>GPIO24</td>
<td>SCLx0</td>
</tr>
<tr>
<td>10</td>
<td>GPIO25</td>
<td>SDAx0</td>
</tr>
<tr>
<td>11</td>
<td>GPIO26</td>
<td>SCLx1</td>
</tr>
<tr>
<td>12</td>
<td>GPIO27</td>
<td>SDAx1</td>
</tr>
<tr>
<td>13</td>
<td>GPIO28</td>
<td>SCLx2</td>
</tr>
<tr>
<td>14</td>
<td>GPIO29</td>
<td>SDAx2</td>
</tr>
<tr>
<td>15</td>
<td>GPIO30</td>
<td>SCLx3</td>
</tr>
<tr>
<td>16</td>
<td>GPIO31</td>
<td>SDAx3</td>
</tr>
</tbody>
</table>

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**

```c
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**

```c
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**

```c
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**
```c
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.

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
<th>Master mode</th>
<th>Slave mode</th>
<th>Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCK</td>
<td>SPI Master Clock</td>
<td>Output</td>
<td>Input</td>
<td>HiZ</td>
</tr>
<tr>
<td>MOSI</td>
<td>Master Out Slave In</td>
<td>Output</td>
<td>Input</td>
<td>HiZ</td>
</tr>
<tr>
<td>MISO</td>
<td>Master In Slave Out</td>
<td>Input</td>
<td>Output**</td>
<td>HiZ</td>
</tr>
<tr>
<td>SS0</td>
<td>Slave Select 0</td>
<td>Output Hi</td>
<td>Input</td>
<td>HiZ</td>
</tr>
<tr>
<td>SS1</td>
<td>Slave Select 1..4</td>
<td>HiZ</td>
<td>Not in use</td>
<td>HiZ</td>
</tr>
<tr>
<td>SS2</td>
<td></td>
<td></td>
<td></td>
<td>HiZ</td>
</tr>
<tr>
<td>SS3</td>
<td></td>
<td></td>
<td></td>
<td>HiZ</td>
</tr>
<tr>
<td>SS4</td>
<td></td>
<td></td>
<td></td>
<td>HiZ</td>
</tr>
</tbody>
</table>

*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**
Functions
- sub_spi_config
- sub_spi_transfer
- sub_spi_transfer_ess
- sub_sdio_transfer

3.5.1 sub_spi_config

Synopsis
```c
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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_ENABLE</td>
<td>Enable SUB-20 SPI module.</td>
</tr>
<tr>
<td>SPI_SLAVE</td>
<td>SPI module is in Slave mode. sub_spi_transfer can work only when SPI module is in Master mode</td>
</tr>
<tr>
<td>SPI_CPOL_RISE</td>
<td>SCK is low when idle. See SPI Polarity and Phase explanations</td>
</tr>
<tr>
<td>SPI_CPOL_FALL</td>
<td>SCK is high when idle. See SPI Polarity and Phase explanations</td>
</tr>
<tr>
<td>SPI_SMPL_SETUP</td>
<td>Sample data on leading SCK edge, setup on trailing. See SPI Polarity and Phase</td>
</tr>
<tr>
<td>SPI_SETUP_SMPL</td>
<td>Setup data on leading SCK edge, sample on trailing. See SPI Polarity and Phase</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_LSB_FIRST</td>
<td>Transmit LSB first</td>
</tr>
<tr>
<td>SPI_MSB_FIRST</td>
<td>Transmit MSB first</td>
</tr>
<tr>
<td>SPI_CLK_8MHZ</td>
<td>SPI SCK frequency 8 MHz</td>
</tr>
<tr>
<td>SPI_CLK_4MHZ</td>
<td>SPI SCK frequency 4 MHz</td>
</tr>
<tr>
<td>SPI_CLK_2MHZ</td>
<td>SPI SCK frequency 2 MHz</td>
</tr>
<tr>
<td>SPI_CLK_1MHZ</td>
<td>SPI SCK frequency 1 MHz</td>
</tr>
<tr>
<td>SPI_CLK_500KHZ</td>
<td>SPI SCK frequency 500 KHz</td>
</tr>
<tr>
<td>SPI_CLK_250KHZ</td>
<td>SPI SCK frequency 250 KHz</td>
</tr>
<tr>
<td>SPI_CLK_125KHZ</td>
<td>SPI SCK frequency 125 KHz</td>
</tr>
</tbody>
</table>

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.

• *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
<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPI_CLK_HZ(FHZ)</td>
<td>FW version</td>
</tr>
<tr>
<td></td>
<td>0.3.3 or grater</td>
</tr>
<tr>
<td></td>
<td>Library version</td>
</tr>
<tr>
<td></td>
<td>0.1.12.25 or grater</td>
</tr>
</tbody>
</table>
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_HHHHLL 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_LLLH  SS goes low and stays low during first 2 bytes transfer, after that it goes high
  SS_LLLLH SS goes low and stays low during first 3 bytes transfer, after that it goes high
  SS_LLO   SS goes low and stays low during first 4 bytes 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

```c
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

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Select SS0</td>
</tr>
<tr>
<td>1</td>
<td>Select SS1</td>
</tr>
<tr>
<td>2</td>
<td>Select SS2</td>
</tr>
<tr>
<td>3</td>
<td>Select SS3</td>
</tr>
<tr>
<td>4</td>
<td>Select SS4</td>
</tr>
<tr>
<td>H</td>
<td>SS High</td>
</tr>
<tr>
<td>L</td>
<td>SS Low</td>
</tr>
<tr>
<td>^</td>
<td>SS High Byte Transfer</td>
</tr>
<tr>
<td>v</td>
<td>SS Low Byte Transfer</td>
</tr>
<tr>
<td>-</td>
<td>SS Low No Byte Transfer</td>
</tr>
<tr>
<td></td>
<td>Small delay without SS changes</td>
</tr>
</tbody>
</table>

Return value

On success function returns 0. Otherwise error code.

Example

```c
/* 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

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

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

1. SS changes according to \texttt{ss\_config} parameter.
2. \texttt{out\_sz} bytes are transmitted via MOSI. MOSI will be in output state during transfer.
3. MOSI state is changed to HiZ after the last bit is transmitted. If required SS changes.
4. \texttt{in\_sz} bytes received from MISO input. MOSI left in HiZ state.
5. SS changes according to \texttt{ss\_config} parameter

SPI Clock - SCK phase and polarity can be configured with \texttt{sub\_spi\_config}.

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

### Parameters
- \texttt{out\_buf} - Output data buffer.
- \texttt{in\_buf} - Input buffer to store read data.
- \texttt{out\_sz} - Number of bytes to transmit in range 0..60.
- \texttt{in\_sz} - Number of bytes to receive in range 0..60.
- \texttt{ss\_config} - Determines selection and operation of SS pin. \texttt{ss\_config} value must be created with macro \texttt{SS\_CONF(SS\_N,SS\_MODE)},

© 2018 xdimax.com rev 2.17
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**
```c
/* 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 a 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**

```
| READ | 111...111 | 01 | 10 | AAAAA | RRRRR | Z0 | DDDDDDDDDDDDDDDDDD |
| WRITE | 111...111 | 01 | 01 | AAAAA | RRRRR | 10 | DDDDDDDDDDDDDDDDD |
```

- **Clause 45 MDIO frame format**
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
Beginning 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 );
3.6.2 sub_mdio45

**Synopsis**

\[
\text{int sub_mdio45( sub_handle hndl, int op, int prtad, int devad,}
\text{ int data, int* content )}
\]

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**

\[
\text{int sub_mdio_xfer( sub_handle hndl, int count, union sub_mdio_frame* mdios )}
\]
\[
\text{int sub_mdio_xfer_ex( sub_handle hndl, int channel, int count,}
\text{ union sub_mdio_frame* mdios )}
\]

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

\[
\text{union sub_mdio_frame}
\]
\[
\{
\]

© 2018 xdimax.com rev 2.17
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 \texttt{op} field value.

\texttt{clause22.op} can be
\begin{itemize}
\item \texttt{SUB_MDIO22_READ} \hspace{1cm} \textit{READ operation}
\item \texttt{SUB_MDIO22_WRITE} \hspace{1cm} \textit{WRITE operation}
\end{itemize}

\texttt{clause45.op} can be
\begin{itemize}
\item \texttt{SUB_MDIO45_ADDR} \hspace{1cm} \textit{ADDRESS operation}
\item \texttt{SUB_MDIO45_WRITE} \hspace{1cm} \textit{WRITE operation}
\item \texttt{SUB_MDIO45_PRIA} \hspace{1cm} \textit{POST-READ-INCREMENT-ADDRESS operation}
\item \texttt{SUB_MDIO45_READ} \hspace{1cm} \textit{READ operation}
\end{itemize}

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

\textbf{Return value}
On success function returns 0. Otherwise \texttt{error code}.

\textbf{Example}
\begin{verbatim}
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 );
\end{verbatim}

\textbf{Compatibility}
\begin{itemize}
\item \texttt{sub_mdio_xfer}
\begin{itemize}
\item FW version \hspace{1cm} 0.2.1 or grater
\item Library version \hspace{1cm} 0.1.12.12 or grater
\end{itemize}
\item \texttt{sub_mdio_xfer_ex}
\begin{itemize}
\item FW version \hspace{1cm} 0.2.8 or grater
\item Library version \hspace{1cm} 0.1.12.18 or grater
\end{itemize}
\end{itemize}
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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>State</th>
<th>Output Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;0&quot;</td>
<td>Input state</td>
<td>HiZ, pull-up</td>
</tr>
<tr>
<td>&quot;1&quot;</td>
<td>Output state</td>
<td>LOW, HIGH</td>
</tr>
</tbody>
</table>

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

```c
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

```c
/* 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

```c
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

```c
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

```c
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

```c
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 current GPIOB input status reads from SUB-20.
· *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**

```c
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 GPIO Watchdog

Beginning from the FW version 0.4.4 and library version 0.1.12.32, SUB-20 provides GPIO Watchdog functionality.

GPIO Watchdog is used to guarantee that SUB-20 GPIO will be returned to their idle state after given period of time. GPIO watchdog is implemented in SUB-20 CPU and once activated works independently from the USB host SW application. Thus if for some reason host SW got stuck or failed, GPIO will be returned to their idle state by SUB-20 CPU once the GPIO Watchdog got expired.

**GPIO Watchdog implementation**
The state diagram below explains GPIO Watchdog implementation.
• RST - means any GPIO write or GPIO configuration operation (sub_gpio_write or sub_i2c_config).
• CFG - means GPIO Watchdog configuration with sub_gpio_wdt_set
• EXPIRED - means that GPIO Watchdog got expired and GPIO are returned to idle state
• STOP: 0 - GPIO Watchdog is stopped, 1 - GPIO Watchdog is counting
• EXP: 0 - GPIO Watchdog is not expired, 1 - GPIO Watchdog is expired

States:
1. After SUB-20 reset GPIO Watchdog is disabled. It's period is set to 0.
2. GPIO Watchdog is enabled by sub_gpio_wdt_set but still not counting.
3. GPIO watchdog starts counting after RST operation.
4. There was no RST operations during GPIO Watchdog period. Watchdog got expired and stopped. GPIO are reset to idle state.
5. GPIO Watchdog is restarted with RST operation.

Functions
sub_gpio_wdt_set
sub_gpio_wdt_get

3.7.7.1 sub_gpio_wdt_set

Synopsis
int sub_gpio_wdt_set( sub_handle hndl, int to_ms, int dir_cfg, int out_cfg )

Configure GPIO Watchdog. Parameters dir_cfg and out_cfg define idle state of GPIO if GPIO Watchdog got expired.

Parameters
to_ms - GPIO Watchdog period in ms. GPIO Watchdog will be disabled if period is set to 0. The maximal value is 4000ms = 4s.

dir_cfg - GPIO direction configuration bits in idle state (similar to set parameter of the sub_gpio_config function).

gpio_get - GPIO output configuration bits in idle state (similar to set parameter of the sub_gpio_write function).

Return value
On success function returns 0. Otherwise error code.

See also
Error Codes

Compatibility
FW version 0.4.4 or grater
Library version 0.1.12.32 or grater

3.7.7.2 Sub_gpio_wdt_get

Synopsis

```c
int sub_gpio_wdt_get( sub_handle hndl, int* pto_ms, int* pdir_cfg, int* pout_cfg, unsigned char* pflags )
```

Read GPIO Watchdog settings and status.

Parameters
- pto_ms - point to store GPIO Watchdog period in ms.
- pdir_cfg - point to store idle state GPIO direction configuration bits.
- pout_cfg - pointer to store idle state GPIO output configuration bits.
- pflags - pointer to store GPIO Watchdog status flags:
  - SUB_GPIO_WDT_EXPIRED: GPIO Watchdog got expired
  - SUB_GPIO_WDT_STOP: GPIO Watchdog is stopped

Return value
On success function returns 0. Otherwise error code.

See also
Error Codes

Compatibility
FW version 0.4.4 or grater
Library version 0.1.12.32 or grater

3.7.8 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.
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

### 3.7.8.1 sub_edge_config

**Synopsis**

```c
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:

<table>
<thead>
<tr>
<th>Ex1</th>
<th>Ex0</th>
<th>Edge Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Any change</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Falling edge</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Rising edge</td>
</tr>
</tbody>
</table>

- `*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**

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

/* EDG0 = 01B - Any change */
/* EDG1 = 01B - Any change */
/* EDG2 = 11B - Rising edge */
```
Sub-20 API

/* EDG7..EDG3 not detected */
rc = sub_edge_config( hndl, 0x0035, 0 );  /* set = 0011.0101B */

See also
Error Codes

3.7.8.2 sub_edge_read

Synopsis

```c
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

3.7.9 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

3.7.9.1 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

3.7.9.2 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

3.7.10 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_config`
- `sub_pwm_set`

3.7.10.1 sub_pwm_config

Synopsis

```c
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

\[
F_{pwm} = \frac{1000000}{(\text{res}_us \times \text{limit})} \text{ Hz}
\]

PWM frequency range is

\[
F_{pwm\_min} = \frac{1000000}{(20 \times 2)} = 25000 \text{ Hz} = 25\text{KHz}
\]
\[
F_{pwm\_max} = \frac{1000000}{(16384 \times 255)} = 0.238 \text{ Hz}
\]

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

3.7.10.2 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
```c
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
```c
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

<table>
<thead>
<tr>
<th>ADC mux code</th>
<th>Single Ended</th>
<th>Differential Positive</th>
<th>Differential Negative</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC_S0</td>
<td>ADC0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC_S1</td>
<td>ADC1</td>
<td></td>
<td></td>
<td>N.A.</td>
</tr>
</tbody>
</table>
### 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

```c
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
```c
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.

<table>
<thead>
<tr>
<th>ANSI C notation</th>
<th>Hex value</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\f</td>
<td>0x0C</td>
<td>Clear LCD and go to first position</td>
<td>&quot;\fHello&quot;</td>
</tr>
<tr>
<td>\r</td>
<td>0x0D</td>
<td>Go to first position</td>
<td>&quot;\r0123&quot;</td>
</tr>
<tr>
<td>\n</td>
<td>0x0A</td>
<td>Go to next string</td>
<td>&quot;\rHello\nWorld&quot;</td>
</tr>
<tr>
<td>\eXY</td>
<td>0x1B X Y</td>
<td>Go to position X, Y</td>
<td>&quot;\e00Hello\e01World&quot;</td>
</tr>
</tbody>
</table>

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
```c
sub_lcd_write( hndl, "\fHello\nWorld" );
sub_lcd_write( hndl, "\f\e20Hello\e21World" );
sub_lcd_write( hndl, "\r1\n2" );
sub_lcd_write( hndl, "\fT:abc\nR:def" );
```

Compatibility
- Required firmware version > 0.0.4
- Required DLL version > 0.1.12.2
3.10 RS232 RS485 Functions

RS232 RS485 Frame format

```
(IDLE) St 0 1 2 3 4 [5] [6] [7] [8] [P] Sp1 [Sp2] (St / IDLE)
```

- **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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS_RX_ENABLE</td>
<td>Enable UART receiver</td>
</tr>
<tr>
<td>RS_TX_ENABLE</td>
<td>Enable UART transmitter</td>
</tr>
<tr>
<td>RS_CHAR_5</td>
<td>Data Bits</td>
</tr>
<tr>
<td>RS_CHAR_6</td>
<td>Data Bits</td>
</tr>
<tr>
<td>RS_CHAR_7</td>
<td>Data Bits</td>
</tr>
<tr>
<td>RS_CHAR_8</td>
<td>Data Bits</td>
</tr>
<tr>
<td>RS_CHAR_9</td>
<td>Data Bits</td>
</tr>
<tr>
<td>RS_PARITY_NONE</td>
<td>Parity</td>
</tr>
<tr>
<td>RS_PARITY_EVEN</td>
<td>Parity</td>
</tr>
<tr>
<td>RS_PARITY_ODD</td>
<td>Parity</td>
</tr>
<tr>
<td>RS_STOP_1</td>
<td>Stop Bits</td>
</tr>
<tr>
<td>RS_STOP_2</td>
<td>Stop Bits</td>
</tr>
</tbody>
</table>

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**

```c
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**

```c
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:

<table>
<thead>
<tr>
<th>rx_msg_us</th>
<th>rx_byte_us</th>
<th>Message reception will last no more then 4s</th>
<th>First byte reception will last no more then 4s. Every next byte should be received in rx_byte_us time</th>
<th>Message reception will last no more then rx_msg_us time</th>
</tr>
</thead>
<tbody>
<tr>
<td>=0</td>
<td>=0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=0</td>
<td>&gt;0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;0</td>
<td>=0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 greater than 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

<table>
<thead>
<tr>
<th>byte 0</th>
<th>byte 1</th>
<th>...</th>
<th>byte n</th>
<th>byte n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit .8 LSB</td>
<td>bits 7..0</td>
<td>...</td>
<td>bit .8 LSB</td>
<td>bits 7..0</td>
</tr>
</tbody>
</table>

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 than 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

© 2018 xdimax.com rev 2.17
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

```c
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

```c
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 \texttt{sz}. In case of error negative error code (defined in \texttt{errno.h}) will be returned and if applicable \texttt{sub_errno} will be set to the corresponding value. Possible error codes are:

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

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

### 3.11.2 SPI Slave

\textbf{Configuration}
To use SUB-20 in SPI slave mode following should be done:

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

\textbf{Example}
Below is an example to exchange 12 bytes with SPI master:

```c
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;
}
```

\textbf{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

\textbf{Configuration}
To use SUB-20 in I2C slave mode following should be done:

- Configure I2C Slave module with \texttt{sub_i2c_config}. Set desired I2C slave address and optional flag(s).
- Connect I2C Slave module to FIFO with \texttt{sub_fifo_config}. Set \texttt{FIFO_SELECT_I2C} flag.
- If required write data to FIFO with \texttt{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:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OK</td>
</tr>
<tr>
<td>1</td>
<td>SUB Device not found</td>
</tr>
<tr>
<td>2</td>
<td>Can't open SUB device</td>
</tr>
<tr>
<td>3</td>
<td>Can't set configuration</td>
</tr>
<tr>
<td>4</td>
<td>Can't claim interface</td>
</tr>
<tr>
<td>5</td>
<td>Failed to setup async transaction</td>
</tr>
<tr>
<td>6</td>
<td>Failed to submit async transaction</td>
</tr>
<tr>
<td>7</td>
<td>Bulk write failed</td>
</tr>
<tr>
<td>8</td>
<td>Bulk read failed</td>
</tr>
<tr>
<td>9</td>
<td>Bulk read incomplete</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Condition</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating Temperature</td>
<td></td>
<td>-20°C</td>
<td>+65°C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Voltage on any Pin with respect to Ground</td>
<td>-0.5V to VCC + 0.5V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum Operating Voltage</td>
<td>+6V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC Current per I/O Pin</td>
<td>40mA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DC Current VCC and GND Pins</td>
<td>200mA</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2 DC Characteristics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Condition</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input Low Voltage VCC = 3.3V - 5.5V</td>
<td>0.5</td>
<td>0.2</td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>Input High Voltage VCC = 3.3V - 5.5V</td>
<td>0.6VCC</td>
<td></td>
<td>VCC + 0.5</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>Output Low Voltage IOL = 10mA, VCC = 5V</td>
<td>0.3</td>
<td>0.2</td>
<td>0.7</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOL = 5mA, VCC = 3.3V</td>
<td></td>
<td>0.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Output High Voltage IOH = -20mA, VCC = 5V</td>
<td>4.2</td>
<td>2.3</td>
<td>4.5</td>
<td></td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>IOH = -10mA, VCC = 3.3V</td>
<td></td>
<td>2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Input Leakage Current I/O Pin VCC = 5.5V, pin low</td>
<td>1</td>
<td></td>
<td></td>
<td>µA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>VCC = 5.5V, pin high</td>
<td></td>
<td>1</td>
<td></td>
<td>µA</td>
</tr>
<tr>
<td></td>
<td>Power Consumption Current VCC = 5V</td>
<td>19</td>
<td></td>
<td>30</td>
<td>mA</td>
<td></td>
</tr>
</tbody>
</table>

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 IOL, for GPIO4-GPIO7 should not exceed 100 mA.
2. The sum of all IOL, for GPIO0-GPIO3, GPIO8-GPIO15 should not exceed 100 mA.
3. The sum of all IOL, for GPIO24-GPIO31 should not exceed 100 mA.
4. The sum of all IOL, for GPIO16-GPIO23 should not exceed 100 mA.
5. If IOL exceeds the test condition, VOL may exceed the related specification. Pins are not guaranteed to sink current greater than the listed test condition.
6. The sum of all IOH, for GPIO4-GPIO7 should not exceed 100 mA.
7. The sum of all IOH, for GPIO0-GPIO3, GPIO8-GPIO15 should not exceed 100 mA.
8. The sum of all IOH, for GPIO24-GPIO31 should not exceed 100 mA.
9. The sum of all IOH, for GPIO16-GPIO23 should not exceed 100 mA.

4.3 AC Characteristics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Condition</th>
<th>Min</th>
<th>Typ</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SPI Master frequency</td>
<td>F_SPI</td>
<td>0.125</td>
<td>8</td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SPI Slave frequency</td>
<td>F_SPI_SLAVE</td>
<td>2</td>
<td></td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MDC frequency</td>
<td>F_MDC</td>
<td>1</td>
<td></td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>High Speed (HS) I2C Master</td>
<td>F_SCLH</td>
<td></td>
<td>3.125</td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td></td>
<td>HS SCL LOW period</td>
<td>t_HLOW</td>
<td>187.5</td>
<td>ns</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HS SCL HIGH period</td>
<td>t_HHIGH</td>
<td>125</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
</tbody>
</table>
5 Ordering Information

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

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.