ALCIOM
5, Parvis Robert Schuman
92370 CHAVILLE - FRANCE

Tel/Fax : 01 47 09 30 51
✉ contact@alciom.com

www.alciom.com

Project :

SmartBrick

DOCUMENT :

SmartBus specification

REFERENCE :

AL/RL/1048/004

DATE :	VERSION :	AUTHOR :
09/05/2011	**1G**	**Robert Lacoste / ALCIOM**

SUMMARY:

This document is the specification of the SmartBus standardized interface used to link SmartBrick modules to a host system as well as SmartBrick modules togheter. It describes the connector pinout, physical layers supported as well as the automatic message routing protocol, standardized message formats and clock and power distribution scheme.

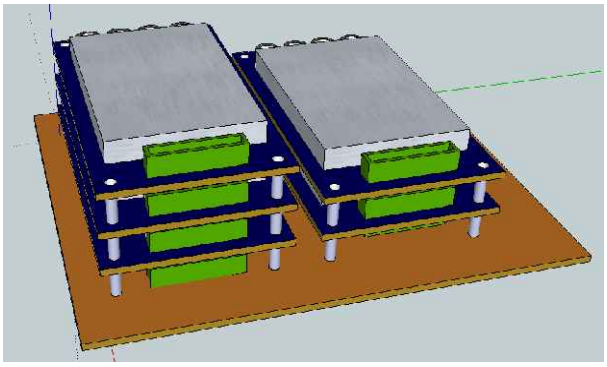
DOCUMENT HISTORY

DATE	VERSION	AUTHOR	COMMENT
05/12/10	1A	R.Lacoste / ALCIOM	Initial version
20/01/11	1B	R.Lacoste / ALCIOM	Extended addressing
21/01/11	1C	R.Lacoste / ALCIOM	Several class per module type
26/01/11	1D	R.Lacoste / ALCIOM	Approved base version
27/01/11	1E	R.Lacoste / ALCIOM	Minor corrections & precisions
31/01/11	1F	R.Lacoste / ALCIOM	Selective indications & simplifications
09/05/11	1G	R.Lacoste / ALCIOM	Identity in ASCII form

CONTENTS

1 Introduction

The SmartBrick products from ALCIOM are high performance analog and mixed-signal OEM modules. These modules are both daisy-chainable and stackable to build plug'n play compact and efficient smart instrumentation and acquisition systems.



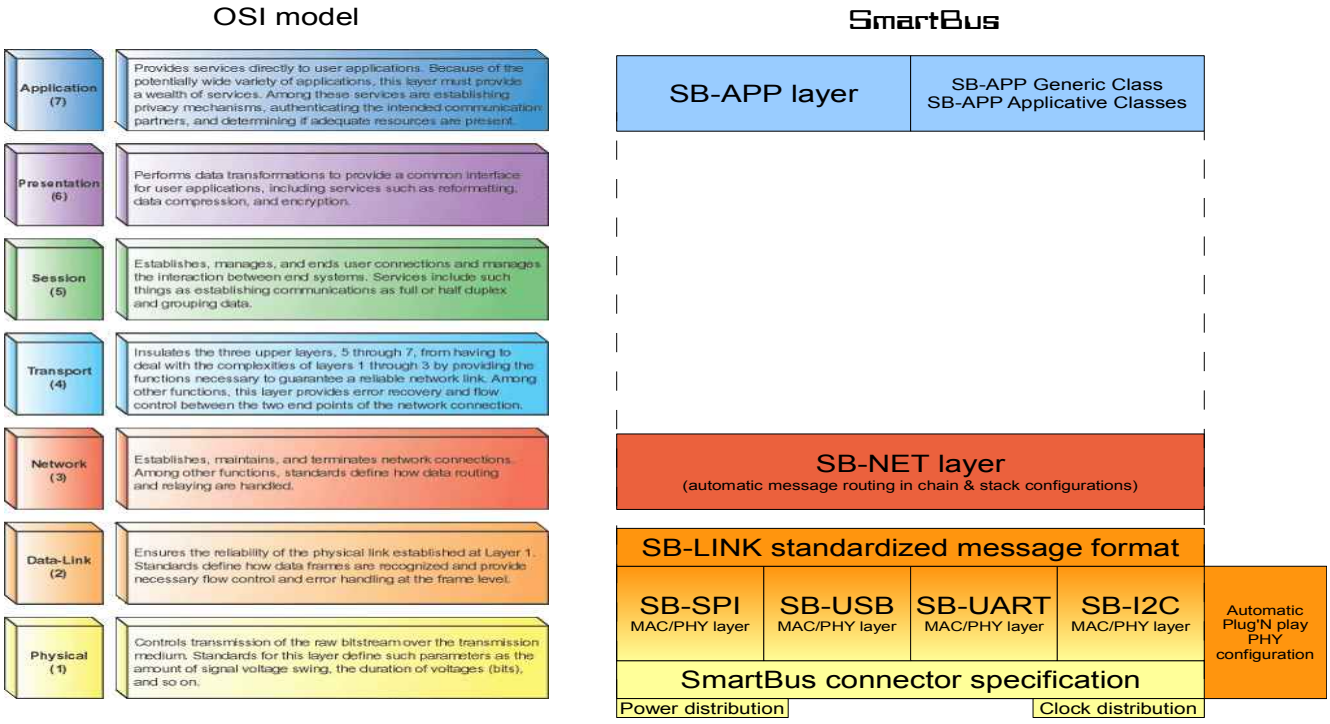
The SmartBrick modules are driven by a host system (microcontroller, DSP or PC, either embedded or remote) through the patented ultra-flexible SmartBus interface, compatible with SPI, USB, UART or I2C links. Open source software libraries provides an easy interface with any user-developed application in virtually any language : Labview, C, C++, Python, C#, Visual Basic, Mathlab/Scilab, etc.

This document is the specification of the SmartBus standardized interface. This interface is used to link SmartBrick modules to a host system as well as SmartBrick modules together. It describes the physical layers supported as well as the automatic message routing protocol, standardized message formats and clock and power distribution scheme.

2 Overall architecture

2.1 OSI Model

In order to be as portable and flexible as possible, the SmartBus protocol is structured as several independent protocol layers, following the OSI 7-layers protocol model as follows :

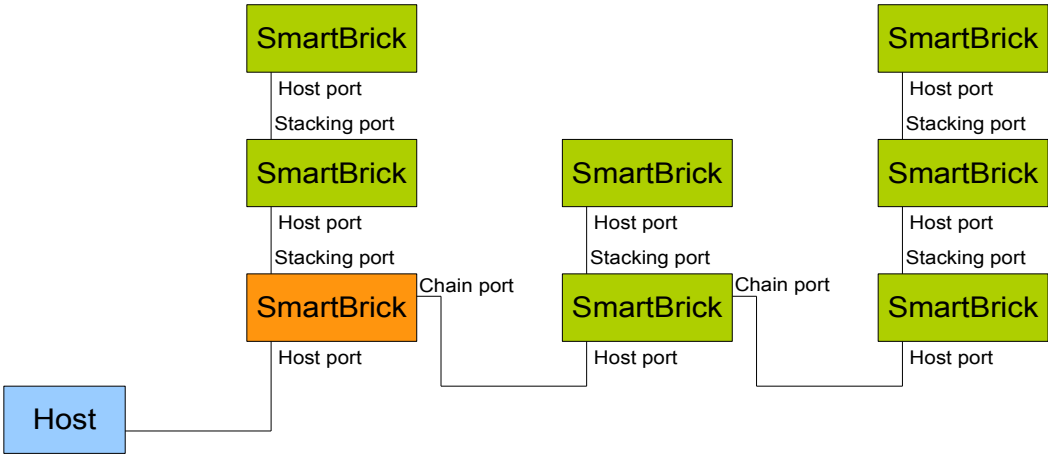


Starting from the lowest layers :

- The SmartBus connector specification describes rigorously the electrical and mechanical characteristics of the top and bottom connectors used on every SmartBrick module ;
- Hardware specifications are related to power, clock and hardware trigger distribution through a SmartBrick system. These are actually hardware only specifications but are included in this document for completeness ;
- An Automatic plug'N play PHY configuration protocol allows to automatically select the required MAC/PHY protocol depending on the user's needs
- Four interchangeable MAC/PHY layers manage the transportation of raw messages on the four supported low level interfaces (SPI, USB, UART and I2C). Each layer specifies how bits are actually encoded, how messages are separated, how flow control is managed, etc
- The SB-LINK standardized message format provides a unified access to low level MAC/PHY protocols to the upper layers ;
- The SB-NET layer manages the automatic routing of commands and answers between the host and the set of SmartBrick modules through modules stacking and/or module chaining, transparently for the user ;
- Lastly the SB-APP layer specifies the set(s) of messages that are supported by a SmartBrick module. These messages are grouped in **Classes**. All modules must support at least the message set defined in the SB-APP Generic Class, but supports also one or several custom SB-APP Applicative Classes

2.2 System topology

The overall topology of a SmartBrick system is the following :



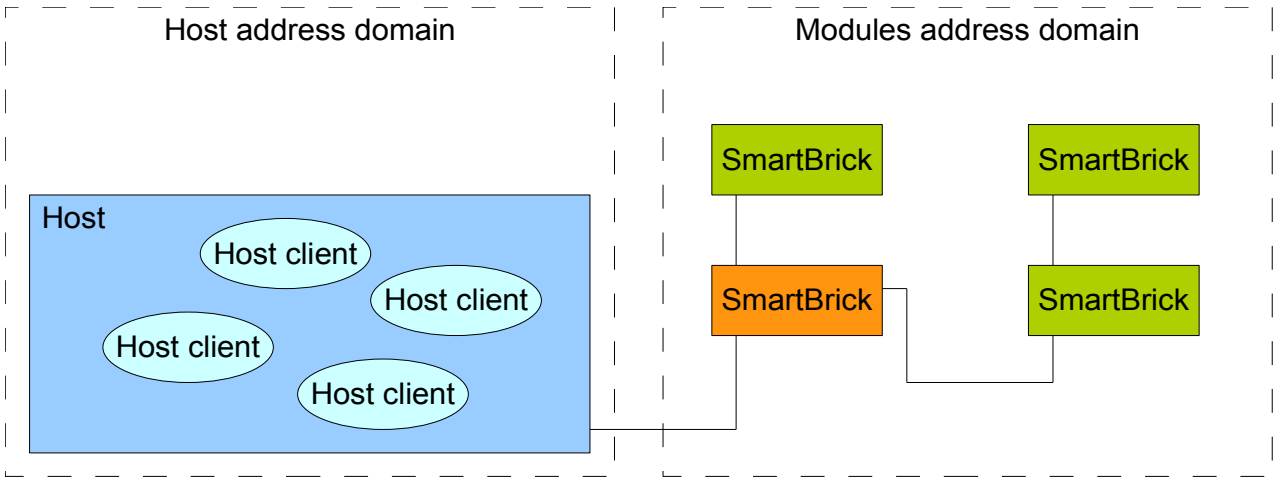
The host system (PC, microcontroller, DSP, etc) is connected to a first SmartBrick module through a very flexible SmartBus link (configure either as USB, SPI, UART or I2C). Each module has then two other SmartBus interfaces, which allows either to stack other modules on the first one or to chain several stacks of modules, or single modules if needed. This allows very flexible configurations, either stacked, chained, or both.

Only the first or these links, between the host and the first SmartBrick module, is configurable as SPI/USB/UART/I2C. All the other links are done through high speed SPI links for best performances. The remapping and protocol conversion is fully transparent to the user.

2.3 Address domains

The SmartBus protocol allows the routing of messages between **nodes**, which can be either SmartBrick modules or **host clients**. The notion of host client allows to have for example several tasks running on the host, each of them able to access to the SmartBrick resources.

From the SmartBus standpoint host clients are identified by specific addresses which are managed by a host-side routing process. Therefore SmartBus addresses are split into two address domains : host client addresses SmartBrick module addresses.



2.4 Modules addresses

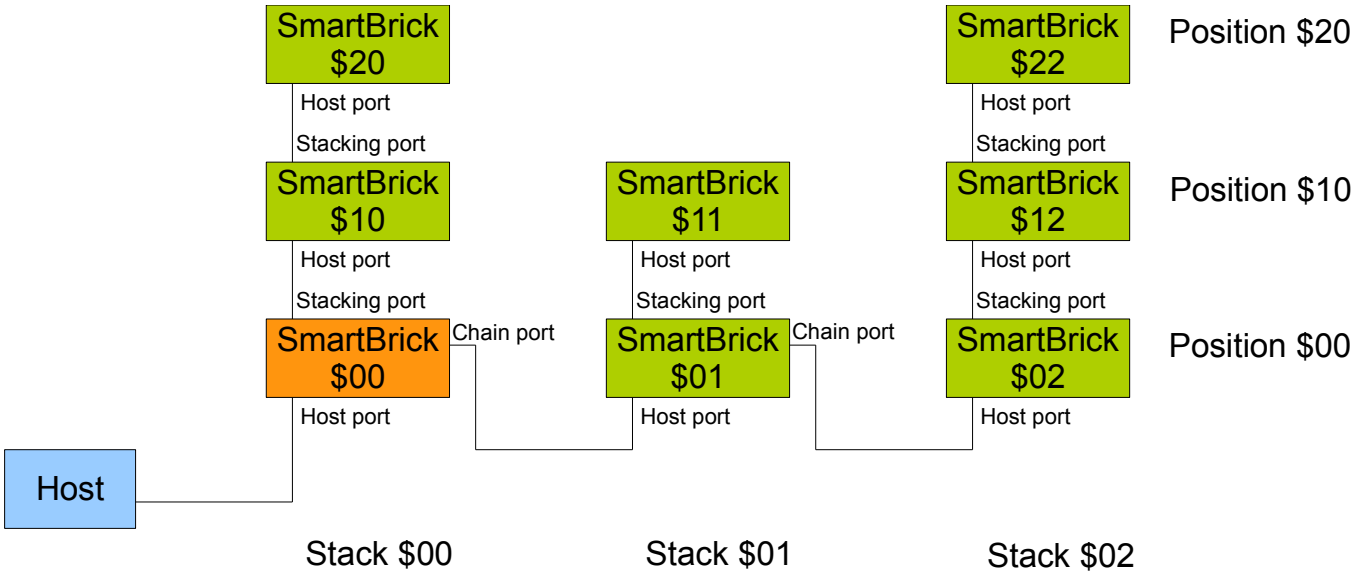
The **SmartBus basic addressing mode** allows up to 16 stacks of 8 modules.

*Nota : The SmartBus protocol is extensible through an **extended addressing mode** (cf chapter 7), but the current implementation is limited to basic addressing mode, meaning one byte addressing or 16 stacks of 8 modules. This implementation provides the best performances and is adequate for the vast majority of applications.*

Each module has a given 7-bit address, defined automatically by the tree structure. In the basic addressing mode :

- The 3 MSB bits of the address define the module position in the stack from $0 to $7
- The 4 LSB bits of the address define the stack number from $0 to $F (15 decimal)

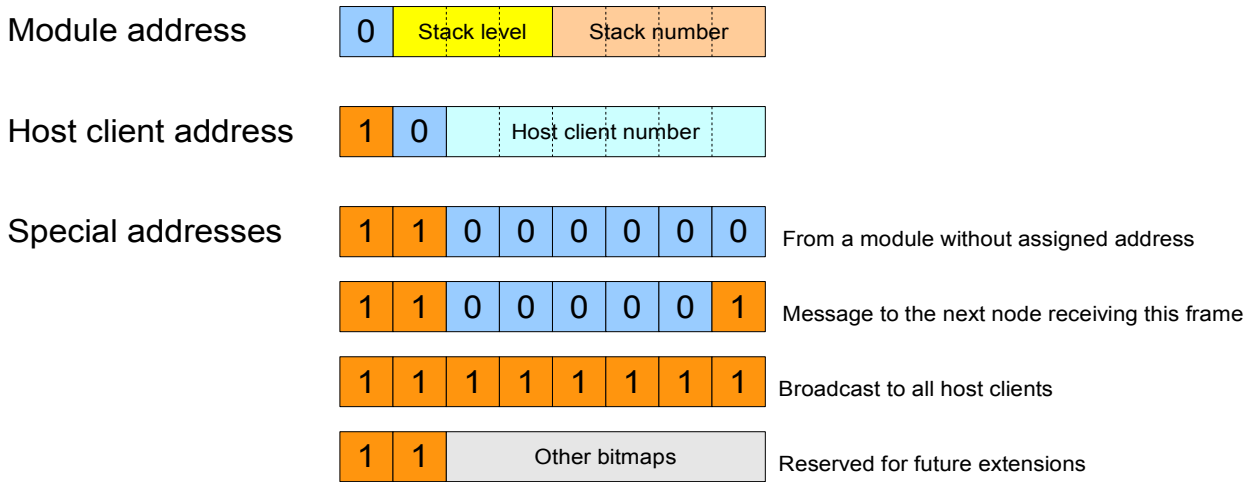
The module connected to the host is then always module $00. This is illustrated below :



At startup, the module $00 (which identifies itself through the plug'n play configuration, cf 3,2) assigns itself the address $00. Each other module asks for its address to its parent module (through its host post), and the address mapping is progressively done through the whole network, transparently to the user (cf 8.2).

2.5 Addresses coding

In basic mode nodes addresses through the SmartBus system are encoded on a **single byte**, using the following structure :



Therefore 0x47 is the 5th module (4+1) of the 8th stack (7+1), 0x85 is the server client number 5, and 0xFF is a broadcast to all server clients.

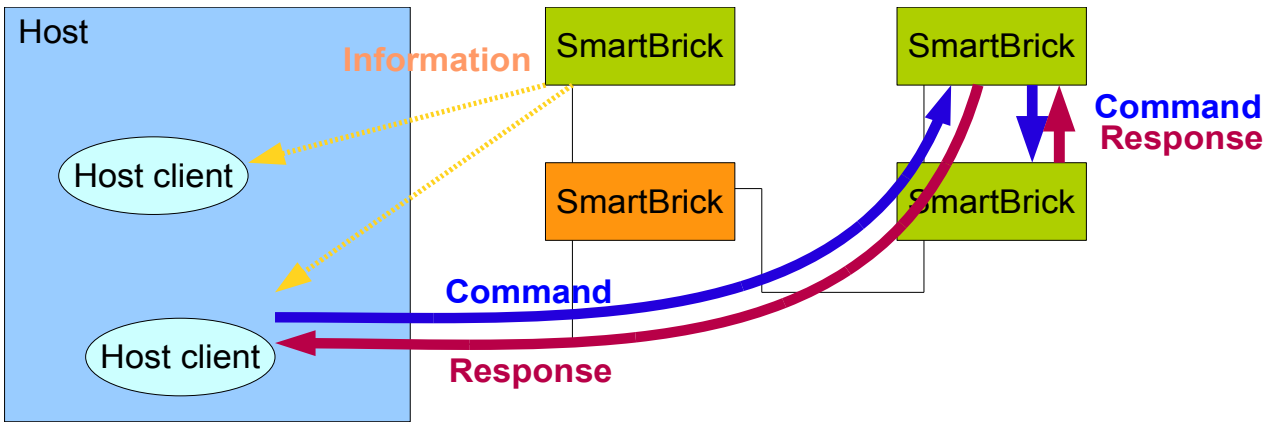
2.6 Message types

Messages through a SmartBrick system are of three different types :

- **Commands** : Are sent by a requesting node to a target node, and request the target node to do a specific action. Commands are **always unicast messages** (one sender, one receiver). The target node must **always send a Response** back to the requesting node. If the target node is not existing then a Response will in any case be sent back to the requester by another node, flagging the error (cf 8,3). The SmartBus protocol allows commands sent by host (usual case) or even from one module to another module for specific applications;

- **Responses** : Are sent back by a target node to the requesting node after a command. Responses are always unicast messages too ;

- **Indications** (optional) : Could be send spontaneously by any module. Indications are always **broadcasted to all host clients.** In order to keep the use of the system as easy as possible Indications are disabled by default. If a host system supports Indications then it can enable Indications on one or several modules through a specific command.



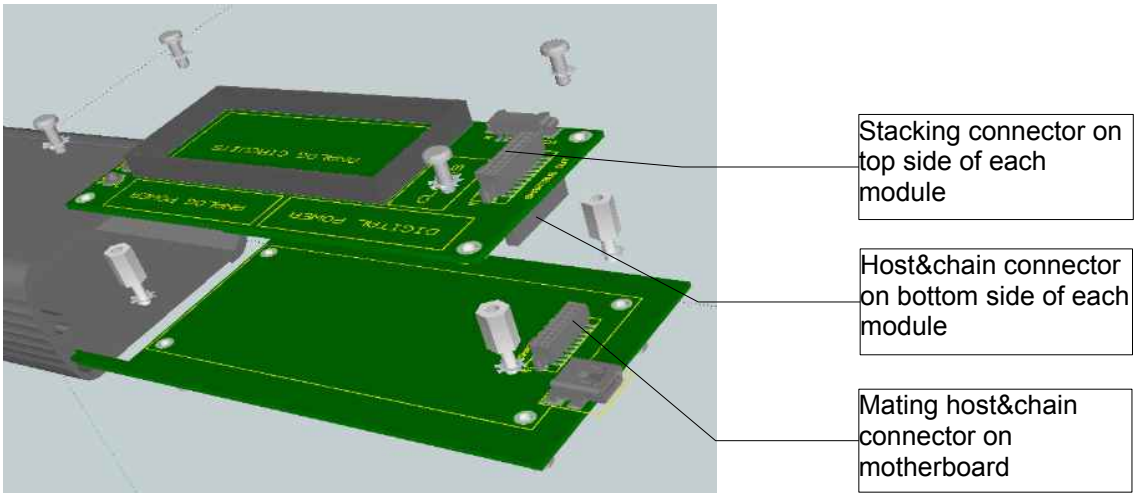
3 SmartBus connector

3.1 Connector types

Each SmartBrick module has two SmartBus connectors :

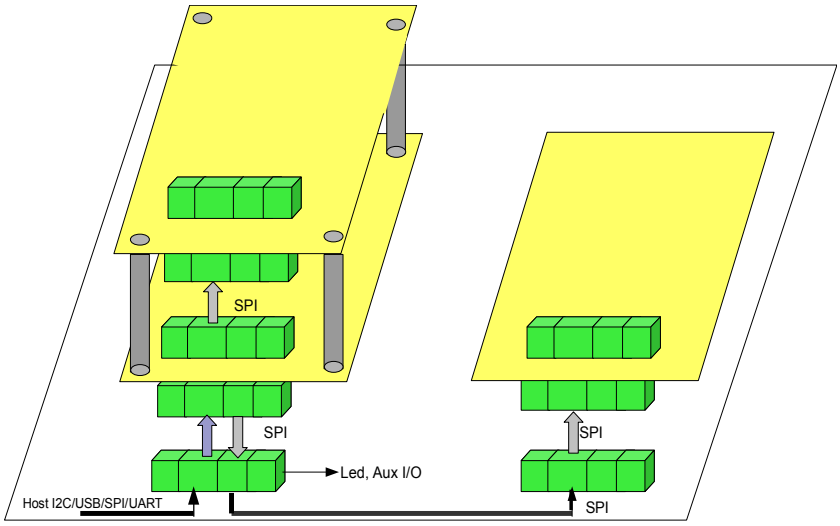
- One "**host&chain connector**" on the bottom side of the module. This connector allows to interconnect the module to its controlling host through SPI, USB, UART or I2C, and allows also to link other stacks of modules through a daisy chaining. The same connector, with opposite gender, is present on the motherboard receiving a module stack

- One reduced function "**stacking connector**" on the top side of the module. This connector allows to stack several modules transparently.



These two connectors are 2x10 pin TYCO Series AMPMODU 50/50 connectors. Precise references and location on the boards are specified in the document « SmartBrick mechanical specification », reference AL/RL/1048/001.

The host&chain connector offers in fact two connections : one to the host, controlling the full SmartBrick system, and one to another stack of modules in a daisy-chain fashion :



3.2 Automatic plug'n play configuration

The determination of the low-level MAC/PHY protocol used to communicate with a given SmartBrick module is done automatically through three mode-selection pins available on the SmartBus host&chain connector : SEL1, SEL2, SEL3. These pins are sensed by the module at power-up (and only at power-up), and define the MAC/PHY protocol through the following three-state matrix :

SEL1	SEL2	SEL3	Mode	Sous-mode
L	L	L	USB to HOST	Slave, full speed
L	L	Z	Reserved	
L	L	H	USB, bootloader	Slave, full speed
L	Z	L	SPI to HOST	S3=CS, active
L	Z	Z	Reserved	
L	Z	H	SPI to HOST	S3=CS, inactive
L	H	L	SPI to MODULE	S3=CS, active
L	H	Z	Reserved	
L	H	H	SPI to MODULE	S3=CS, inactive
Z	L	L	UART to HOST	1200bps, 8N1
Z	L	Z	UART to HOST	9600bps, 8N1
Z	L	H	UART to HOST	19200bps, 8N1
Z	Z	L	UART to HOST	38400bps, 8N1
Z	Z	Z	UART to HOST	57600bps, 8N1
Z	Z	H	UART to HOST	115200bps, 8N1
Z	H	L	UART to HOST	230400bps, 8N1
Z	H	Z	UART to HOST	19200bps, 8E1
Z	H	H	UART to HOST	230400bps, 8E1
H	L	L	I2C to HOST	Address 0x50
H	L	Z	I2C to HOST	Address 0x51
H	L	H	I2C to HOST	Address 0x52
H	Z	L	I2C to HOST	Address 0x53
H	Z	Z	I2C to HOST	Address 0x54
H	Z	H	I2C to HOST	Address 0x55
H	H	L	I2C to HOST	Address 0x56
H	H	Z	I2C to HOST	Address 0x57
H	H	H	I2C to HOST	Address 0x58

In this table L means a logic 0 (<0,8V), H means a logic 1 (>2V), and Z means a high impedance (not connected) pin.

Therefore four configurations are supported for the module $00, connected to the host :

- If SEL1, SEL2 and SEL3 are grounded the module is configured as a USB full speed slave ;
- If SEL1 is not connected then the module is configured as a UART slave. SEL2 and SEL3 allow to set the bit rate and format from the 9 supported configurations ;
- If SEL1 is connected to +3V3 or 5V then the module is configured as an I2C slave. SEL2 and SEL3 allow to define the module I2C address from a pool of 9 addresses.
- If SEL1 and SEL2 are grounded and SEL3 is connected to +3V3 or 5V then the module is configured in bootloader USB mode. This specific mode allows to reflash the on-board firmware through Alciom's supplied tools. Only the first module of the chain could be reflashed in the current version of the protocol, therefore modules should be temporary be inserted in position $00 one per one for reflashing a full system if required.

For chained and stacked modules only one configuration is supported : high speed SPI mode (this mode is indeed identical to the host-SPI mode except that the module doesn't initiate auto-configuration of the module chains.

- If SEL1 is grounded and SEL2 is connected to +3V3 or 5V then the module is configured as an SPI slave and is controlled by another module. SEL3 is then used as a chip select input.

3.3 Host&chain (bottom) connector pin-out

The Pin-out of the host&chain connector (bottom side of modules, also used on the mainboard) is the following, depending on the configuration mode selected through the S1/S2/S3 pins as explained above :

SPI HOST (bottom side)

	Row A	Row B	
1	+5V	GND	2
3	<- DATARDY	**<- MISO/BUSY (HOST)**	4
5	**-> SCK (HOST)**	GND	6
7	-> CS (HOST)	**-> MOSI (HOST)**	8
9	<> HWTRIG	-> SEL1 (GND)	10
11	-> SEL2 (+5V, Z if host)	**<- MOSI (CHAIN)**	12
13	**-> MISO/BUSY (CHAIN)**	<- STATUS	14
15	<- CS (CHAIN)	**<- SLK (CHAIN)**	16
17	**-> CLKIN (or NC)**	-> DATARDY (CHAIN)	18
19	+5V	GND	20

Power
Slave SPI port to host
Mode selection inputs
Master SPI to chained modules
Ancillary pins

(in bold : high speed signals)

<- : module output
-> module input

USB HOST (bottom side)

	Row A	Row B	
1	+5V	GND	2
3	<- CONNECTED	**<-> USB-DP**	4
5	-> +5V SENSE	GND	6
7	-> SEL3 (GND, H= boot!)	**<-> USB-DM**	8
9	<> HWTRIG	-> SEL1 (GND)	10
11	-> SEL2 (GND)	**<- MOSI (CHAIN)**	12
13	**-> MISO/BUSY (CHAIN)**	<- STATUS	14
15	<- CS (CHAIN)	**<- SLK (CHAIN)**	16
17	**-> CLKIN (or NC)**	-> DATARDY (CHAIN)	18
19	+5V	GND	20

Power
Slave USB port to host
Mode selection inputs
Master SPI to chained modules
Ancillary pins

UART HOST (bottom side)

	Row A	Row B	
1	+5V	GND	2
3	-> CTS	**<- TX**	4
5	<- RTS	GND	6
7	-> SEL3 (0, NC or +5V)	**-> RX**	8
9	<> HWTRIG	-> SEL1 (N/C)	10
11	-> SEL2 (0, NC or +5V)	**<- MOSI (CHAIN)**	12
13	**-> MISO/BUSY (CHAIN)**	<- STATUS	14
15	<- CS (CHAIN)	**<- SLK (CHAIN)**	16
17	**-> CLKIN (or NC)**	-> DATARDY (CHAIN)	18
19	+5V	GND	20

Power
UART port to host
Mode selection inputs
Master SPI to chained modules
Ancillary pins

SEL2/SEL3	Mode
L/L	1200bps, 8N1
L/Z	9600bps, 8N1
L/H	19200bps, 8N1
Z/L	38400bps, 8N1
Z/Z	57600bps, 8N1
Z/H	115200bps, 8N1
H/L	230400bps, 8N1
H/Z	19200bps, 8E1
H/H	230400bps, 8E1

I2C HOST (bottom side)

	Row A	Row B	
1	+5V	GND	2
3	<- DATARDY	**<-> SDA**	4
5	<- BUSY	GND	6
7	-> SEL3 (0, NC or +5V)	**<-> SCL**	8
9	<> HWTRIG	-> SEL1 (+5V)	10
11	-> SEL2 (0, NC or +5V)	**<- MOSI (CHAIN)**	12
13	**-> MISO/BUSY (CHAIN)**	<- STATUS	14
15	<- CS (CHAIN)	**<- SLK (CHAIN)**	16
17	**-> CLKIN (or NC)**	-> DATARDY (CHAIN)	18
19	+5V	GND	20

Power
UART port to host
Mode selection inputs
Master SPI to chained modules
Ancillary pins

SEL2/SEL3	Mode
L/L	Address 0x50
L/Z	Address 0x51
L/H	Address 0x52
Z/L	Address 0x53
Z/Z	Address 0x54
Z/H	Address 0x55
H/L	Address 0x56
H/Z	Address 0x57
H/H	Address 0x58

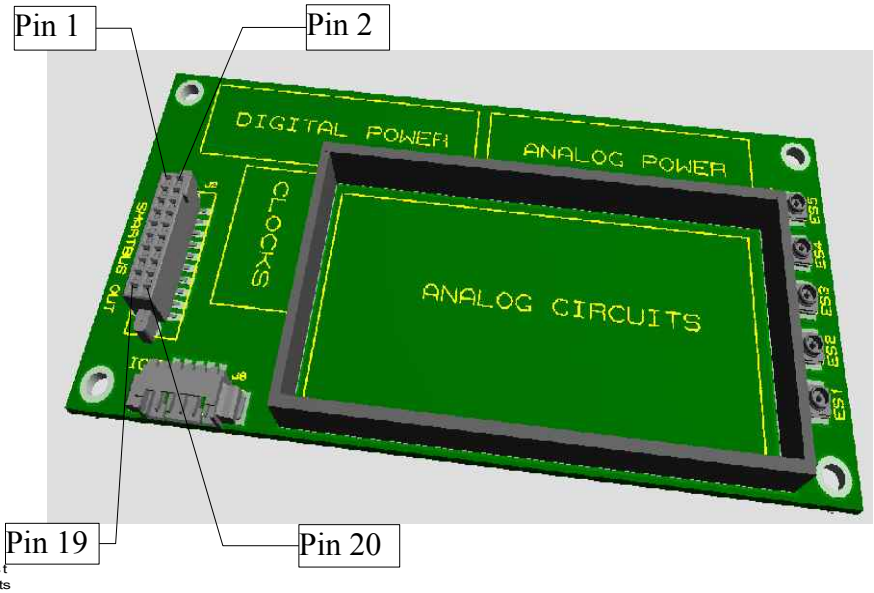
3.4 Stacking (top) connector pin-out

Similarly the pin-out of the stacking connector (top side of modules) is the following. This is subset of the host&chain connector in SPI mode. The illustration on the right precise the location of pins 1 to 20 on the connectors :

STACKING (top side)

	Row A	Row B	
1	+5V	GND	2
3	-> DATARDY	**-> MISO/BUSY**	4
5	**<- SCK**	GND	6
7	<- CS	**<- MOSI**	8
9	<> HWTRIG	<- SEL1 (GND)	10
11	<- SEL2 (+5V)	N/C	12
13	N/C	-> STATUS	14
15	N/C	N/C	16
17	**<- CLKOUT (or NC)**	N/C	18
19	+5V	GND	20

Power
Slave SPI port to host
Mode selection inputs
Master SPI to chained modules
Ancillary pins



3.5 Signals specifications

3.5.1 Generic electrical specifications

The following specifications apply to all SmartBus signals except otherwise stated :

Parameter	Min	Typical	Max	Unit
Input voltage for logic 1	2,7	3,3	6	V
Input voltage for logic 0	-0,3	0	0,5	V
Input leakage current			+/-100	µA
Output low voltage (@5mA)			0,4	V
Outout high voltage (@-2mA)	2,8			V
Current sourced or sunk by any output pin			25	mA

Nota : I/Os are then 3V3 LV-TTL compatible, but also 5V tolerant.

3.5.2 Common pins – Host&chain (bottom) connector

Pin name	Module input or output	Function	Comment
GND	Ground	System ground	
+5V	Power input	Power supply of the module stack	See chapter 4
SEL1	Input	Host MAC/PHY mode configuration, cf 3.2	Each pin should be connected to GND, +3V3/+5V or left unconnected
SEL2	Input		
SEL3	Input		
HWTRIG	Input/output	Provision for inter-modules high precision hardware triggering	Open-collector line, 1kohm pull-up to 3V3 or to 5V must be provided on base board
STATUS	Output	Module status output	1 if module is active
CLKIN	Input	12MHz reference clock input	See chapter 5
CS (CHAIN)	Output	Chaining SPI bus chip select	See chapter 6.1
MOSI (CHAIN)	Output	Chaining SPI master out / slave in	
MISO/BUSY (CHAIN)	Input	Chaining SPI master in / slave out, combined as BUSY input	
SCK (CHAIN)	Output	Chaining SPI clock output	
DATARDY (CHAIN)	Input	Chaining SPI data ready input. Pulled high by a 10Kohm resistor on master side	

3.5.3 SPI mode pins – Host&chain (bottom) connector

Pin name	Module input or output	Function	Comment
CS	Input	Host SPI bus chip select	See chapter 6.1
MOSI	Input	Host SPI master out / slave in	
MISO/BUSY	Output	Host SPI master in / slave out, combined as BUSY output (logic 0 if module if busy and can't accept new commands)	
SCK	Input	Host SPI clock input	
DATARDY	Output	Host SPI data ready output (logic 0 if module has data to transmit).	

3.5.4 USB mode pins – Host&chain (bottom) connector

Pin name	Module input or output	Function	Comment
USB-DP	Input/Output	Host USB bus	See chapter 6.2
USB-DM	Input/Output	Host USB bus	
CONNECTED	Output	Level 1 if USB connection active	
+5V-SENSE	Input	Level 1 if host is powering the bus	

3.5.5 UART mode pins – Host&chain (bottom) connector

Pin name	Module input or output	Function	Comment
TX	Output	Host UART data transmit output	See chapter 6.3
RX	Input	Host UART data receive input	
RTS	Output	Host UART requ. to send (logic 0 if module can accept new commands)	
CTS	Input	Host UART clear to send (logic 0 if host can accept new responses)	

3.5.6 I2C mode pins – Host&chain (bottom) connector

Pin name	Module input or output	Function	Comment
SDA	Input/Output	Host I2C bus (ext pullup needed)	See chapter 6.4
SCL	Input/Output	Host I2C bus (ext pullup needed)	
BUSY	Output	Host I2C requ. to send (logic 0 if module is busy and can't accept new commands)	
DATARDY	Output	Host I2C data ready output (logic 0 if module has data to transmit)	

3.5.7 Stack (top) connector

Pin name	Module input or output	Function	Comment
GND	Ground	System ground	
+5V	Power input	Power supply of the module stack	See chapter 4
SEL1 (GND)	Output	Host MAC/PHY mode configuration, configuring the stacked module in SPI mode, cf 3.2	
SEL2 (GND)	Output		
HWTRIG	Input/output	Provision for inter-modules high precision hardware triggering	Open-collector line, 1kohm pull-up to 3V3 or to 5V must be provided on base board
STATUS	Input	Module status output of the stacked module	1 if module is active
CLKOUT	Output	12MHz reference clock output	See chapter 5
CS	Output	Stacking SPI bus chip select	See chapter 6.1
MOSI	Output	Stacking SPI master out / slave in	
MISO/BUSY	Input	Stacking SPI master in / slave out, combined as BUSY input	
SCK	Output	Stacking SPI clock output	
DATARDY	Input	Stacking SPI data ready input. Pulled high by a 10Kohm resistor on master side	

4 Power and trigger distribution

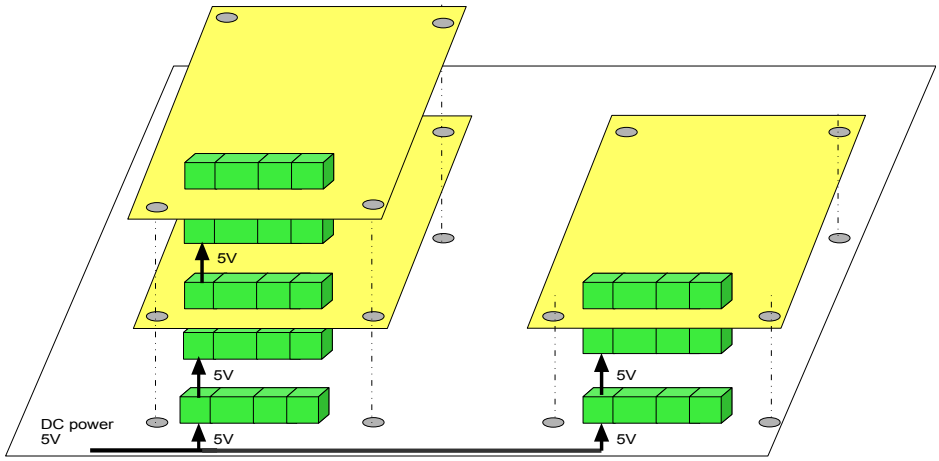
4.1 Modules power specification

Each SmartBrick module is powered by a single 5V DC power supply source. All other required voltage are generated by the module itself through on-board converters as required. Module DC specifications are the following :

Parameter	Min	Typical	Max	Unit
Module supply voltage	4,75	5	5,25	V
Module supply current			200	mA

4.2 Power distribution scheme

DC power source is provided by the motherboard through the host&chain connector, and routed transparently from bottom up through each stack of modules. The motherboard must of course insure that the available current exceeds Nx200mA, N being the total number of SmartBrick modules :



4.3 Hardware trigger distribution

A dedicated open-collector hardware trigger pin (HWTRIG) is included on every SmartBus connectors. These pins are automatically interconnected between stacked modules, and must be interconnected between stacks through the main board. The main board must also includes a 1Kohm pull-up resistor between the HWTRIG line and either +3V3 or +5V power lines.

The usage of the HWTRIG signal is module-specific : Some modules are able to generate a trigger event on HWTRIG and/or to be triggered when the line state changes. This allows high precision inter-modules synchronisation and triggering. UFL connectors provides access to the HWTRIG signal on SmartBrick Solo and XL standard baseboards.

5 Clock distribution

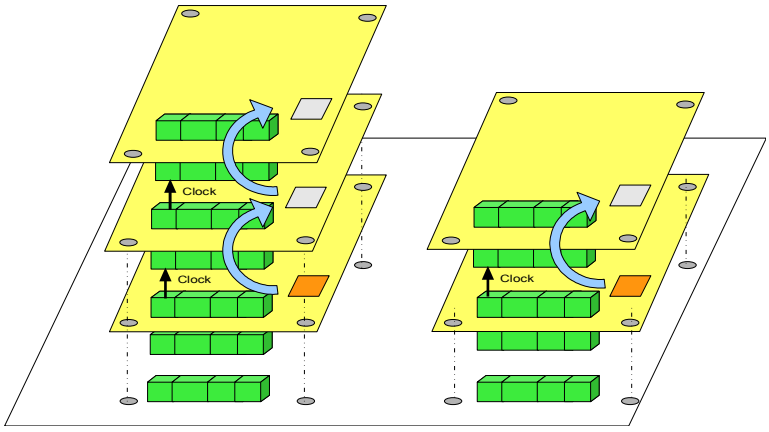
5.1 On-board clocks

The reference clock for all SmartBrick is a 12MHz clock. Each module includes its own 12MHz reference oscillator, which is either a standard +/-50ppm quartz oscillator or a high performance +/-2.5ppm TCXO oscillator depending on the module's requirements. See module specifications for details.

5.2 Clock distribution scheme

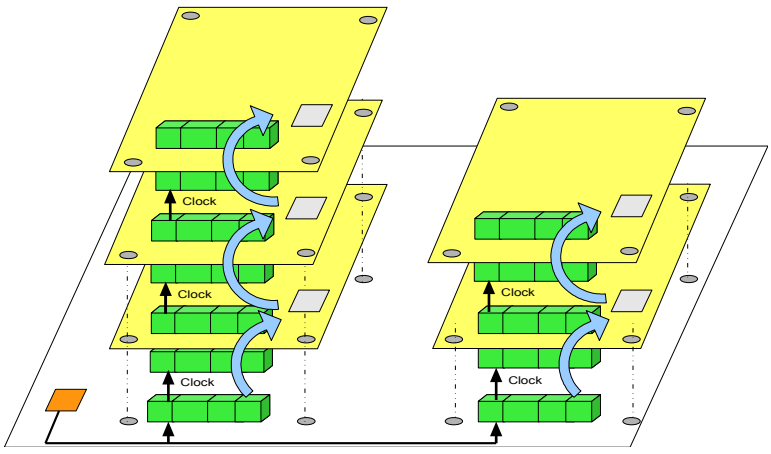
In order to offer the best possible performance and to support frequency-synchronous applications the SmartBus interface includes an automatic clock distribution system : Each module has, in addition to its own reference clock, an external 12MHz clock input on the bottom host&chain connector and a clock output on the top stacking connector. An internal clock selection circuitry automatically selects the external input if there is a clock present on this line.

This implies that for each stack of modules, the clock of the bottom module is automatically used as a reference clock for all modules of the stack. If some modules have high performance TCXO clock it is therefore important to position such a module in the bottom position of the stack.



5.3 External clock support

For applications where more stringent clock synchronisation and accuracy are required, it is also possible to include a high precision 12MHz reference clock on the motherboard and to distribute it to all stacks of modules. The on-board oscillators are then disabled and the system is fully synchronous. This is proposed for exemple on the standard SmartBrick XL enclosure.

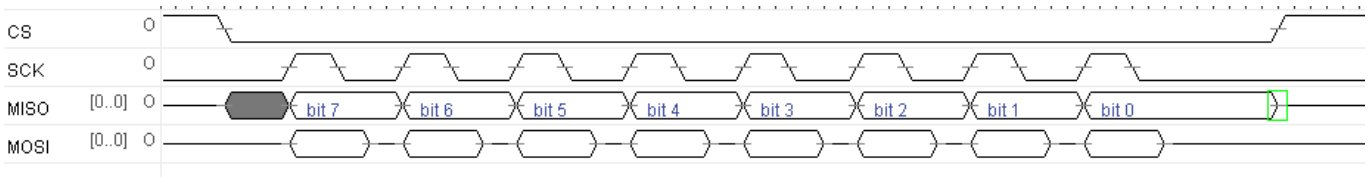


6 MAC/PHY layers

6.1 SPI MAC/PHY layer

6.1.1 Bit-level data transfer

When the MAC/PHY layer is configured in SPI mode the data transfer to/from a SmartBrick module is done using a four-wires LV-TTL synchronous serial interface. A transmission is initiated by the host through a low level on the CS signal, and then successive bytes are transferred through the MOSI and MISO lines under control of the host SCK generated bit clock. Bytes are transferred MSB first, data are sampled on the falling edge of SCK. Clock speed must be below 16MHz :



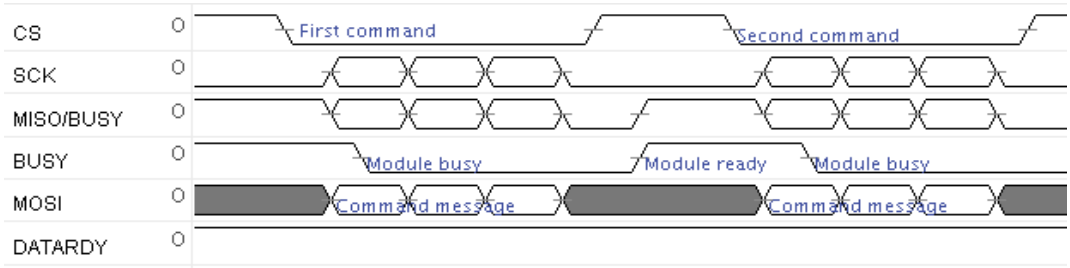
6.1.2 Flow control

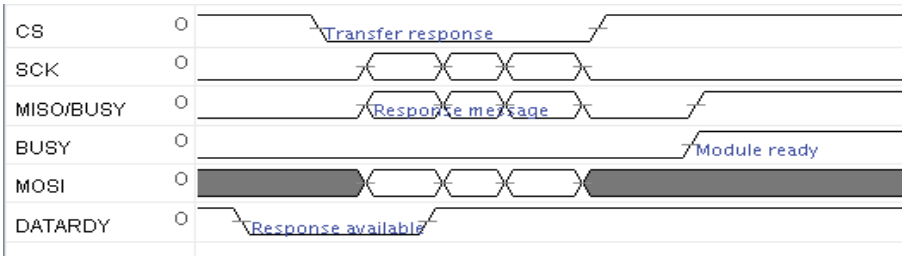
In addition to these data transfer lines the SPI mode interface uses two other flow control lines :

- DATARDY : This output indicates when the module has a response to be transmitted to the host. When this line is LOW the host must initiate a SPI transfer cycle to read the response before transferring any new command to the module. A logic low on DATARDY indicates that a response must be read, a logic high indicates that no response is waiting to be read. Data sent back through SPI by the module is a repeated 00 if DATARDY is low.

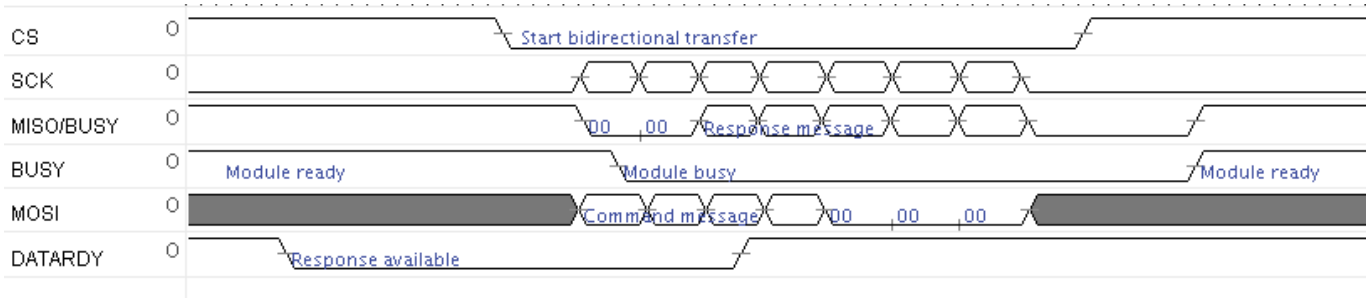
- BUSY : This output indicates when the module is busy and can't accept any new command. A logic LOW on busy means that the host must not send any new command, a logic high indicates that the module is ready to accept a new command through SPI. Commands sent through SPI are ignored by the module if BUSY is low. The BUSY line is an open-collector line, shared by the different SPI slaves present on the same bus (through a 10Kohm pull-up on master side and commuted 1Kohm pull-down on slaves)..

The BUSY line is SHARED with the MISO line, meaning on the same pin of the SmartBus connector. The MISO/BUSY line is in BUSY mode when the SPI port is not active (CS high), and in MISO mode when the SPI port is active (CS low). The following diagram illustrate the possible operations (diagrams generated with timingtool.com) :





The SPI MAC/PHY layer also allows simultaneous bi-directional transfers : If the host starts to send a command when a module has a response ready AND when the module is ready to accept a new command then both transfers can be done simultaneously, on the host's initiative. In such cases the host must insure that the SPI transaction is long enough to allow the module to send its full answer, using stuffing null bytes as needed. Similarly the module can send one or some stuffing null bytes before the start of its actual message, as illustrated below :



If the host stops the transfer by toggling CS high before the full transfer of a command or response then the module will discard the message.

6.1.3 MAC/PHY frame format

The SPI physical frame format is the following :

Stuffing	Byte count MSB	Bytes count LSB	SB-LINK message	Stuffing
Optional : any number of null bytes	1111	Number of bytes of the layer 3 message (5 to 2053, including 5 bytes for address & message code + up to 2048 data bytes, cf chapter 7)	SB-LINK standardized message (cf chapter7)	Optional : any number of null bytes

In SPI mode messages are always transferred in raw binary mode.

Exemple : $00 $00 $F0 $03 $11 $22 $33 $00 $00 $00 (SB_LINK message is $11 $22 $33)

6.2 USB MAC/PHY layer

6.2.1 Bit-level data transfer

When a SmartBrick module is configure in USB mode the module emulates a USB CDC slave device, and is recognized automatically as a virtual COM port by the host. Data transfers are done according to the USB 2.0 full speed specifications.

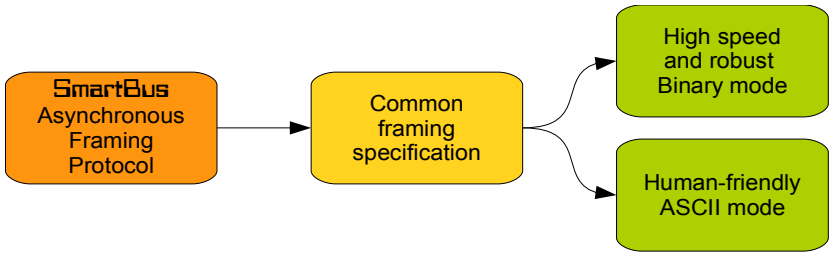
6.2.2 Flow control

Managed by the USB protocol. Exchanges are bidirectional and full duplex, the host can send at any time a command through the USB link (that will be blocking if the module is not ready to receive any more commands), and the module can send back responses through the same USB connection at any time.

6.2.3 MAC/PHY frame format : The SmartBus asynchronous framing protocol (SAFP)

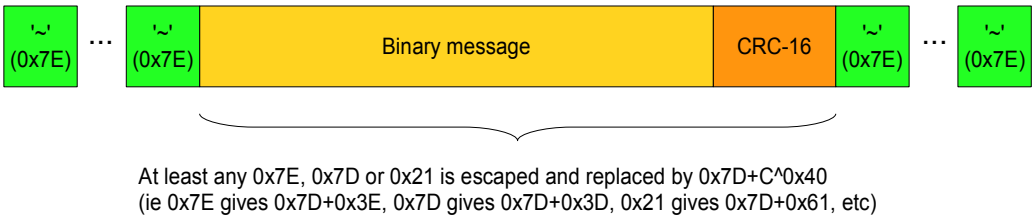
In order to be as flexible as possible the USB MAC/PHY layer implements a very flexible protocol, the SmartBus asynchronous framing protocol (SAFP). SAFP transparently supports two transfer modes on the module side, and host-side softwares can use either mode or both :

- An efficient binary-coded mode with robust framing and error detection through CRC checking ;
- A human-friendly ASCII mode, which allows easy control and interactive use of the modules through any virtual terminal program like Teraterm Pro.



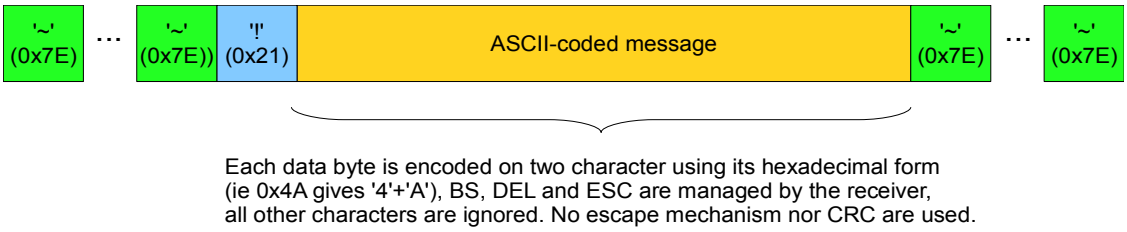
Binary-coded mode :

- The character '~' (0x7E) is defined as the SAFP flag character. If indicated that the serial asynchronous link is alive. When idle, either nodes can transmit this flag character if desired
- The first non-flag character (any byte different from 0x7E) determines the beginning of a message block
- The end of a message block is determined by a flag character (0x7E)
- Inside the message any appearance of the flag character must be escaped. The escape character is defined to be 0x7D. The escaping algorithm is the following : if payload character XX is to be escaped, it is replaced in the transmit stream by the two-character sequence 0x7D XX^0x40, that is: 0x7D is sent, followed by the payload character with bit 0x40 toggled (XORed with 0x40).
- At least the following characters must be escaped: 0x7E, 0x7D, 0x21. The transmitter may freely choose to escape any other character.
- For error detection, a 2-character CRC over un-escaped payload data is appended to the “binary” frame. The CRC is calculated following CRC-16 CCIT algorithm (see annex), transmitted MSB first, and is also subject to escaping.



Human-friendly ASCII mode :

- The character '~' (0x7E) is defined as the SAFP flag character as well in human-friendly mode. If indicated that the serial asynchronous link is alive. When idle, either nodes can transmit this flag character if desired
- A human-friendly frame is recognized when the character '!' (0x21) is immediately following a '~' flag character.
- Every binary payload character is then transmitted as 2 characters, these two characters being the ASCII representation of the two hexadecimal nibbles of the binary payload character. ASCII characters allowed are '0'..'9', 'A'..'F', 'a'..'f'. For example if the binary payload character 0xF4 is to be transmitted, it will be transmitted as the two-character sequence 'F4' or 'f4', that is, 0x46 0x34 or 0x66 0x34.
- The characters 0x08 (BS, backspace) and 0x7F (DEL, delete) indicate to the friendly receiver that the previous character is to be discarded.
- The character 0x1D (ESC, escape) indicates to the friendly receiver that the full frame is to be discarded (aborted) – the receiver should wait for a new frame, starting with a flag.
- Any other character received (ie different from '0'..'9', 'A'..'F', 'a'..'f', BS, DEL, ESC) is accepted and discarded.
- A human-friendly mode frame is ended by a '~' flag character. No CRC is added to the frame



Mode switching rules :

- All SmartBrick module are able to receive messages either in ASCII mode or binary mode.
- Hosts can send their commands either using ASCII mode messages or binary mode messages or both.
- Whenever a module (using asynchronous mode, which imply connected to the host) receives a message in binary mode the it will send all following answers in binary mode, and vice-versa.

These rules allows a very simple and flexible mechanism on the host side :

- For simple systems (using only one host client) then the host can use either full ASCII or full binary or a mix of both modes for its commands, and will always receive an answer using the same coding mode.
- For system having multiple host clients then all hosts client must either use all the same coding mode, or must all support both coding modes for the received messages (in particular when Indications are used).

Example 1 – Sending of SBLINK message: 0x12 0x34 0x56

Transmit in binary format: CRC-16 calculation gives 0xDE 0x61:

0x7E **0x12 0x34 0x56** 0xDE 0x61 0x7E

Transmit in friendly format:

0x7E 0x21 **0x31 0x32 0x33 0x34 0x35 0x36** 0x7E, that is ~!123456~

White spaces and other formatting characters may be added.
One of the many possible alternate valid friendly forms is:

0x7E 0x21 0x20 **0x31 0x32 0x33** 0x0D 0x0A **0x34 0x35** 0x20 **0x36** 0x7E 0x0D 0x0A ,

that is
~!1 23
45 6~

Example 2 – Sending of SBLINK message: 0x21 0x12 0x7D 0x34 0x7E 0x56

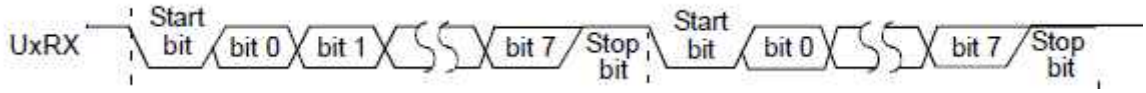
Transmit in binary format: CRC-16 calculation gives is 0x43 0x82

0x7E 0x7D **0x61 0x12** 0x7D **0x3D 0x34** 0x7D **0x3E 0x56** 0x43 0x82 0x7E

6.3 UART MAC/PHY layer

6.3.1 Bit-level data transfer

When a SmartBrick module is configure in UART mode the data transfer is done asynchronously and in full duplex on RX and TX signal lines. After one start bit the eight successive bits are transmitted LSB first, and ended by stop bit preceded by an option even parity bit (cf 3.2) :



6.3.2 Flow control

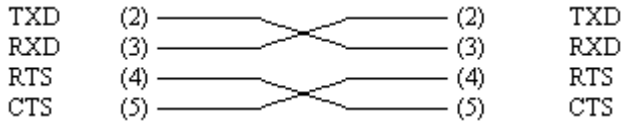
In UART mode the SmartBus system uses CTS/RTS handshaking :

- CTS is an input of the module, and allows the host to indicate if it can receive any new data characters. If CTS = 1, then the module will not send data bytes.

- RTS is an output of the module, and allows the module to indicate to the host if it can receive any new characters. If RTS=1 then the host should stop to send data immediately.

These lines are asynchronously managed, meaning that CTS or RTS can change state in the middle of a command or answer, the transfer will be stopped immediately and will be resumed when these flow control lines will allow it.

RTS/CTS and RX/TX pins must be crossed between the module and the host for a propel full duplex data exchange :



6.3.3 MAC/PHY frame format

In order to be as flexible as possible the UART MAC/PHY layer supports exactly the same SmartBus Asynchronous framing protocol (SAFP) than the USB mode, including :

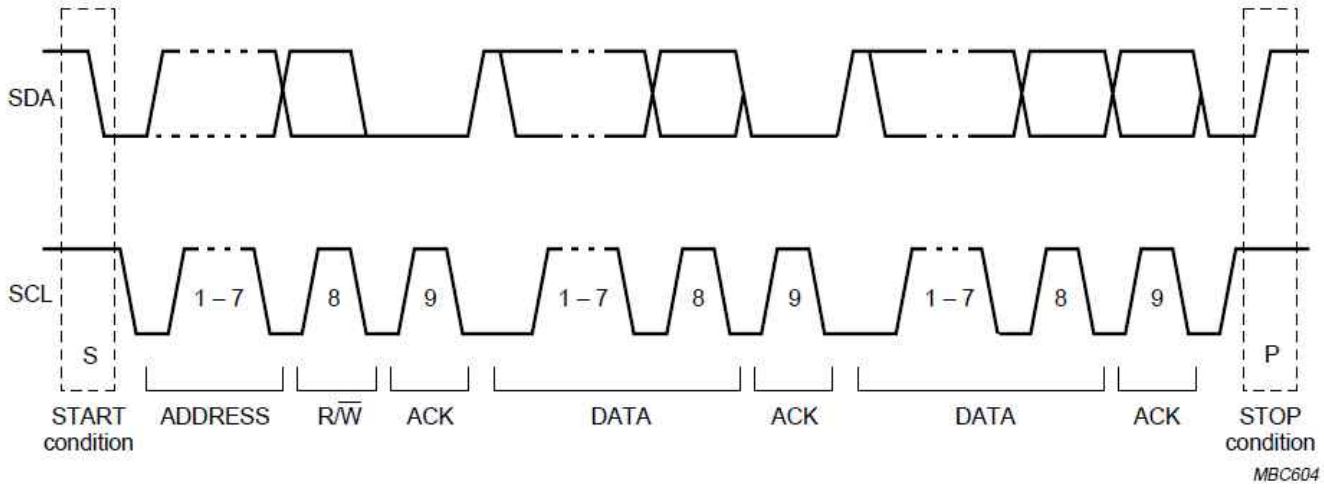
- An efficient binary-coded mode with robust framing and error detection through CRC checking ;
- A human-friendly ASCII mode, which allows easy control and interactive use of the modules through any virtual terminal program like Teraterm Pro.

See 6.2.3 for message framing specifications.

6.4 I2C MAC/PHY layer

6.4.1 Bit-level data transfer

When the MAC/PHY layer is configured in I2C mode the data transfer to/from a SmartBrick module is done using a two-wires bidirectional I2C interface with clock speed up to 400Kbps, the module being an I2C slave and the host an I2C master. A transmission is initiated by the host through a START condition on the I2C bus, then an address and R/W flag is transferred followed by successive data bytes (sent by the host for a write operation or by the slave for a read operation) and ended by a bus STOP condition. Bytes are transferred MSB first, and each byte is acknowledged by the receiver :



The module ignore any I2C transaction that doesn't match its preconfigured I2C address ($50 to $58, cf 3.2).

6.4.2 Flow control

In addition to these data transfer lines the I2C mode interface uses two other flow control lines :

- DATARDY : This output indicates when the module has a response to be transmitted to the host. When this line is LOW the host must initiate a I2C READ transfer cycle to read the response before transferring any new command to the module. A logic low on DATARDY indicates that a response must be read, a logic high indicates that no response is waiting to be read.

- BUSY : This output indicates when the module can't accept any new command. A logic LOW on BUSY means that the module is busy and can't accept any new I2C WRITE command, a logic high indicates that the module is ready to accept a new command through I2C. Commands sent through I2C are ignored by the module if BUSY is low.

6.4.3 PHY frame format

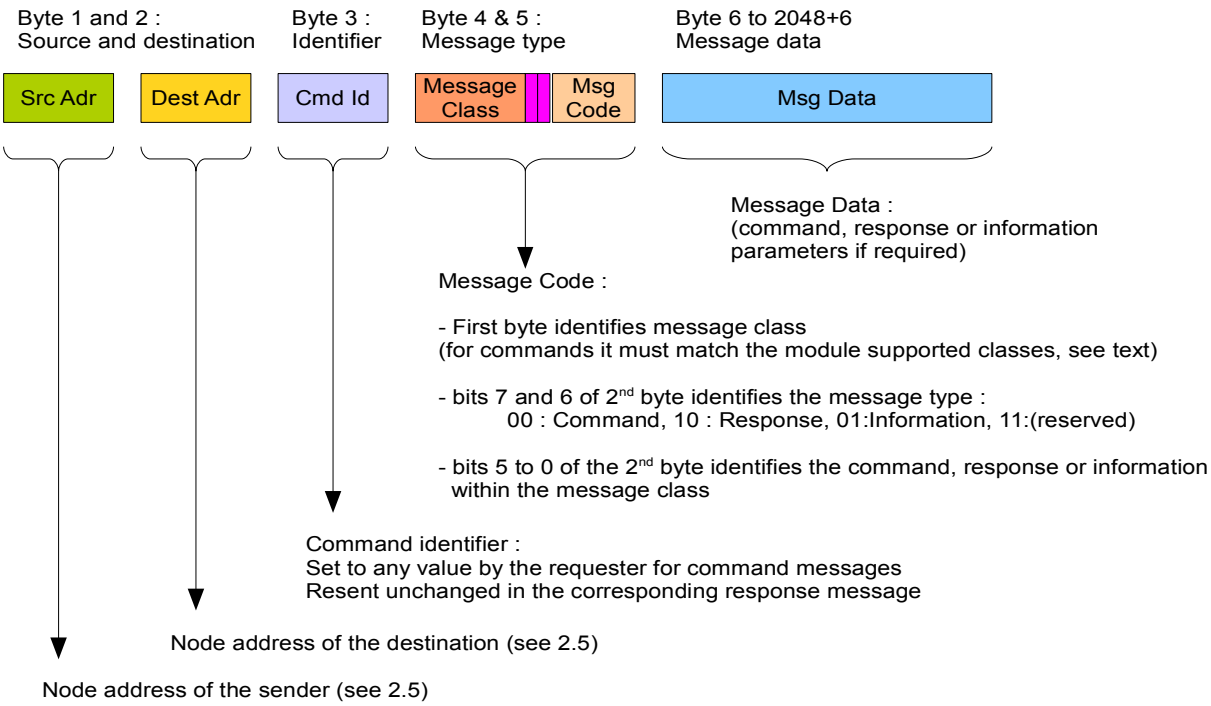
The I2C physical frame format is the following :

Header	I2C adress byte	SB-LINK message	Footer
I2C START condition	I2C module address + R/W bit	SB-LINK standardized message (cf chapter7), 5 to 2053 bytes long (5 bytes for address & message code + up to 2048 data bytes, cf chapter 7), in raw binary mode	I2C STOP condition

Exemple : START $A0 $11 $22 $33 STOP (SB_LINK message is $11 $22 $33, I2C slave adress is $50, write mode)

7 SB-LINK standardized message format

Independently from the low level MAC/PHY protocol used, and independently from the SmartBrick module used, all products uses the same standardized message format through the system, from the host client to the applicative firmware inside the module. This standardized format, named SB_LINK format, is the following :



The field **CommandIdentifier** (byte 3) allows the requester to send several commands and to match the corresponding reponses. It is initialised to any value by the requester (hosts should however avoid value 0x00), sent within the command, and resent unchanged in the corresponding response. For Indication messages this field is either set to the identifier of the currently executing command (if the indication should be correlated to a given command) or set to 0x00 otherwise.

The field **MessageClass** (byte 4) indicates the class for any command, associated response and indications. It must match with one of the classes supported by the target node.

A node receiving a command with an unsupported class will send back an error message.
A node receiving a response which doesn't match with one the commands its has sent will ignore the message.

The following classes are pre-defined :

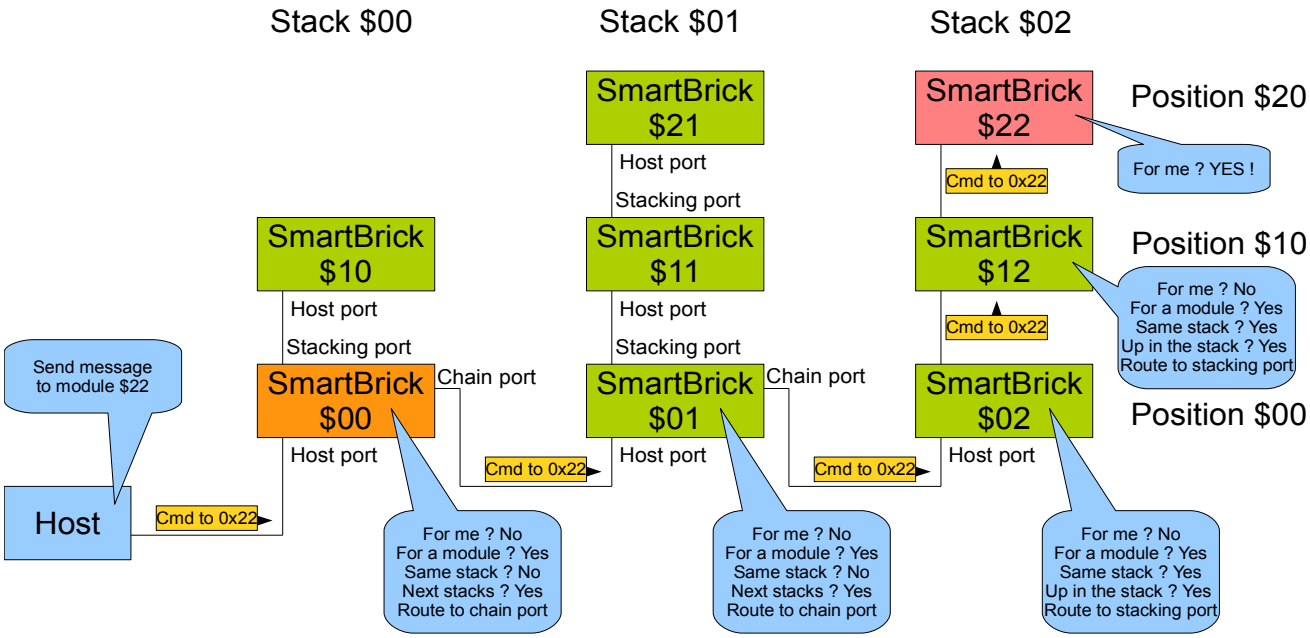
Message Class	Description
0x00	Generic module message class. All modules must understand and manage all commands within this class and should be able to send the corresponding responses and indication messages. This class is specified in chapter 9
0x01 to 0x0F	Reserved for future extensions
0x10 to 0xEF	Module-specific classes. One class correspond to a given behaviour and command set, and includes modules with the same semantics (for example one class maps to all basic ADC, etc)
0xF0 to 0xFF	Reserved for future commands directed to host clients

8 SB-NET routing protocol

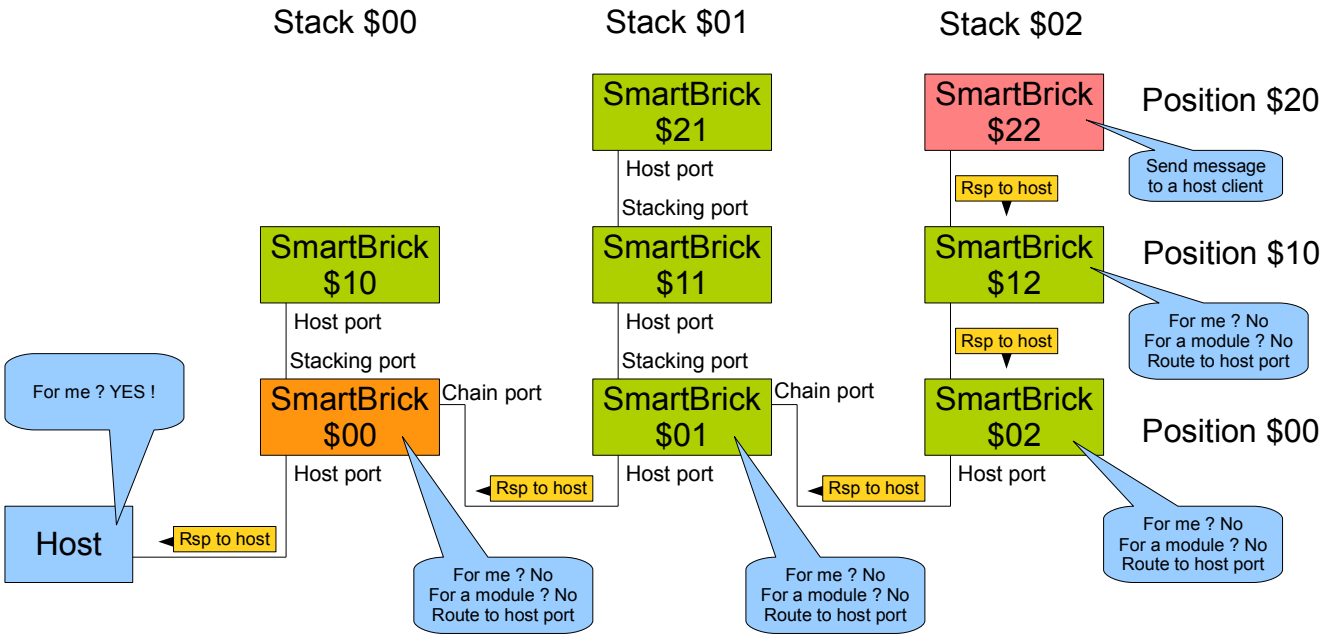
8.1 Routing mechanism

The SB-NET routing protocol insures a delivery of command messages to the proper module based on the module address and system topology, fully transparently to the user.

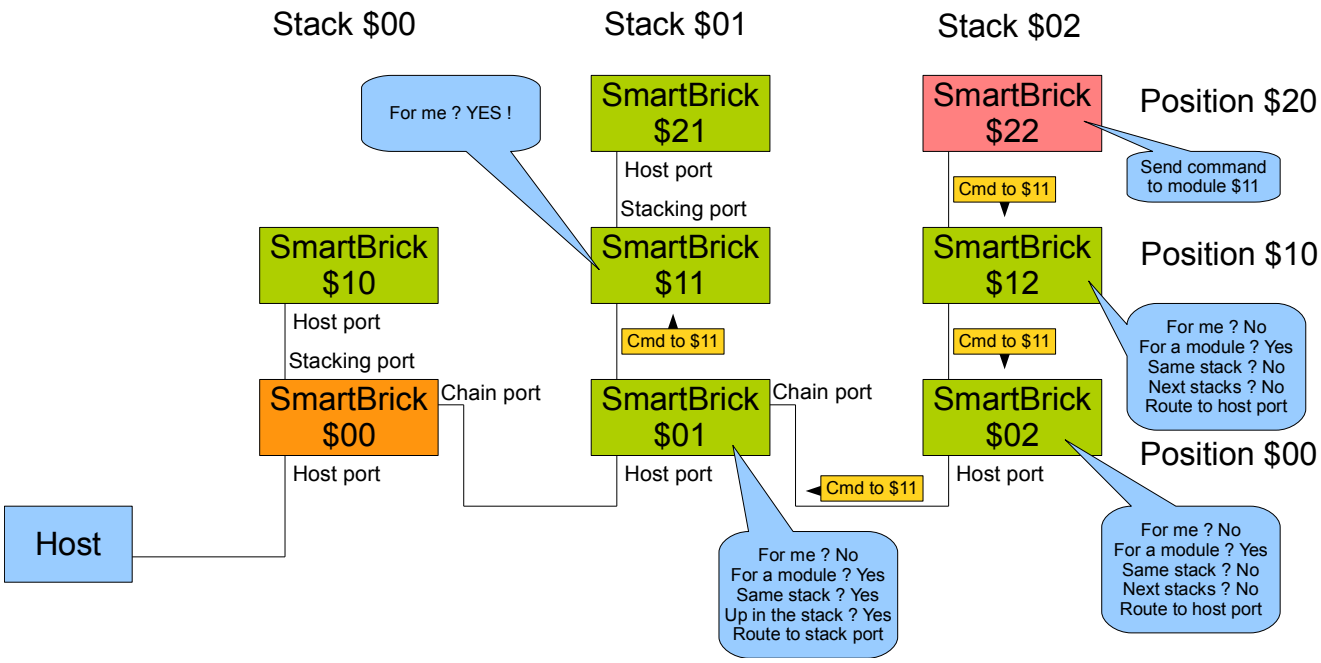
In the case of a host-to-module message the decision tree is the following :



Messages to a host client, as well as messages broadcasted to all host clients, are simply always forwarded to the host port of every module up to the host :



The same mechanism, with minor variants, is also used for module to module routing :



8.2 Network initialisation

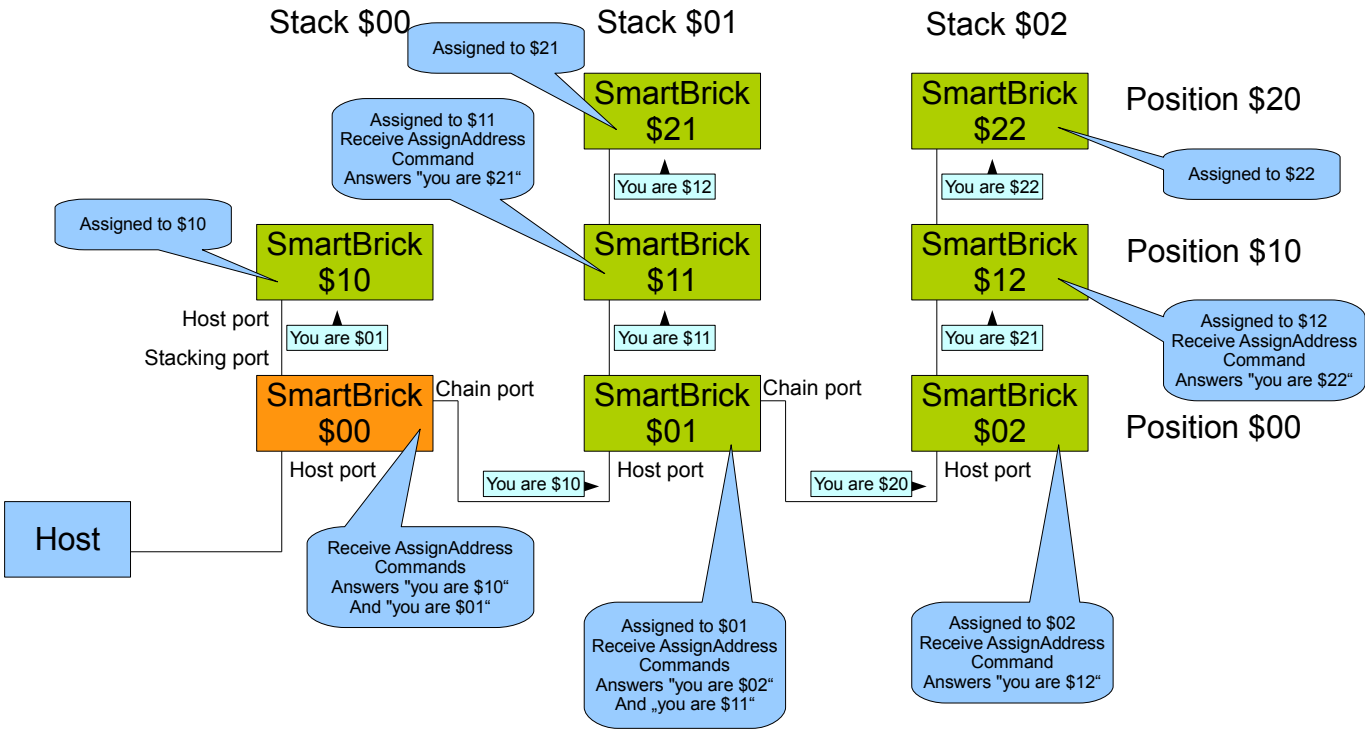
At startup, the module $00, which is the module connected to the host, **identifies itself through the plug'n play configuration**, as the selection pin configuration for a module-to-module link is distinguishable for module-to-host links at the electrical level (cf 3,2). This module then assigns itself the address $00.

All other modules starts in a special “no address affected” mode. In this mode modules don't route messages nor read messages from their stacking and chaining ports. **Each node then sends a specific Assign-Address command (cf 9.1.1) to their host port**, using the following specific settings (cf chapter 7) :

- Source address = 1100 0000 (special address : Message from a node with an unassigned address)
- Destination address address = 1100 0001 (Message to the receiving node, whoever it is)

In order to send this message the modules raise their DATAAVAIL pin and waits for the upstream module to be ready and to read, process and answer to this message. The node then gets its address and switch to normal operating and routing mode. This process is automatically repeated through the full network, transparently for the host and user.

This address configuration occurs before the management of the first message coming from the host.



The same process is used if a module gets resetted for any reason : It ask again an address from its upstream node. The event will be flagged by an error in the next command send by the host and/or by an Indication message is enabledWarning : the host must then take care that messages could have been lost during the reset procedure of the module.

8.3 Dead lock avoidance

The flow control mechanisms used in all SmartBrick MAC/PHY layers avoid any dead lock in the message routing while insuring a minimization of the routing delay through the system.

There is only one important condition that the host software must comply with : The host must never stay blocked waiting to send a new command if the SmartBrick system has a response to send back. Responses should always be managed with a higher priority than new commands. This easy to implement “always out first” rule insures best and predictable performances.

8.4 Message to a non-existing node

Nothing forbid a node to send a message to a node using a non existing destination address (non existing module position or non-existing host client address). In that case the SmartBus protocol rules are the following :

- Responses and Information messages sent to a non-existing node are discarded without any other action ;
- Command messages sent to a non-existing node are discarded however a corresponding Response message must be sent back to the requester, indicating that there was an error (cf Not-Existing-Address Response, 9.4). This message is sent by the node who discover the error, which means by the last module which gets the message and can't route it farer, or by the host-side software which receive and dispatch the messages. In this specific case the sender address of the message is set to the address of the requested but not existing module, not the address of the effectively responding module.

9 SB-APP generic class (class 0)

9.1 Generic mandatory commands

All SmartBrick modules must support the following commands and generates the associated responses. This baseline, the SB-APP Generic Class, or Class-0, allows auto-discovery and basic management of any SmartBrick system.

9.1.1 Assign-Address Command

Description : Ask the receiving module to provide an address to the sending node. This message is always sent from the special address 1100 0000 (message from a node without an assigned address) to the special address 1100 0001 (message to the receiving node, whoever it is), and the associated response is sent from a node with a standard address to a node identified with the special address 1100 0001 also (message to the receiving node, whoever it is). Cf 8.2.

Command Format :

Message class	Message code	Message data
0x00	0x00	None

Response format (module can also send back an Error Response, cf 9.2):

Message class	Message code	Message data	
0x00	0x00	Error code=0x00	(in case of error cf 9,2,1)
		Address	1 byte, 0x01 to 0xff

9.1.2 Get-Identification Command

Description : Ask a module to identify itself

Command Format :

Message class	Message code	Message data
0x00	0x01	None

Response format (module can also send back an Error Response, cf 9.2):

Message class	Message code	Message data	
0x00	0x01	Error code=0x00	(in case of error cf 9,2,1)
		Protocol_Version	Version of the SmartBus protocol (1 byte, 0x01)
		Mod_Model_MSB Mod_Model_LSB	Identification code of the module (2 bytes)
		Mod_Version	Version of the module (1 byte)
		Num_Classes Mod_Classes	Number of supported classes (1 byte) Classes supported by the module (1 to 256 bytes)
		ExtDescr	Extended description, module dependent . This field must start by the ASCII name of the module,

		ended by a 0 (0 to 256 bytes)

9.1.3 Module-ping Command

Description : Ask a module to send back the same frame (for performance measurement and debugging)

Command Format :

Message class	Message code	Message data	
0x00	0x02	Data	0 to 2047 bytes

Response format (module can also send back an Error Response, cf 9.2):

Message class	Message code	Message data	
0x00	0x02	Error code=0x00	(in case of error cf 9,2,1)
		Data	Same content than command (0 to 2047 bytes)

9.1.4 Get-Status Command

Description : Ask the current status of a module

Command Format :

Message class	Message code	Message data
0x00	0x03	None

Response format (module can also send back an Error Response, cf 9.2):

Message class	Message code	Message data	
0x00	0x03	Error code=0x00	(in case of error cf 9,2,1)
		Status	Status information (1 byte) : bit 0 : 1 if module busy, 0 if not bit 1 : 1 if module correctly configured, 0 if not bit 2 : 1 if module armed, 0 if not bit 3 : 1 if module triggered, 0 if not bit 4 : module specific bit 5 : module specific bit 6 : module specific bit 7 : 1 if module in error, 0 if not
		Module in error code	if module in error: code of this error
		Additional bytes : module-specific data, cf module documentation	

9.1.5 Module-reset Command

Description : Ask a module to reset itself

Command Format :

Message class	Message code	Message data
0x00	0x04	Reset type: 0x00 : module reset, other : general reset reset

After a module reset, the module send ping command to know if it have slaves.
A general reset, reset the module and after it get his address, send reset command to its slave.

Response format (module can also send back an Error Response, cf 9.2):

Message class	Message code	Message data
0x00	0x04	Error code=0x00 (in case of error cf 9,2,1)

Nota : after reseting the module will re-ask its parent module for an address and will then send a “critical error-module resetted” when receiving the next command from the host, see 9.4.

9.1.6 Enable-Indications Command

Description : Enable the sending of spontaneous indications from the target module.

Command Format :

Message class	Message code	Message data
0x00	0x05	Indication class bit mask

Response format (module can also send back an Error Response, cf 9.2):

Message class	Message code	Message data
0x00	0x05	Error code=0x00 (in case of error cf 9,2,1)

After receiving this command the module will broadcast to all host clients all Indication messages which have a matching class, meaning for which the logical AND of their Indication class with the command bit mask is not 0.

9.2 Non specific responses

9.2.1 Error Response

Sent back by a node when he can't execute the requested command (command for an unsupported class, for an unsupported command code, with a badly formatted command, etc)

Response format :

Message class	Message code	Message data
Identifical to the class of the offending command	Identifical to the class of the offending command	Error code, not null (cf 9.4) Error context (application dependant, 0 to 256 bytes)

9.2.2 Non-existant-address Response

When a node receive a message but can't route it farer because the requested destination address is not existing it discards the message but send back the following response to the requester (cf 8.4).

Response format :

Message class	Message code	Message data
Identifical to the class of the offending command	Identifical to the class of the offending command	Error code = 0x01 (cf 9.4) Address of the last module found in the chain (1 byte)

Nota : In this specific case the sender address of the message is set to the address of the requested but not existing module, not the address of the effectively responding module.

9.3 Generic mandatory Indication messages

All SmartBrick modules must be able to receive and transmit the following Indication messages. These messages are sent only when enabled by the Indication-Enable command (9.1.6).

9.3.1 Out-of-command-error Indication

Description : Indicates that a given module has flagged an error in its processing, independently from the execution of a given command.

Indication message Format :

Message class	Message code	Message data	
0x00	0xFF	Error code	Application dependant, not nul
		Criticity	0x00 if minor, for information only 0x40 if commands execution could be impacted 0x80 if messages could be lost 0xFF if module is in an unkown state Other : reserved
		Error context	0 to 256 additional bytes, application dependant

Indication class = 0x80

9.4 Error codes

The following error codes could be returned as part of Class 0 responses :

Error code	Description	Error context field	Comment
0x01	No module at this address	Byte 1 : Address of the last module found in the tree is sent after the error code	
0x02	Unsupported message type	None	
0x03	Unsupported command class	None	Command class doesn't match with the module class and is not the generic 0x00 class
0x04	Unsupported command code	None	This specific command is not supported by the module for this requested class
0x05	Wrong command length	Byte 1 MSB of the command length Byte 2 : LSB of the command length	
0x06	Illegal parameter value in command	Byte 1 : Position of the offending parameter	
0x07	Illegal command in that context	Module dependent	
0x09	Message too long		When the received message is longer than the buffer size
0x0A	Transmission ended before complete reception		
0x0B	CRC error		
0x0C to 0x1E	Reserved		
0x1E	Critical error, module resetted	Module dependant	A critical error has happened either during the execution of the command or before receiving the command. The node has resetted itself and will access next commands however more messages may have been lost
0x20 to 0xFF	Class specific	Module dependent	

10 Annex : CRC-16 algorithm

The CRC-16 added by the SmartBus protocol to any USB and UART frames in binary mode is calculated using the common CRC-CCIT algorithm (XModem), using the following polynom :

$$x^{16} + x^{12}+x^5+1 \ (0x1021)$$

Exemple of compatible source code :

```
/* CRC16 implementation acording to CCITT standards */

static const unsigned short crc16tab[256]= {
        0x0000,0x1021,0x2042,0x3063,0x4084,0x50a5,0x60c6,0x70e7,
        0x8108,0x9129,0xa14a,0xb16b,0xc18c,0xd1ad,0xe1ce,0xf1ef,
        0x1231,0x0210,0x3273,0x2252,0x52b5,0x4294,0x72f7,0x62d6,
        0x9339,0x8318,0xb37b,0xa35a,0xd3bd,0xc39c,0xf3ff,0xe3de,
        0x2462,0x3443,0x0420,0x1401,0x64e6,0x74c7,0x44a4,0x5485,
        0xa56a,0xb54b,0x8528,0x9509,0xe5ee,0xf5cf,0xc5ac,0xd58d,
        0x3653,0x2672,0x1611,0x0630,0x76d7,0x66f6,0x5695,0x46b4,
        0xb75b,0xa77a,0x9719,0x8738,0xf7df,0xe7fe,0xd79d,0xc7bc,
        0x48c4,0x58e5,0x6886,0x78a7,0x0840,0x1861,0x2802,0x3823,
        0xc9cc,0xd9ed,0xe98e,0xf9af,0x8948,0x9969,0xa90a,0xb92b,
        0x5af5,0x4ad4,0x7ab7,0x6a96,0x1a71,0x0a50,0x3a33,0x2a12,
        0xdbfd,0xcbdc,0xfbbf,0xeb9e,0x9b79,0x8b58,0xbb3b,0xab1a,
        0x6ca6,0x7c87,0x4ce4,0x5cc5,0x2c22,0x3c03,0x0c60,0x1c41,
        0xedae,0xfd8f,0xcdec,0xddcd,0xad2a,0xbd0b,0x8d68,0x9d49,
        0x7e97,0x6eb6,0x5ed5,0x4ef4,0x3e13,0x2e32,0x1e51,0x0e70,
        0xff9f,0xefbe,0xdfdd,0xcffc,0xbf1b,0xaf3a,0x9f59,0x8f78,
        0x9188,0x81a9,0xb1ca,0xa1eb,0xd10c,0xc12d,0xf14e,0xe16f,
        0x1080,0x00a1,0x30c2,0x20e3,0x5004,0x4025,0x7046,0x6067,
        0x83b9,0x9398,0xa3fb,0xb3da,0xc33d,0xd31c,0xe37f,0xf35e,
        0x02b1,0x1290,0x22f3,0x32d2,0x4235,0x5214,0x6277,0x7256,
        0xb5ea,0xa5cb,0x95a8,0x8589,0xf56e,0xe54f,0xd52c,0xc50d,
        0x34e2,0x24c3,0x14a0,0x0481,0x7466,0x6447,0x5424,0x4405,
        0xa7db,0xb7fa,0x8799,0x97b8,0xe75f,0xf77e,0xc71d,0xd73c,
        0x26d3,0x36f2,0x0691,0x16b0,0x6657,0x7676,0x4615,0x5634,
        0xd94c,0xc96d,0xf90e,0xe92f,0x99c8,0x89e9,0xb98a,0xa9ab,
        0x5844,0x4865,0x7806,0x6827,0x18c0,0x08e1,0x3882,0x28a3,
        0xcb7d,0xdb5c,0xeb3f,0xfb1e,0x8bf9,0x9bd8,0xabbb,0xbb9a,
        0x4a75,0x5a54,0x6a37,0x7a16,0x0af1,0x1ad0,0x2ab3,0x3a92,
        0xfd2e,0xed0f,0xdd6c,0xcd4d,0xbdaa,0xad8b,0x9de8,0x8dc9,
        0x7c26,0x6c07,0x5c64,0x4c45,0x3ca2,0x2c83,0x1ce0,0x0cc1,
        0xef1f,0xff3e,0xcf5d,0xdf7c,0xaf9b,0xbfba,0x8fd9,0x9ff8,
        0x6e17,0x7e36,0x4e55,0x5e74,0x2e93,0x3eb2,0x0ed1,0x1ef0
};

unsigned short crc16_ccitt(const void *buf, int len)
{
        register int counter;
        register unsigned short crc = 0;
        for( counter = 0; counter < len; counter++)
                crc = (crc<<8) ^ crc16tab[((crc>>8) ^ *(char *)buf++)&0x00FF];
        return crc;
}
```