ATLAS Internal Note
LARG-No-93
24-03-98

SPAC : Serial Protocol for the Atlas Calorimeter

R.Bernier, D.Breton, P.Cros (LAL ORSAY), A.Gara (NEVIS labs).

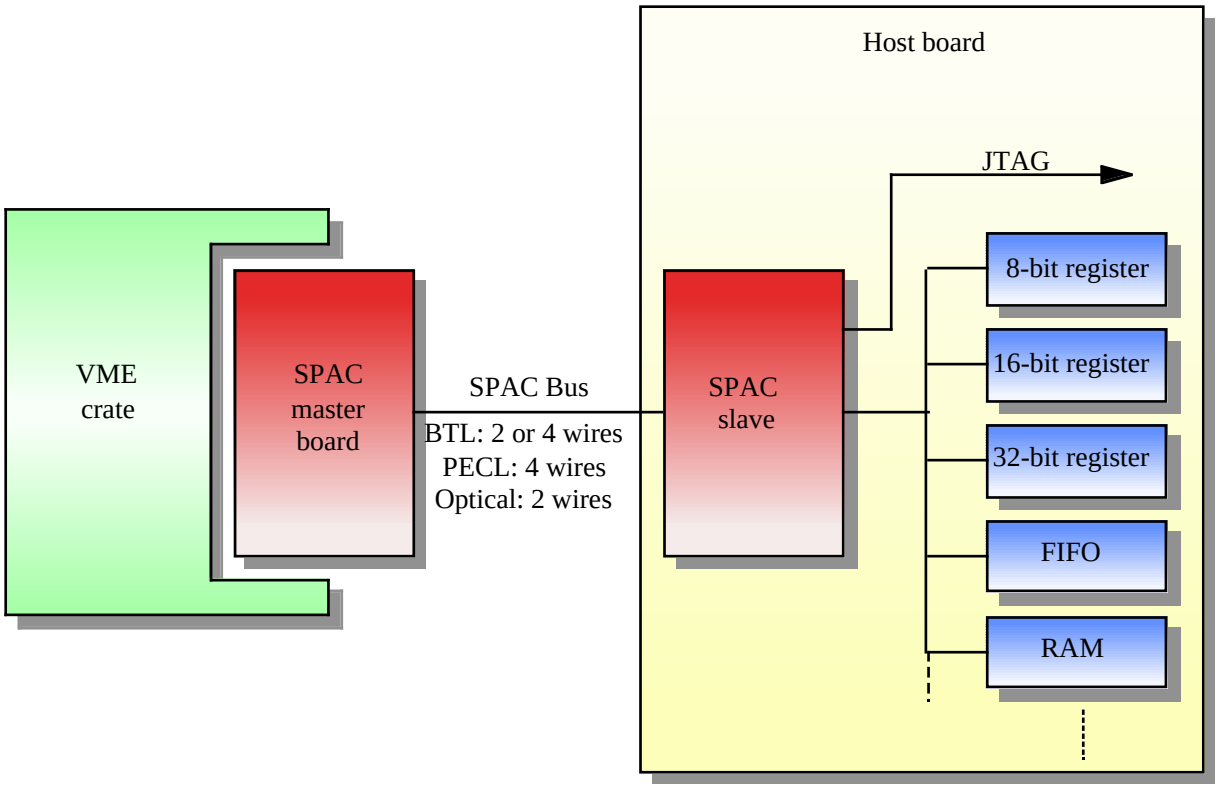
Version 1.0

January 26th 1998

The aim of this protocol is to provide the loading and reading of all registers and memories sitted on the calorimeter of the ATLAS detector. It has been studied to be fast (10Mbit/s), reliable (the error detection rate is high) and cheap. The slave interface fits in an unique circuit and offers several facilities (SPAC -> VME transcoder to drive a VME bus thus allowing crate interconnections, JTAG outputs for on board FPGA programming, ...). The SPAC bus can be PECL/BTL and uni/bidirectionnal. The user's software is written in C, and graphic interfaces are running on UNIX and MacIntosh. The SPAC bus induces very low noise and small power consumption. The protocol is simple and powerful, and allows a immediate understanding of data transfers with an oscilloscope.

As the possibilities of the SPAC bus seem wide, it could also be used for many other applications.

SPAC general view



1. Description

1.1. Main features

The main points of this one-master n-slaves bus are described below. The following definitions allow any kind of transfer, including as many words as desired, in each direction between the master and the slaves. The specific adaptations for custom applications are left to the choice of the users. Nevertheless, some concrete propositions are made concerning the block mode transfers.

Here are the main features of the SPAC bus :

- The protocol requires only two bidirectionnal wires in BTL technology : SCL for clock/strobe, SDA for data. But it can be used identically with four unidirectionnal wires, in BTL or PECL technology. The master and slaves can either be considered as emitters or as receivers on the line.

- There is no problem of master arbitration as this bus is single-mastered by definition. Nevertheless, to prevent the collisions from different slaves, the protocol forbids the broadcast reading command, except after a checksum error and only for reading the slave status register.

- Each slave connected to the bus is addressable by a unique 7 bit address. One address is reserved for the global broadcast mode, which allows the addressing of all of the slaves. Moreover, 15 other addresses are reserved for local broadcast modes, which allow the addressing of various groups of slaves, defined by the users. These groups realise a partition of the totality of the slaves (each slave belongs to one group). The broadcast modes are only available for write commands coming from the master.

- The data always travel in the same direction as the clock. Data is transfered at 10Mbit/s. The slaves use their local 40MHz clock to generate the 10MHz return data clock, so no additional clocks are needed. The slave interface clocks are internally resynchronized during each transfer from the master.

- The data packets are 9bit long (see below) with always exactly one missing clock period between packets. This allows to separate clearly the packets for simplicity purpose and gives time for data transfer within the receiver (this will help to simplify the receiver electronics). The 9bit words are tranfered with LSB first (this allows the checksum to be calculated sequentially).

- The SDA and SCL lines follow the start and stop conditions of the I2C protocol. Conversely, there will be no acknowledge from the slave during a data transfer as this is the limiting point for the bus speed. In a general way, all the timings are secure while several conditions about the board distances are respected. This makes this system very safe.

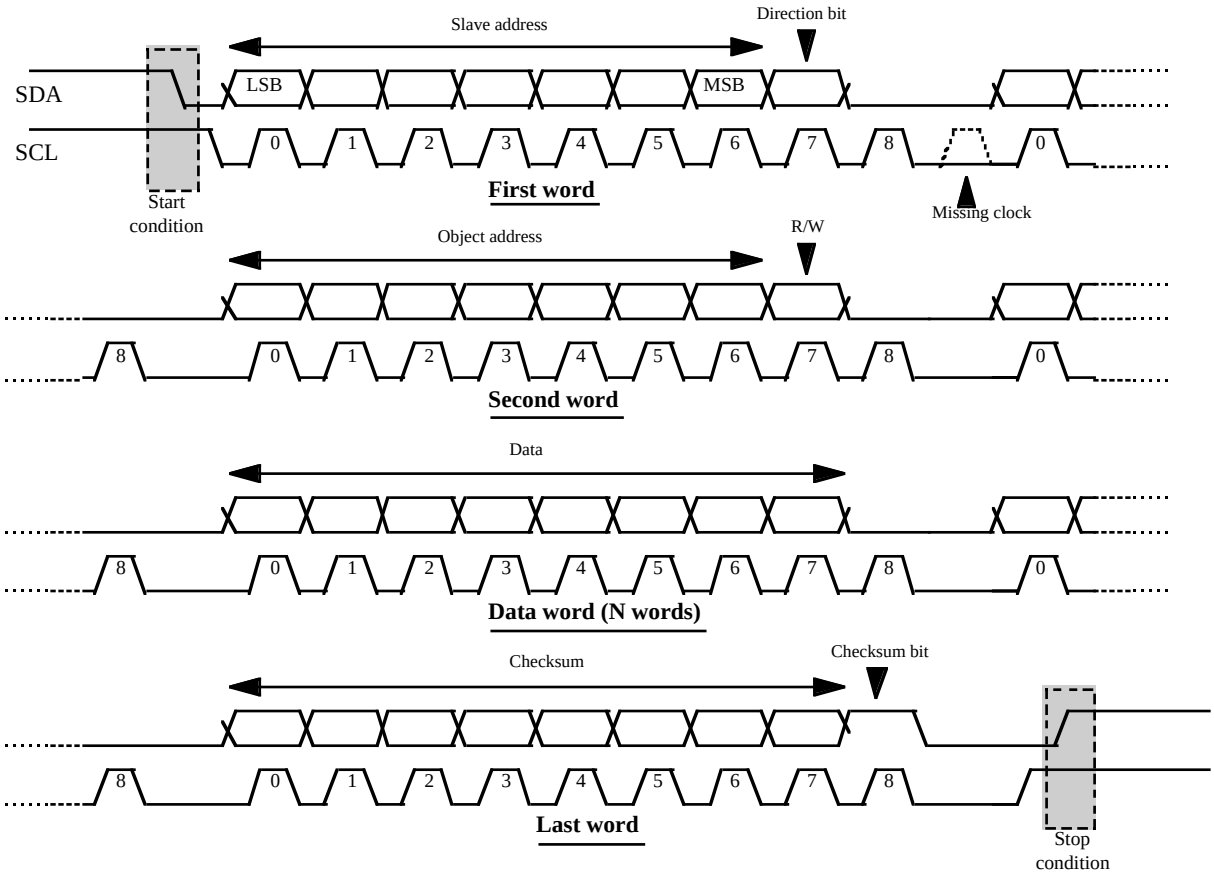
- To prevent the collisions, the emitter always checks that the line is not busy before taking hand on it. Moreover the open collector structure protects the bus against any short.

- The format of the response to a read request is the same as the format of the request except for the direction bit in the first word. This means that the data contained in the

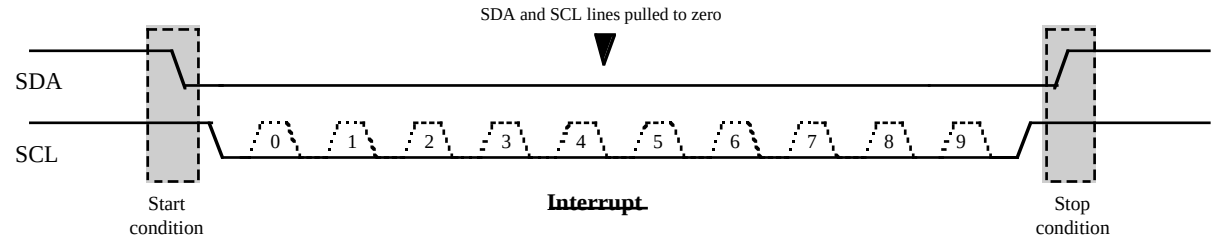
two first words is a copy of the one received from the master. This allows crosscheck and makes the control software more convenient.

- The slave provides a JTAG output in order to program any other FPGA on the host board.

Format of standard data transfers in the SPAC protocol



Format of the interrupt in the SPAC protocol



1.2. Frame description

The 9bit data packets look as follows...

bit #	8	7	6	5	4	3	2	1	0
1st word	0	direction	[a	d	d	r	e	s	s]
2nd word	0	R/W	[s	u	b	a	d	d	r]
3rd word*	0	[...	…	d	a	t	a	…	...]
4th word*	0	[...	…	d	a	t	a	…	...]
… *	0	[...	…	d	a	t	a	…	...]
last word	1	[c	h	e	c	k	s	u	m]

* : these words are optionnal.

The first word contains the board address and a bit to select the direction (1 for the master -> slave transfer and 0 for the other direction). The broadcast mode definition is included in the address field. The second contains the R/W bit and the internal subaddress.

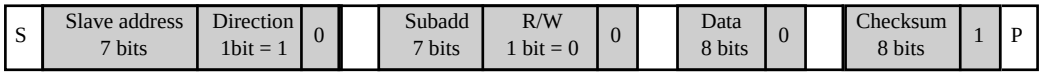
The number of additional data words is unspecified. The last word will transmit a checksum which allows the receivers to check for errors. The last bit 8 will be a flag for recognizing this checksum word.

After any master -> slave transfer on the line, all the concerned slaves verify the validity of the checksum byte, and the correct reception of the frame. In case of error, the slave sends an interrupt back to the master. An interrupt can also be sent on an external request, that has to be managed by the user. The slave status register can inform the master if the interrupt comes from a bus error or an external command.

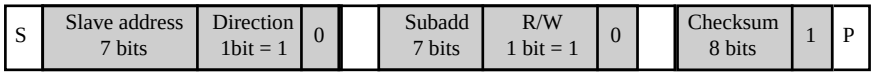
The interrupt signal consists of pulling down both SDA and SCL lines during a normal 9bit command length. The master will then check the slave status register before taking a decision.

Examples of data transfers in the SPAC protocol

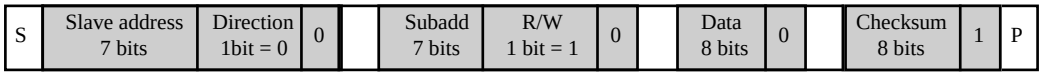
The master loads an 8bit-register within one slave.



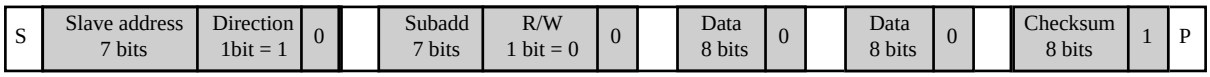
The master asks for reading an 8 bit register within one slave.



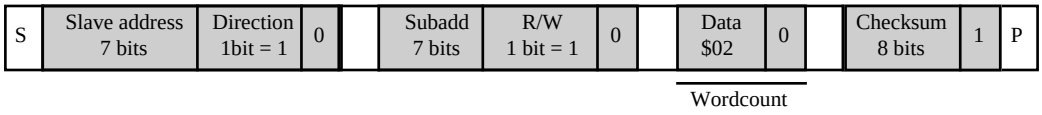
A slave sends the content of an 8 bit register to the master.



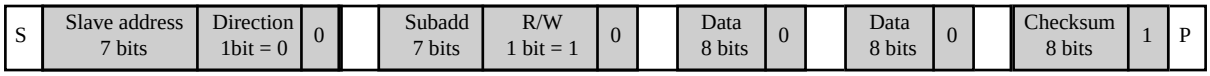
The master loads a 16bit-register within one slave.



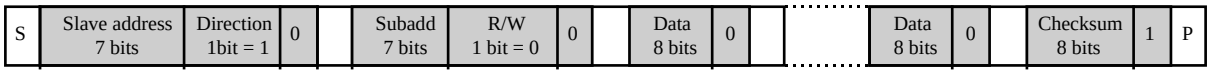
The master asks for reading a 16 bit register within one slave.



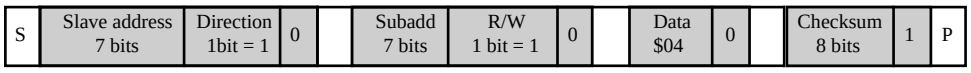
A slave sends the content of a 16 bit register to the master.



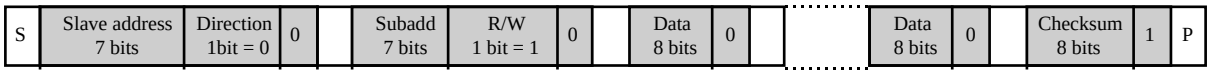
The master loads a 32 bit register within one slave.



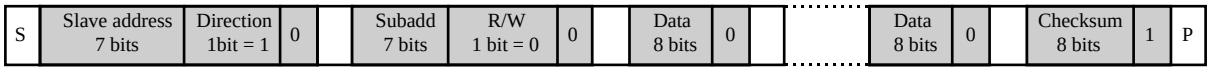
The master asks for reading a 32 bit register in one slave.



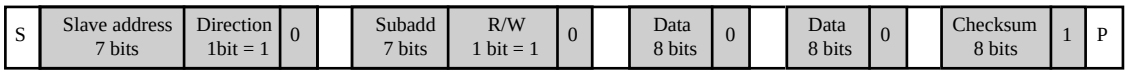
A slave sends the content of a 32 bit register to the master.



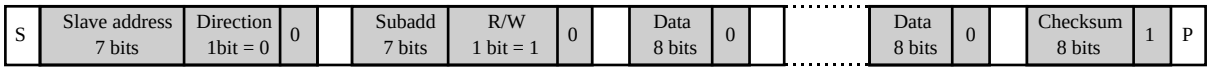
The master loads a FIFO within one slave.



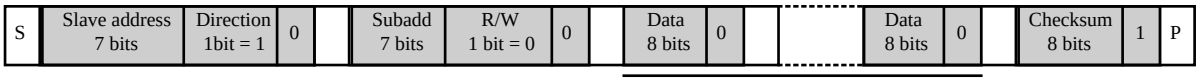
The master asks for reading a FIFO within one slave.



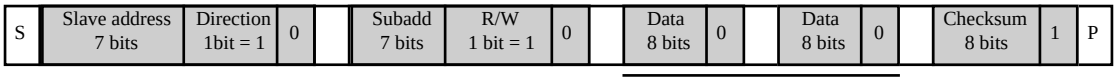
A slave sends the content of a FIFO to the master.



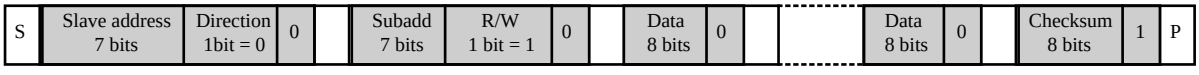
The master loads a N*word RAM within one slave (the NTA has previously been loaded).



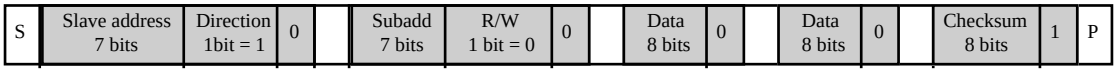
The master asks for reading a RAM in block mode in one slave (the NTA has previously been loaded).



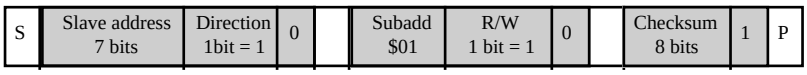
A slave sends the content of a RAM in block mode to the master.



The master loads a 16bit-register within a group of slaves in broadcast mode.



The master asks for reading the slave status register of a group of slaves in broadcast mode, after an interrupt.



1.3. About the collisions

Concerning the collisions, all the slaves and the master are permanently spying the data transfers. They can't speak if the line is busy, even for sending an interrupt. The protocol is intended so that no collision is possible between data transfers. The only possible collision may be due to an interrupt crossing a master to slave data transfer. This can happen only in two cases : either there was a checksum error detected by an adressed slave, or an external interrupt was generated in a slave board. There are two different ways of dealing with such a occurrence :

a) When the master has to emitt, it waits for a back to back propagation delay on the line plus some extra time before sending a new command. Then it gives time to a possible interrupt due to a cheksum error to arrive and avoids collision.
b) If the master doesn't wait between commands, or in case of an external interrupt, an occuring interrupt may destroy a current data transfer. The master detects the interrupt and immediatly stops the transfer. Parallely, all the slaves go back to idle state. Then the source of interruption is looked for through the broadcast read status command. The stopped transfer will resume later.

In case of a broadcast write, a checksum error could be seen by several slaves, and generate the sent of several interrupts at the same time. Then the open collector structure prevents the short circuits, and the interrupt message is not modified by superposition.

Moreover, if a broadcast read status is requested, each of the slaves will wait for a different delay (address $\times$ 100 ns) after each transfer. So, if the user respects some distance conditions (less than 10 meters between any couple of slaves), no collision is possible. In this case, a software delay depending of the number of slaves and their addresses should be calculated for giving time to the master to wait for all slave replies.

1.4. Distance conditions

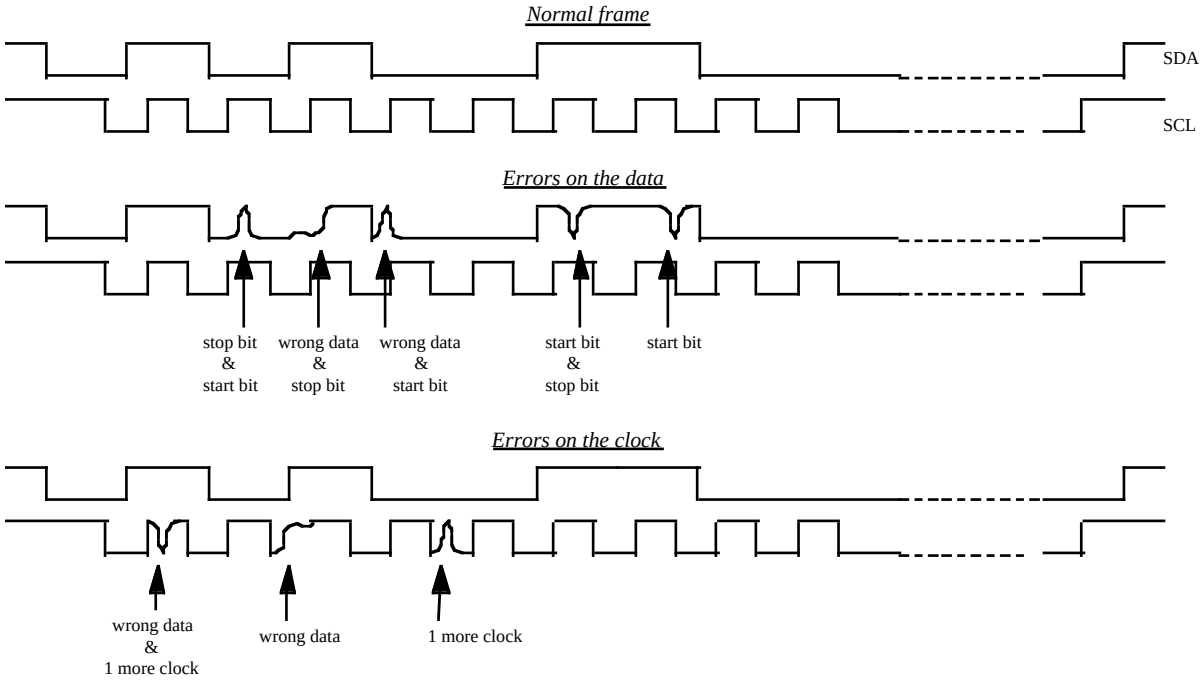
The maximum distance between any set of two slaves usable with the BTL level bus is 10 meters. The master should not be placed too far of the first slave for impedance adaptation reasons. In the case of a single master/single slave BTL implementation, the distance could go up to 30 meters.

The safest way to use the bus over a long distance is to use the differential link with PECL levels and transceive the levels into BTL around the slave physical location.

1.5. Error protection

Since the error rate is low and the protocole allows the master to repeat a message on a slave request (interrupt), it is no use to correct the errors (a correction device is very heavy, and expensive). The major point is to be able to detect the errors, and flag them.

Typical error sources



Two typical errors can occur :

- Most of the time the errors will generate a *frame syntax error*, that is to say *one more clock period*, or an unexpected *start or stop bit*. These errors will generate a *frame syntax error*, and most of the time a checksum error. Their detection is systematical. A frame can't be changed without generating an error, so the error detection rate is 100%. It also should be quoted that any glitch that modifies the data reception generates a frame syntax error.
- In a few cases, data can be changed without generating a frame syntax error. This can only happen if the line is held at a wrong value for at least 25 ns on SCL, and 50 ns on SDA (the glitches are excluded). Then, the *checksum error* flag should be set to one. A simple calculation shows that if ε_0 is the error rate per word (that doesn't generate a syntax frame error) and n the number of words in the frame, then the probability of missing a data error is approximatively of :

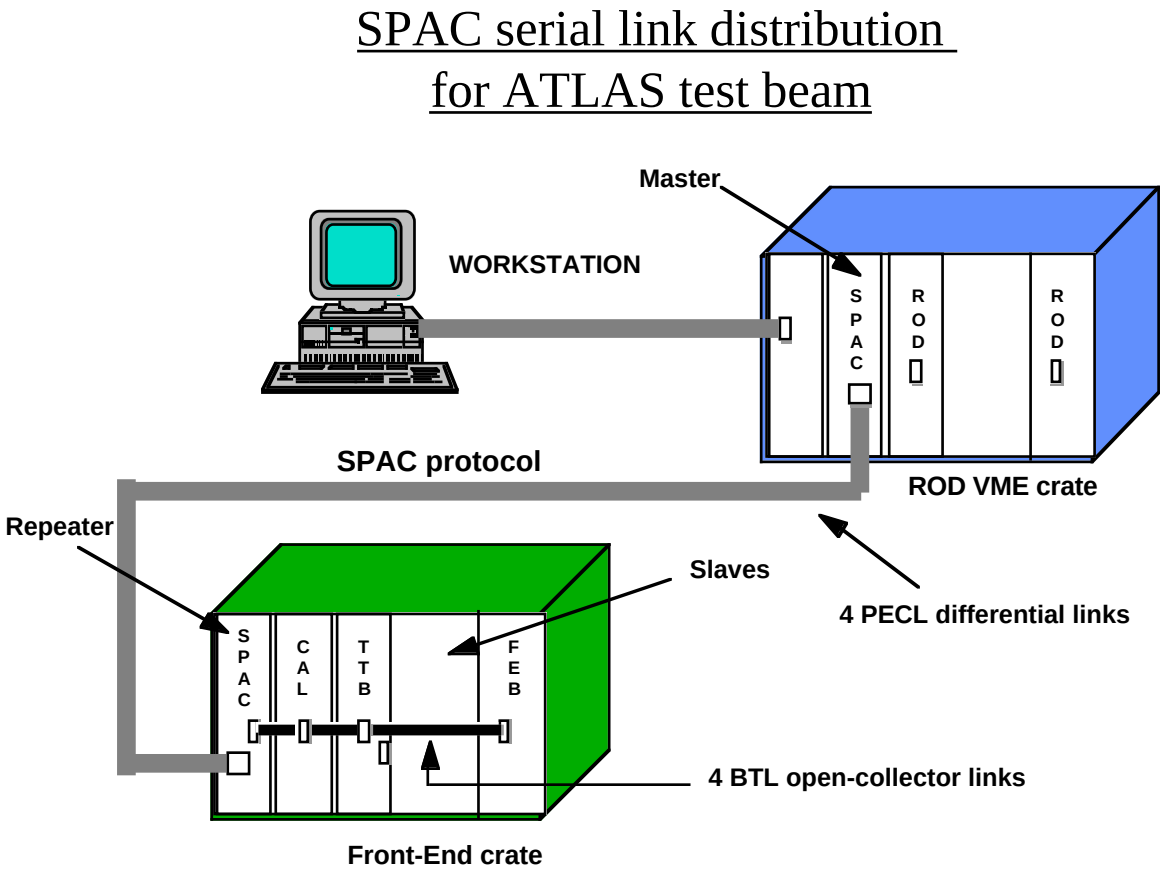
  $\eta = (n-1).n^2\varepsilon_0^2/(8n-1)$

  ex: n=10 ; $\varepsilon_0 = 10^{-6}$ err/word (<-> 1err/s) -> $\eta = 10^{-11}$ err/word

A *frame syntax error* may freeze the bus in an waiting state if no stop bit is seen. A *timeout* is programmed in the FPGAs. If one of them is busy for more than the longest message that can be sent, then the timeout device puts the slaves or the master in the idle state, and sets the timeout flag to one. The timeout waits for 786433 clock periods, i.e. 79 ms (10 Mhz clock).

In conclusion, the chance of missing an error is extremely low, and in no case the slave or the master can remain blocked in a wrong state. The SPAC bus error detection is thus very reliable.

1.6. System hardware for test beam.



For the test beam that will occur at CERN in 1998, the system will be used with the following hardware implementation :

- 4 wire bus in the front-end crate.
- BTL logic levels.
- one repeater board in the crate which will be connected to the master with 4 differential PECL links.

1.7. Performance considerations

Let us assume that we have to load a configuration of 40 registers and a RAM of 50 kB for each front-end board, and that we have 15 front-end boards in the crate.

1.7.1. Time to load the front-end boards

To write a 16-bit register, we have to send a 5 word frame. For all the registers, 200 (40×5) words are necessary. If we load the RAM with 10 kB accesses for example, we have to load once the NTA (5 words), and 5 times ten kilobyte (10000+3 words).

The whole loading of a board requires 200+5+50015 = 50220 words. As we can write all the boards in broadcast mode, and that one word corresponds to one µs, the time required to load the whole crate is : t_{write} = 50.2 ms.

1.7.2. Time to read the front-end board

To read a 16-bit register, we must send a read request (4 words) and read the slave reply (5 words). To read the RAM with 10 kB accesses, we have to load once the NTA (5 words), to send 5 reading requests (5×5 words), and to read 5 replies (5×(10000+3) words). The total is : 40×(4+5)+5×(5+10003) = 50400 words.

As the broadcast reading is not possible, we have to exchange 15×50400 = 756000 words. The time to read the crate is : t_{read} = 756.0 ms.

1.7.3. Time to write and read the front-end board

For this operation , we have to exchange 50220+756000 = 806220 words. The time to write and read is : $t_{wr/rd}$ = 806.2 ms.

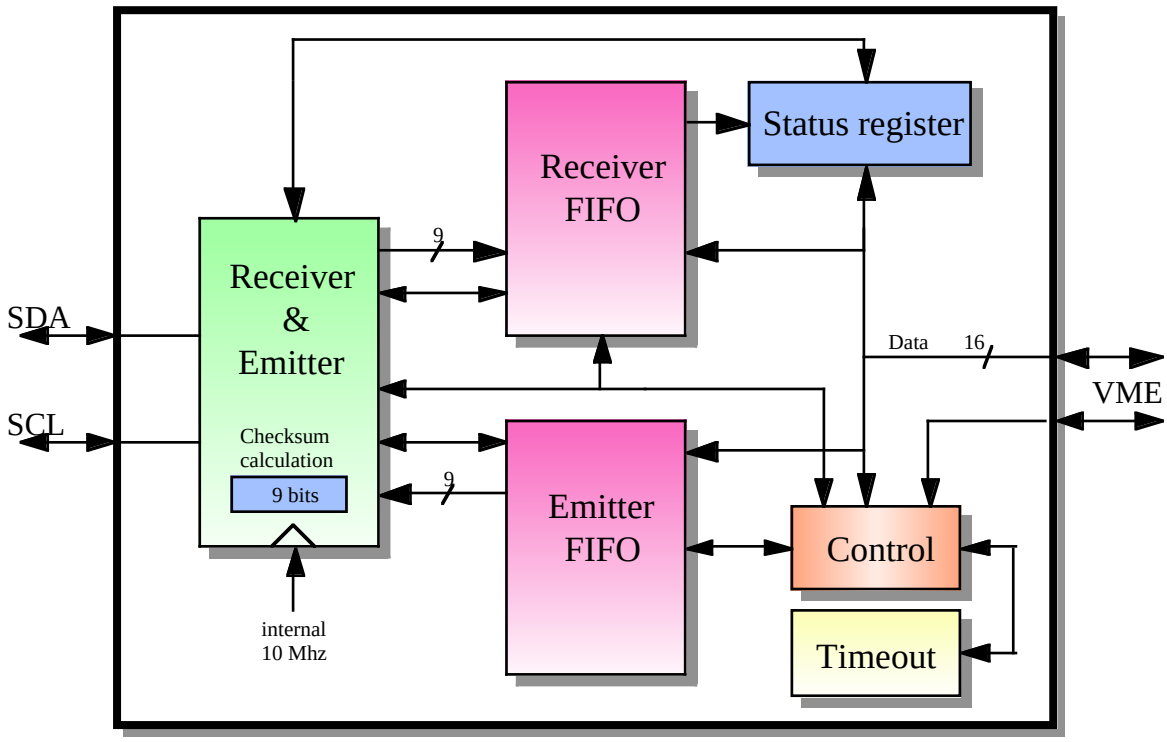
2. Hardware user's guide

2.1. The SPAC master board

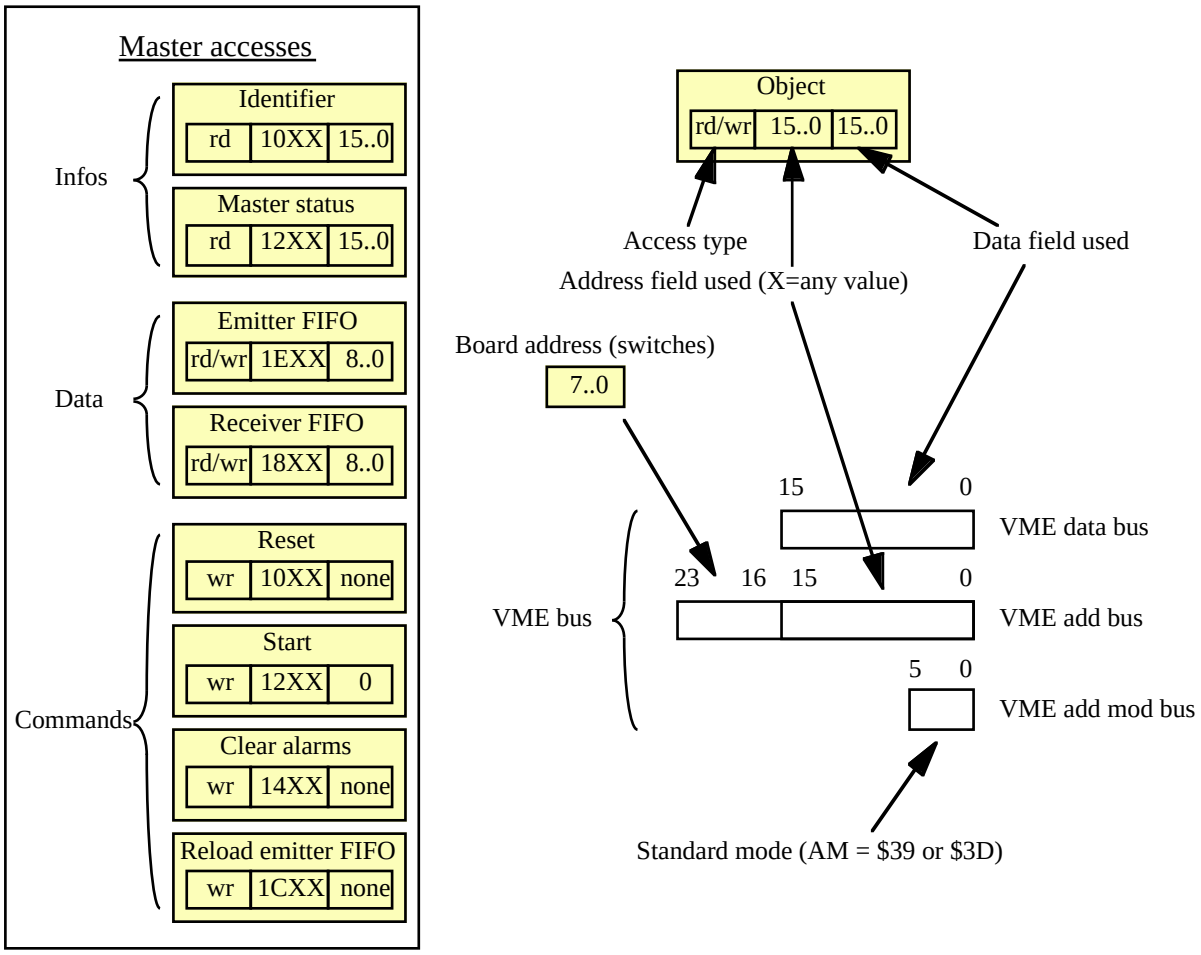
The SPAC master board is intended to be put in a VME crate, and driven by the VME bus.

2.1.1. The architecture

Architecture of the SPAC bus master



2.1.2. The VME master communications



- *Identifier* returns A110 in hexadecimal (16 bits)
- *Master status* is a 16 bit register :

2^9	Master checksum error
2^8	Timeout
2^7	Interrupt
2^6	Ready to receive message
2^5	Ready to send message
2^4	Receiver FIFO empty
2^3	Receiver FIFO full
2^2	Emitter FIFO empty
2^1	Emitter FIFO full
2^0	Run

Run : one bit register allowing the master to emitt.
Ready to send message is set to one if :
-*Emitter FIFO empty* =0 (means that EmitterFIFO is not empty)
-*Receiver FIFO full* =0 (to prevent data crashing)
- the bit 2^8 of the last word loaded in the EmitterFIFO is one (last word of a message)
Ready to receive message is set to one if :
- *Receiver FIFO empty* =0
- the bit 2^8 of the last word loaded in the ReceiverFIFO is one
Interrupt is set to one when an interrupt signal has been transmitted from a slave. This flag cannot appear with any other data transfer. No information is loaded in the ReceiverFIFO.

Timeout occurs when a slave has received a deficient message from the master. The ReceiverFIFO contains a one word message which contains the address of the complaining slave.

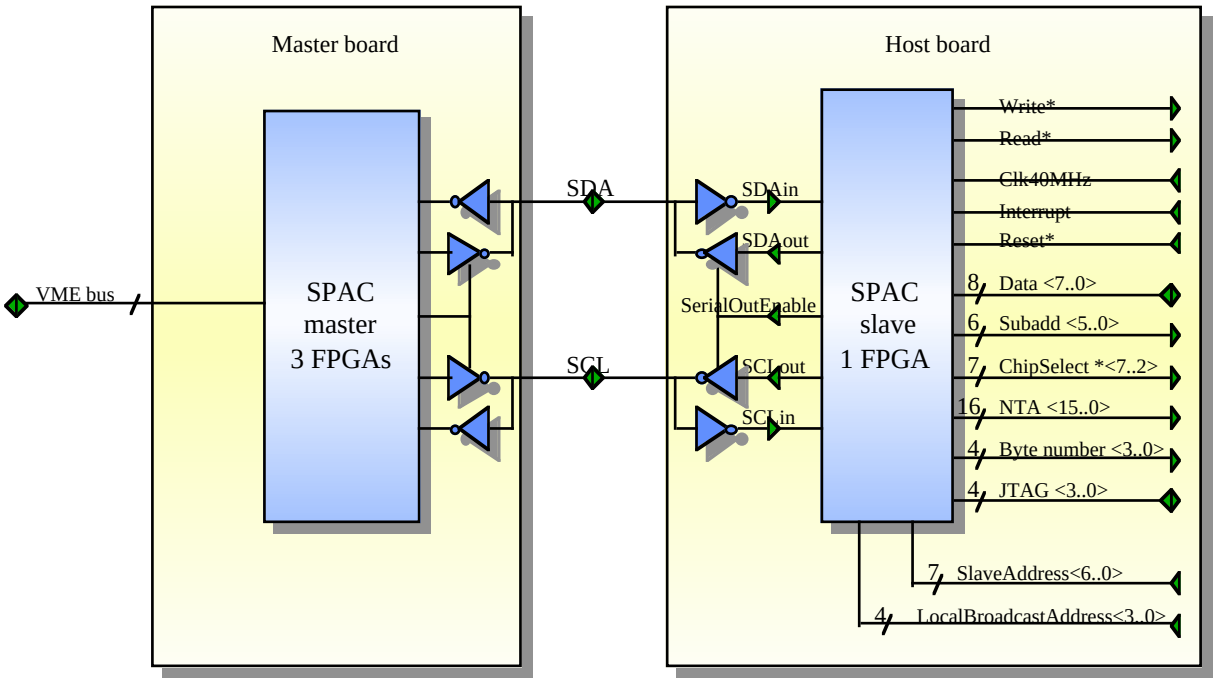
Master checksum error occurs when the master has received a deficient message from a slave. ReceiverFIFO contains the whole message. Its first word is the (supposed !) address of the complaining slave.

- *Reset* : resets the board during the access.
- *Run* : this bit controls the activity of the master. If Run is set to one, the master will begin to emitt as soon as the internal flag *Ready to send message* goes to one.
- *Clear alarms* resets Interrupt, Timeout and Master checksum error. Otherwise, these flags are never cleared !
- *Reload emitter FIFO* : moves the reading pointer of the Emitter FIFO to the first word. Thus, the FIFO is ready to be read again. A very careful use of this function should be made.

2.2. The SPAC slave installation

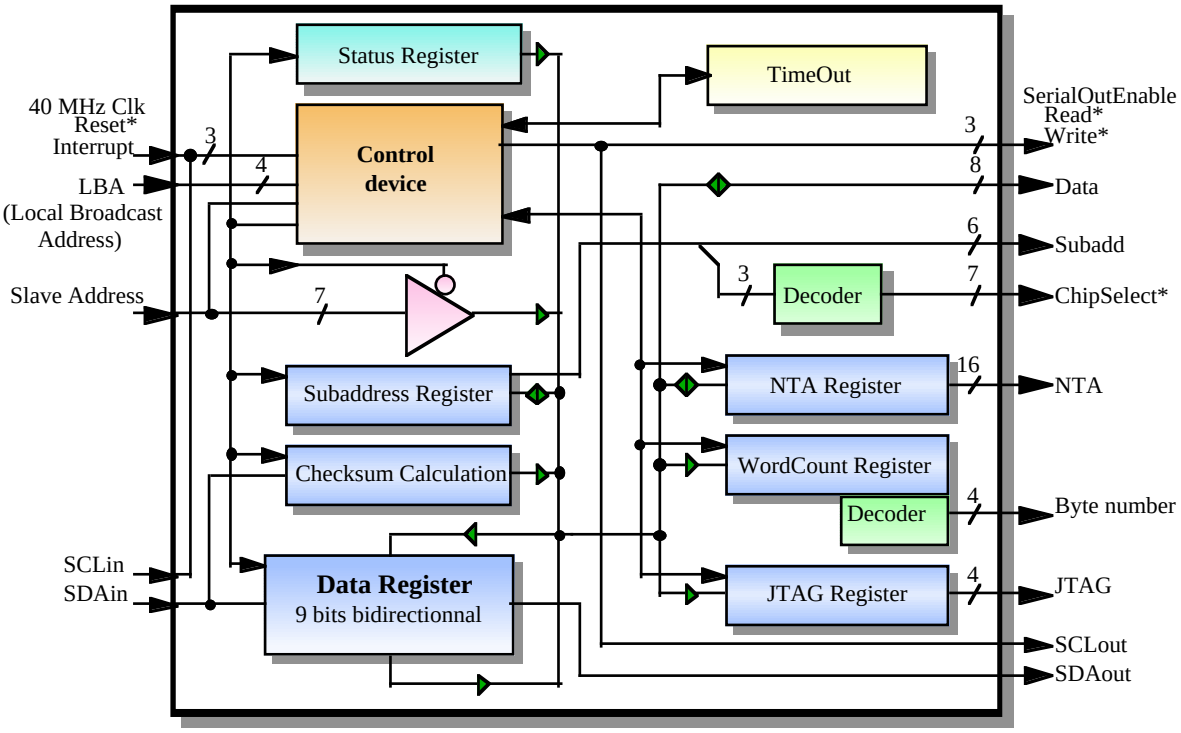
The SPAC slave FPGA will be implemented on the user's board.

Inputs/outputs of the SPAC bus



2.2.1. The architecture

Architecture of the SPAC bus slave



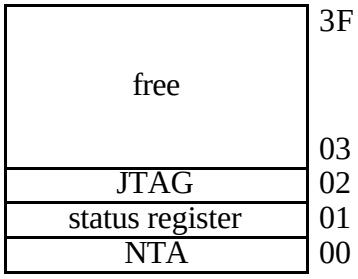
2.2.2. The Slave SPAC accesses

- *Address* defines a slave seen from the SPAC bus. *Address* is generated with a 7-bit switche on the host board. The *address<6..0>* range is not wholly available :

free	7F … 10
local broadcast call	0F … 01
global broadcast call	00

- *Local broadcast address* defines the broadcast group of a slave. 15 different groups can be chosen, using a 4-bit switch on the host board.

- *Subadd* defines an object that the slave corresponding to *address* can access. The *subadd* array is structured as shown :



2.2.3. How to build a message

The general form of the message to transfer through VME is the following one (remember that the checksum word exists in the data transfers through the SPAC bus, but is not transfered through VME) :

bit #	8	7	6	5	4	3	2	1	0
1st word	0	1 (direction)	a	d	d	r	e	s	s
2nd word*	0 (1 if no data)	R/W	s	u	d	a	d	d	r
3rd word*	0			d	a	t	a		
4th word*	0			d	a	t	a		
… *	0			d	a	t	a		
last word	1			d	a	t	a		

*:optionnal words

- The *direction* bit is set to 1 for a master to slave transfer, so the user will always give the value 1 to *direction*.
- *R/W* is set to1 for a reading request, 0 for a writing one.

The transfer is always big endian, that is to say that the first byte sent is the less significant, and the last one the most, in a 16 to 32-bit register. This rule applies for the NTA (16 bits), the wordcount (16 bits), and any register bigger than 1 byte.

In case of a write command (R/W=0), the data will be sent to the object pointed by *subadd* .
In case of a read command (R/W=1), the data will be sent to the 16-bit wordcount register. As the transfer is big endian, and the wordcount is set to $0001 before any loading, it is possible to send only one data word in a read command, if the wordcount is below $00FF. Indeed, the most significant byte of the wordcount will be 0 by default.
Two other special cases should be noticed. In a command for NTA reading, it is no use to load a value into the wordcount. The size of the NTA register (2 bytes) is internally known. And in a command for FIFO reading, the user can either load a value (n≤FFFF) to the wordcount, in order to read n bytes from the FIFO, or not load any data. Then, the whole FIFO will be read (until reception of a FIFO empty flag).

2.2.4. How to read a slave reply

The message brought back through VME appears as below. The slaves can only reply to the master, so *direction* = 0, and *R/W* =1. *Address* contains the address of the slave which is talking. *Subadd* gives the address of the object being read.

bit #	8	7	6	5	4	3	2	1	0
1st word	0	0 (direction)	a	d	d	r	e	s	s
2nd word*	0	1 (R/W)	s	u	b	a	d	d	r
3rd word*	0			d	a	t	a		
4th word*	0			d	a	t	a		
… *	0			d	a	t	a		
last word	1			d	a	t	a		

*:optionnal words

In case of an interrupt, the message is empty, but the *interrupt* flag of the master status register is set to 1.

2.2.5. The slave state machine

The slave is composed of one module, which is integrated within an FPGA. It can address FIFOs, RAMs, registers up to 32 bits and other memories.

Whatever the type of your memory, the slave generates 2 buses, which will be used or not :
- Byte number which gives the number of the byte of a pointed register
- Next address register (NTA) which gives the RAM address pointed

In case of a read command, the number of bytes to read is loaded in an internal 16-bit register, Wordcount. This register is special because it doesn't have any address, is written by a special protocol, and cannot be read (which anyway is no use). $0001 is loaded by default.

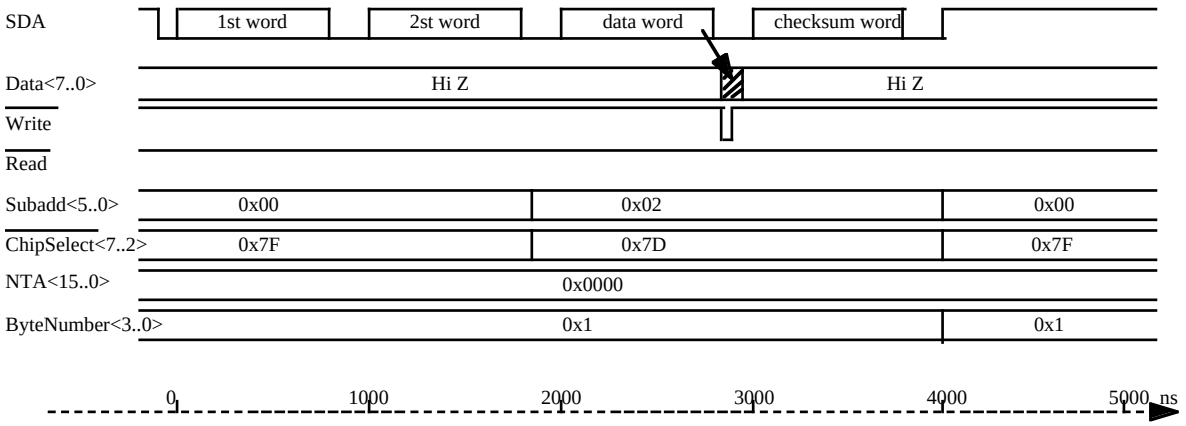
In case of a RAM access, data will be sent to the internal address loaded in the 16-bit next transfer address register (NTA). Any access to a RAM (read & write) implies a previous loading of the board's NTA.

Seen from the slave, the general protocol is desribed below:

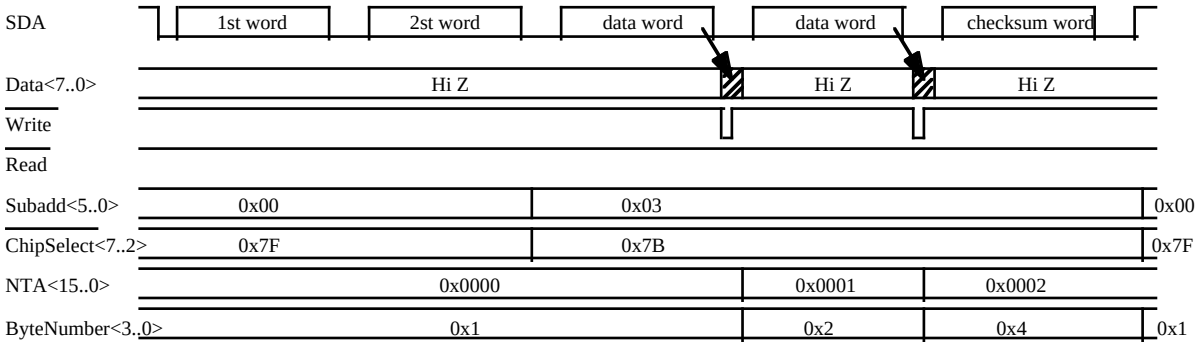
		Slave actions
write	received	(NTA already loaded in case of a RAM access) byte number = 1 Loop: next data byte -> memory byte-number incremented (1 to 4) NTA incremented (0 to $FFFF) goto Loop
read	received	no data byte -> (wordcount = 1 by default) or 1 data byte -> wordcount (= 1 to $FF) or 2 data bytes -> wordcount (= 1 to $FFFF)
	replied	(NTA already loaded in case of a RAM access) byte number = 1 Loop: memory -> next bytes read byte-number incremented (1 to 4) NTA incremented (0 to $FFFF) Wordcount decremented Loop until Wordcount = 0
		memory pointed with subadd

2.2.6. The timings of the signals between the slave and the host board

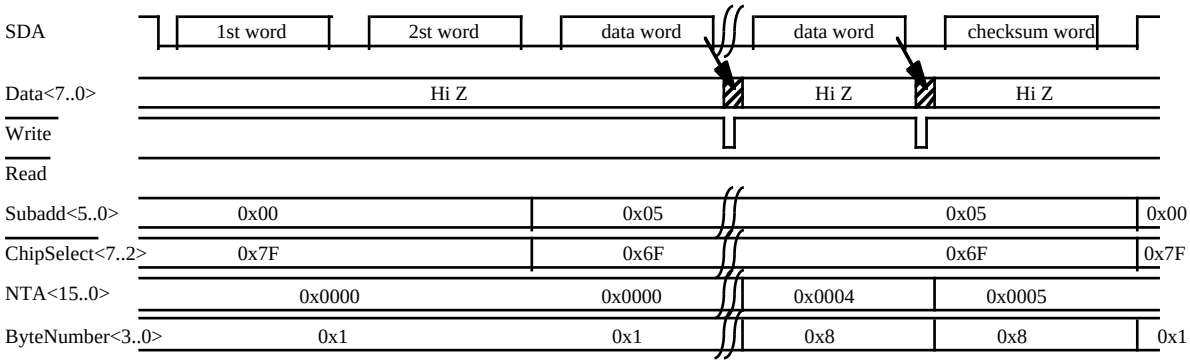
8-bit register writing



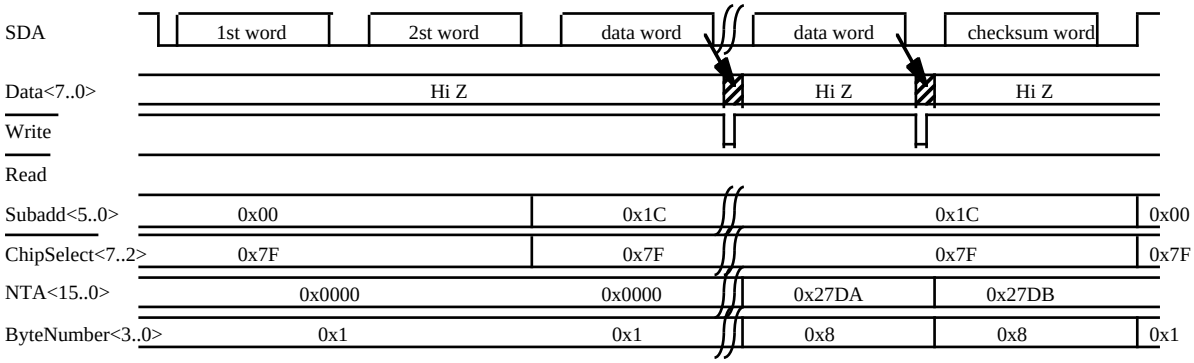
16-bit register writing



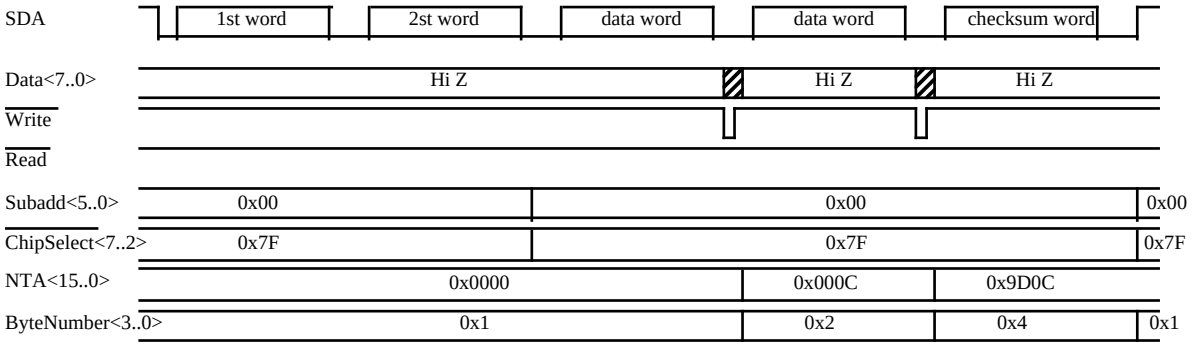
32-bit register writing



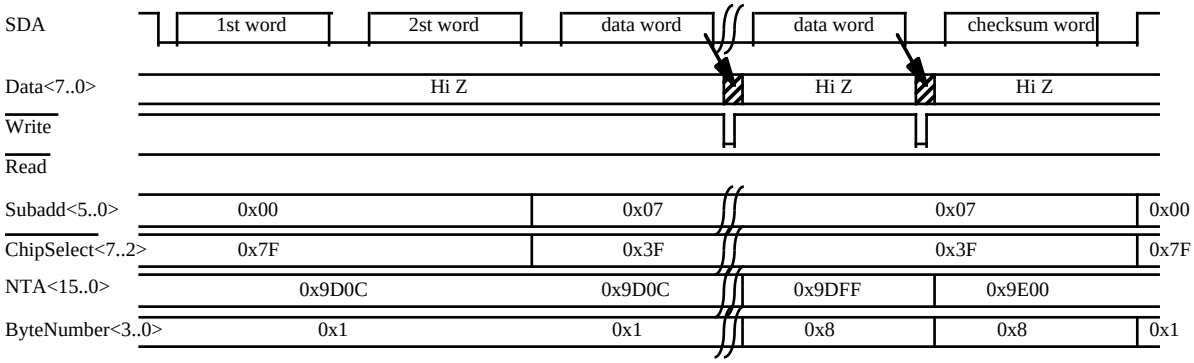
FIFO writing



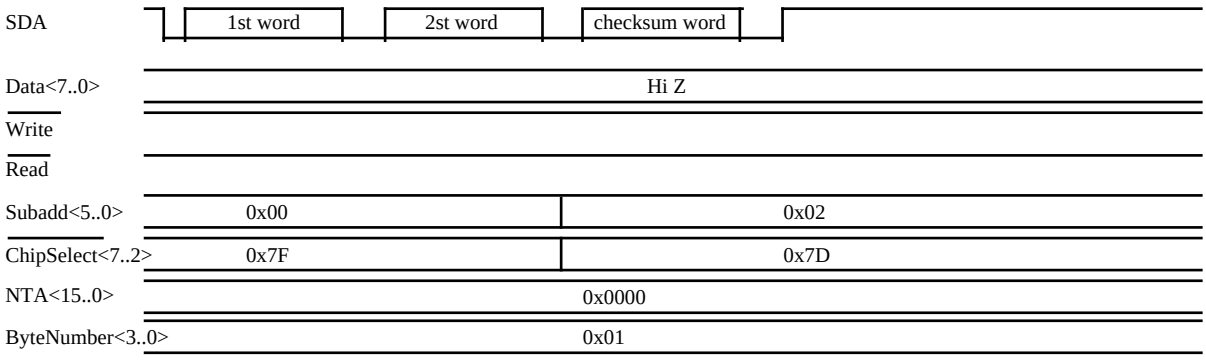
NTA register writing



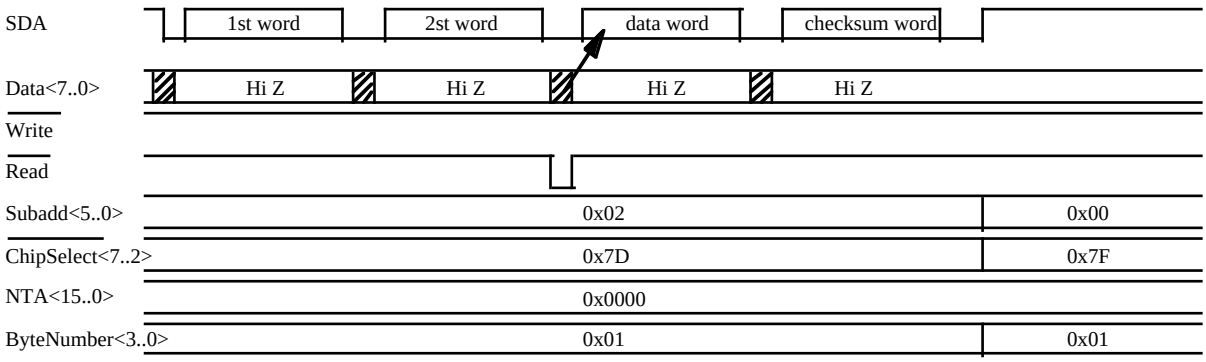
RAM writing (NTA already loaded)



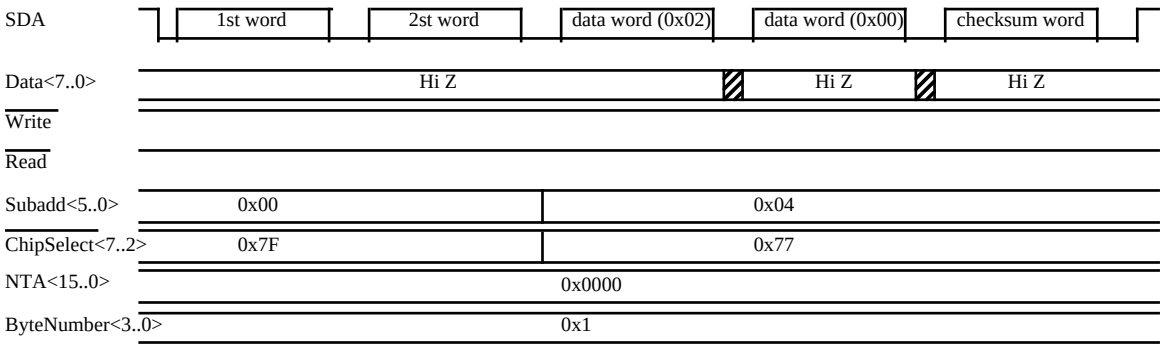
8-bit register reading command



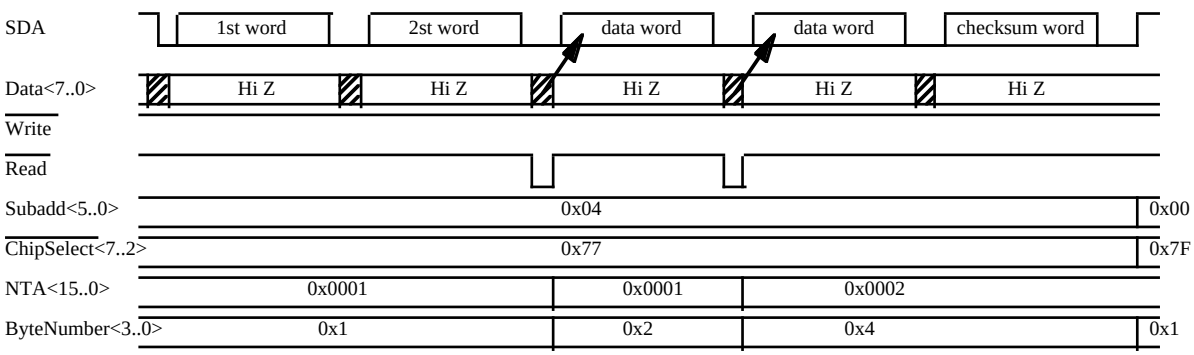
8-bit register reading execution



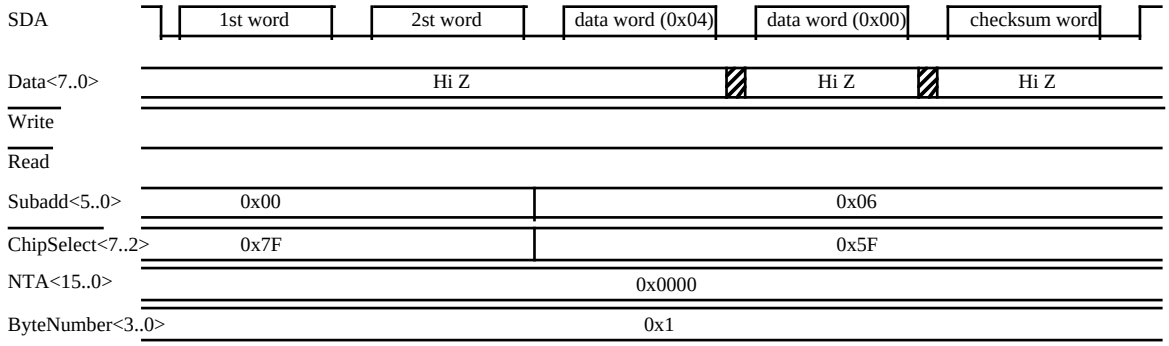
16-bit register reading command



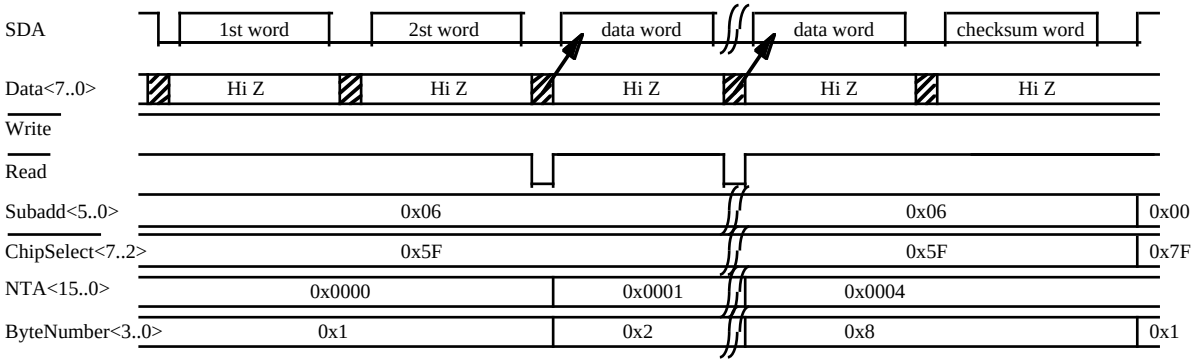
16-bit register reading execution



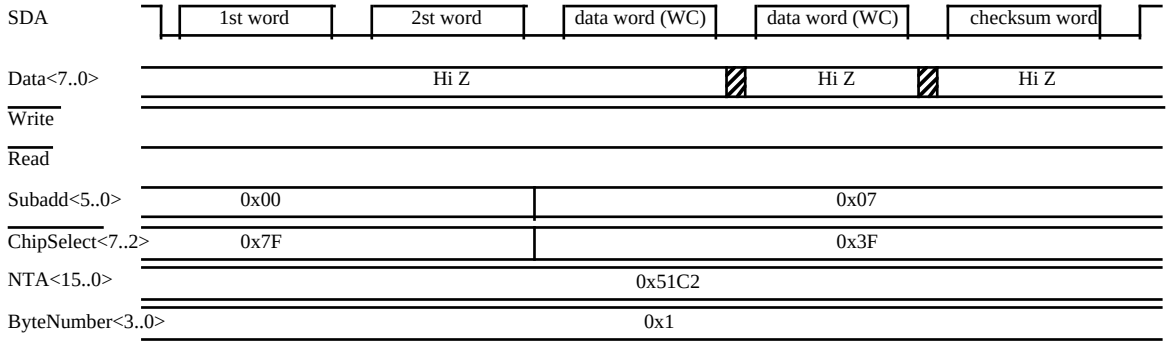
32-bit register reading command



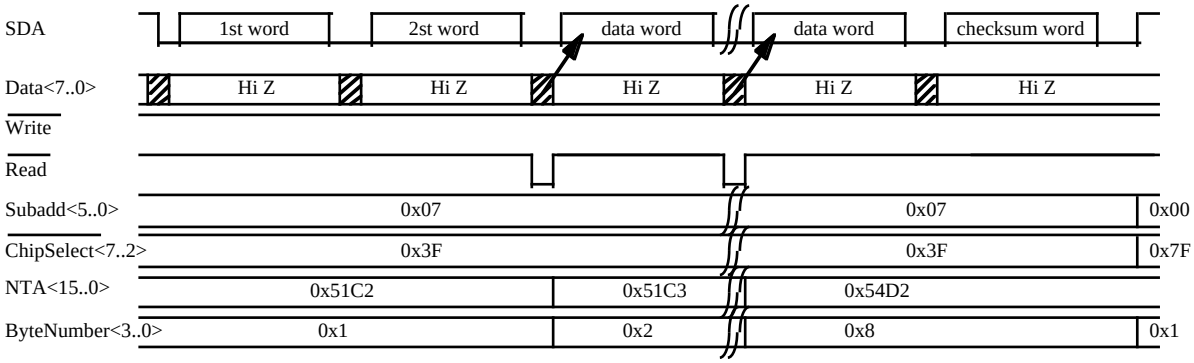
32-bit register reading execution



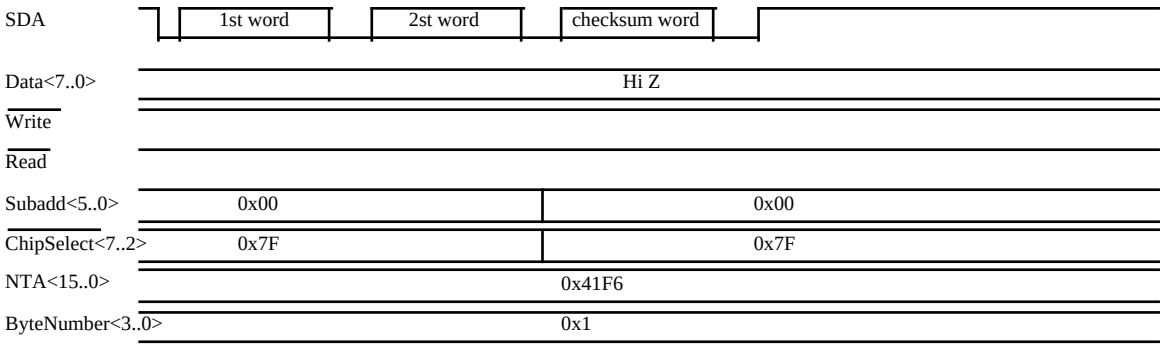
FIFO reading command



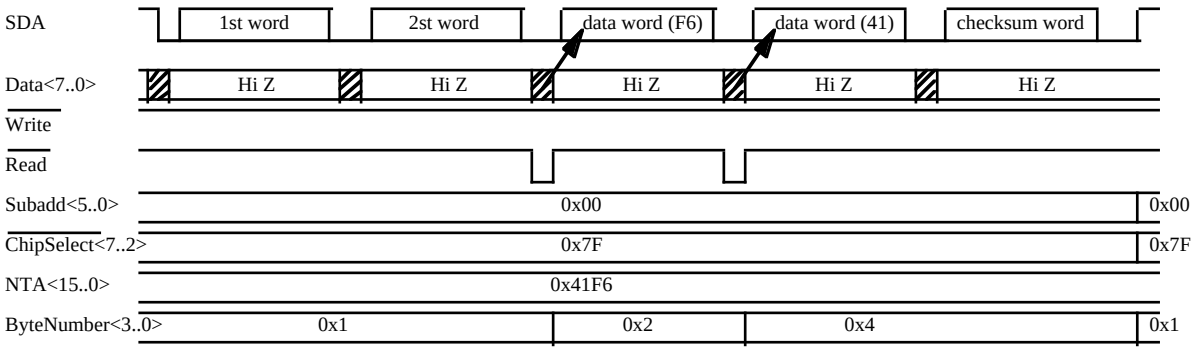
FIFO reading execution



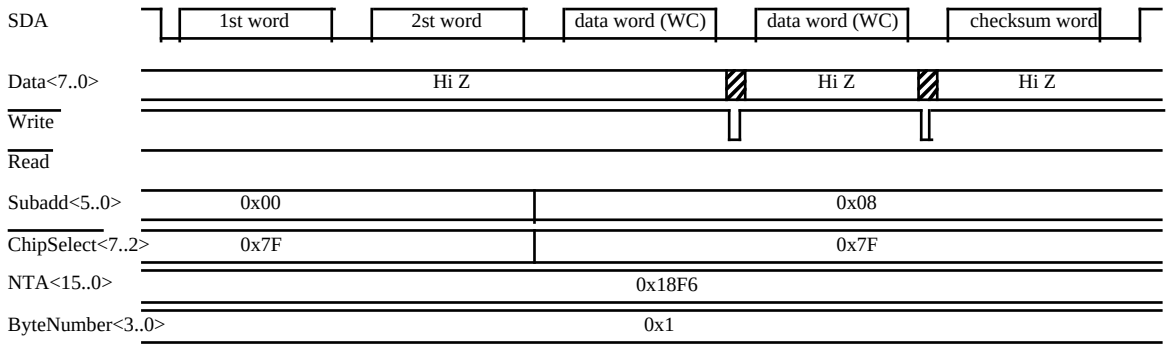
NTA register reading command



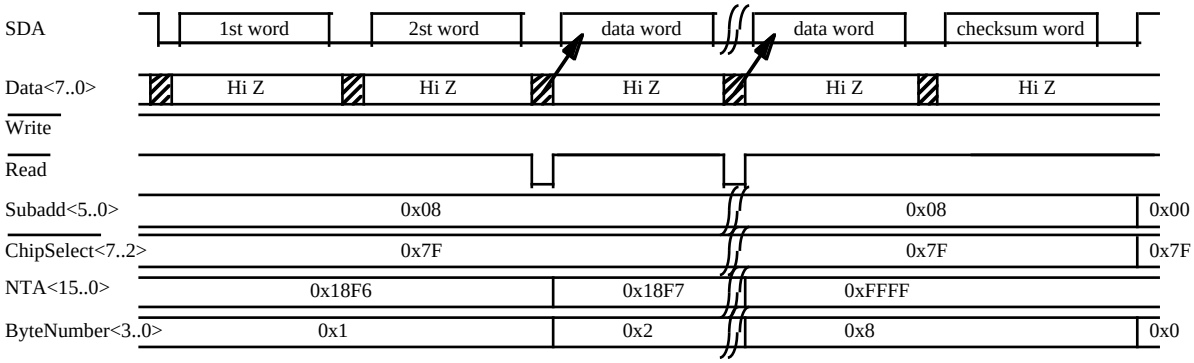
NTA register reading execution



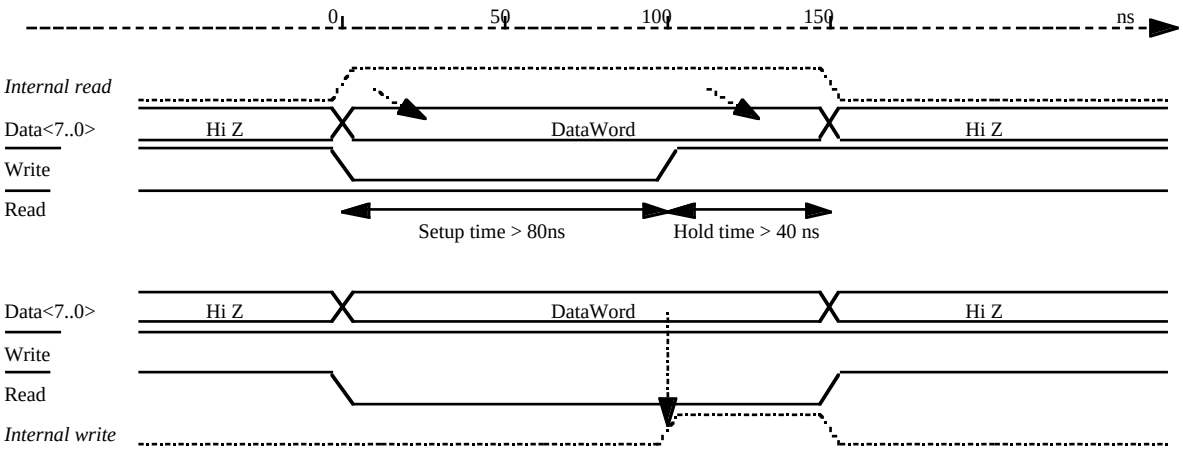
RAM reading command (NTA already loaded)



RAM reading execution



Read/Write timings

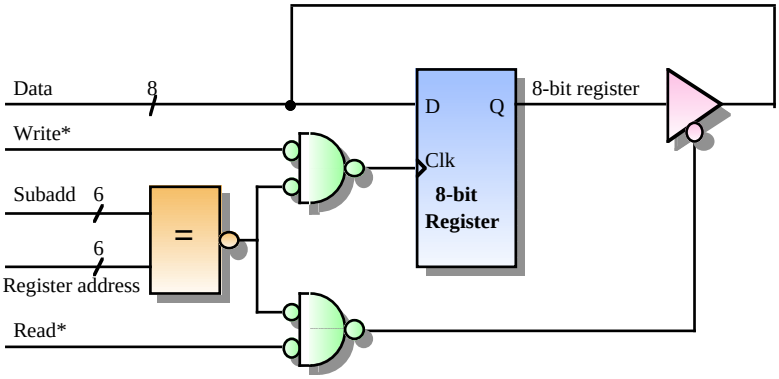


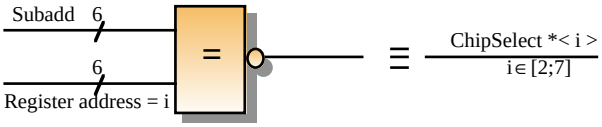
2.2.7. The implementation of the slave on its host board

See the `spac_kit.ps` file on the web.

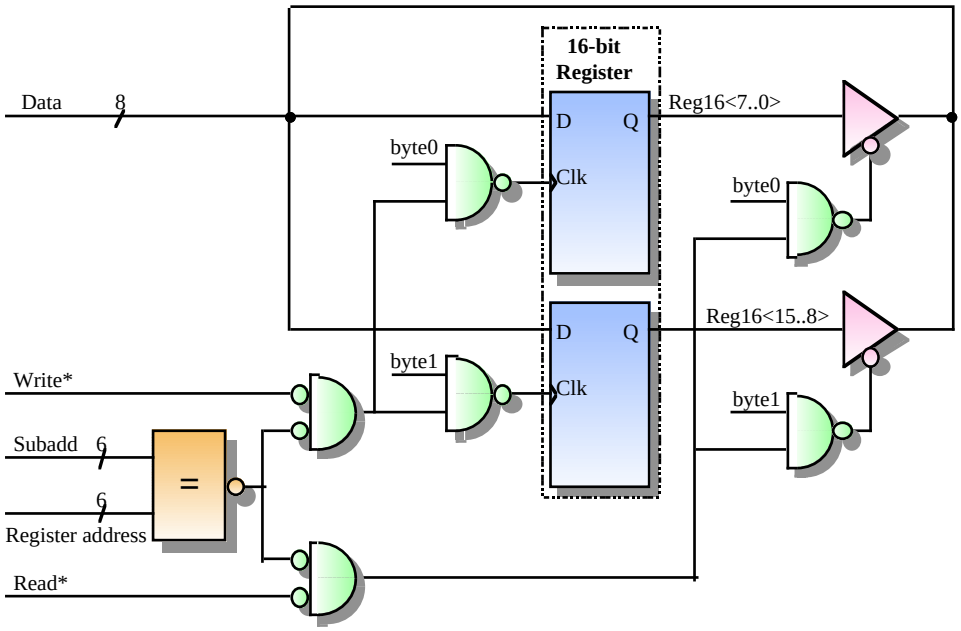
2.2.8. The objects implementation

8-bit register implementation

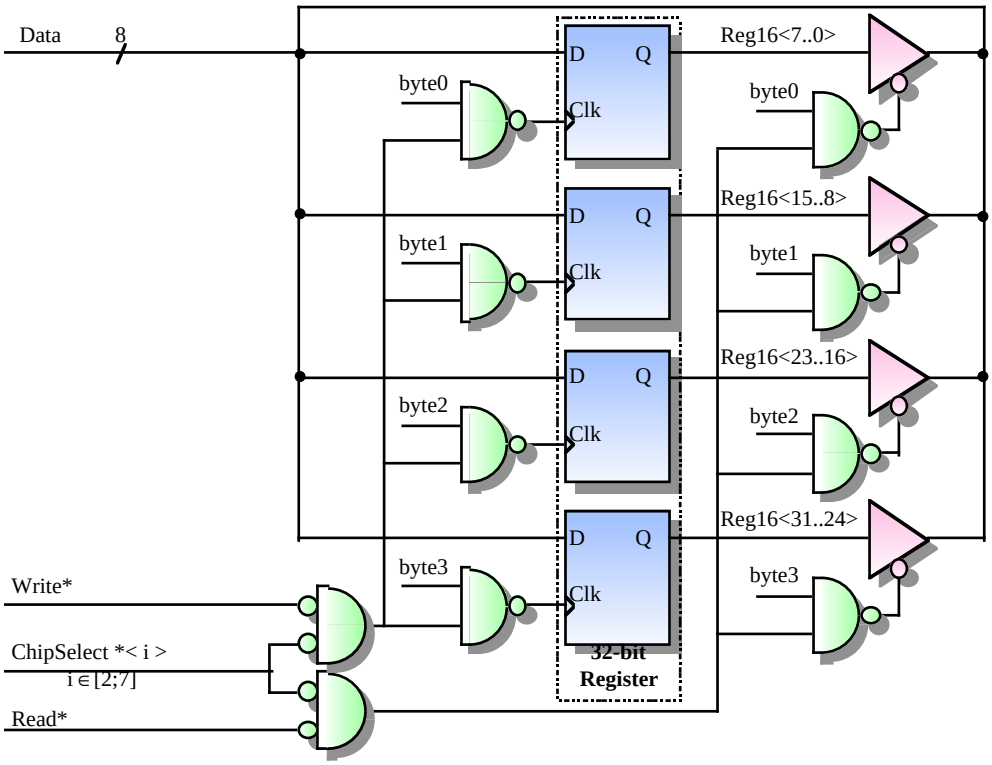




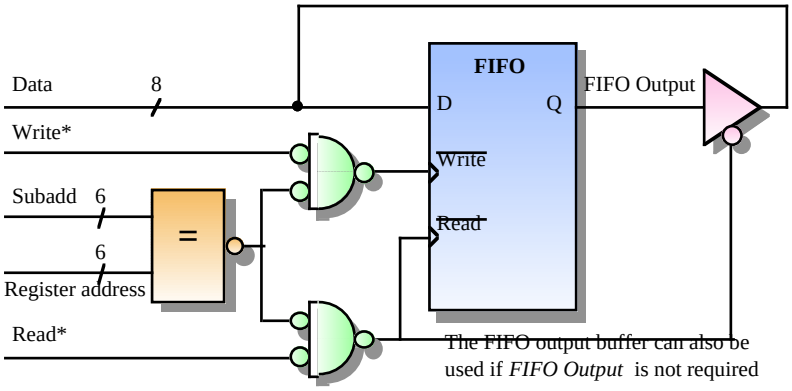
16-bit register implementation



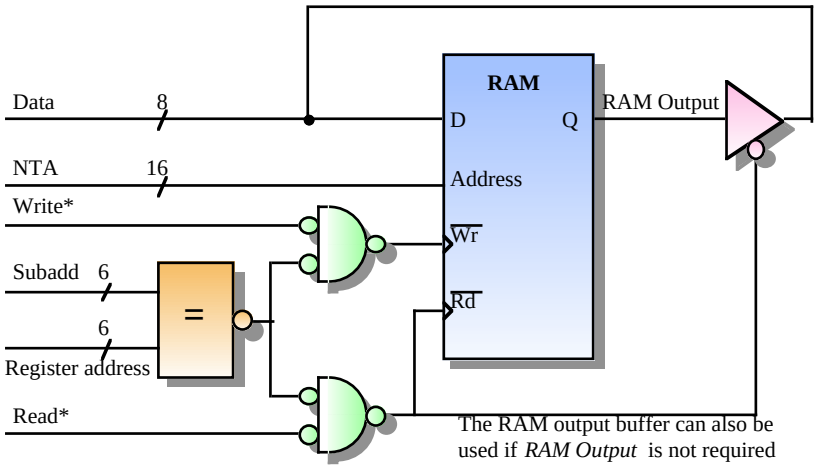
32-bit register implementation



FIFO implementation



RAM implementation



2.3. Standard connectivity

2.3.1. BTL technology

- For the bidirectional solution, the connector type is a differential Lemo :

A	SDA
B	SCL

A shielded 2 wire cable is required.

- For the unidirectionnal bus, a 10 pin connector is used (male HE-10, 2*5 pins) :

gnd	1	2	master to slave SDA
gnd	3	4	master to slave SCL
gnd	5	6	gnd
slave to master SCL	7	8	gnd
slave to master SDA	9	10	gnd

A flat 10 wire cable is required.

2.3.2. PECL technology

- A PECL link can only be unidirectionnal. A 10 pin connector is used (the same as above) :

master to slave SDA -	1	2	master to slave SDA +
master to slave SCL -	3	4	master to slave SCL +
slave to master SDA -	5	6	slave to master SDA +
slave to master SCL -	7	8	slave to master SCL +
gnd	9	10	gnd

A 10 wire cable, with twisted pairs is required.

3. SPAC software user's guide

3.1. The SPAC library of functions : spac.h

The SPAC library is a set of functions to manage the communications of the SPAC bus. This library allows a simple use of the SPAC bus, with functions optimized for speed. All the usual applications of the SPAC bus can be managed by the library. However, the vme library may be useful for special applications (debugging,...). In any case, the vme library is used by the SPAC library, and has to be adapted to the VME controller. Conversely, the SPAC library is universal.

The type SPACMaster is a transparent structure that is wholly defined SPACDeclareBoard().

```
SPACMaster* SPACDeclareBoard(u_short CrateNumber, u_short
AddressModifier, u_short BoardAddress);
```

SPACDeclareBoard defines and initializes a SPAC board. `CrateNumber` depends on the VME crate. The accepted values of `AddressModifier` are 0x39 and 0x3D. `BoardAddress` is the VME address of the Master card. The returned pointer ought to be declared as `SPACMaster*` because it points to a structure. *run* is set to one.

```
void SPACWriteRegister (SPACMaster* Card, u_char Address,
u_char SubAddress, u_long Data, u_char Size);
```

SPACWriteRegister writes `Data` in the `Size` byte(s) register pointed by `SubAddress`, on the slave board `Address`, and from `Card`. `Size` must be 1, 2, 3 or 4. `Address` set to 0 produces a broadcast write.

```
u_long SPACReadRegister (SPACMaster* Card, u_char Address,
u_char SubAddress, u_char Size);
```

SPACReadRegister returns the value of the `Size` byte(s) register pointed by `SubAddress` on the slave board `Address`, and from `Card`. `Size` must be 1, 2, 3 or 4. This function does not support broadcast, so 0 is not a valid value for `Address`.

```
void SPACWriteFIFO (SPACmaster* Card, u_char Address,
u_char SubAddress, u_char* Data, u_short Size);
```

SPACWriteFIFO writes `Size` byte(s) of the buffer `Data` in the FIFO pointed by `SubAddress`, on the slave board `Address`, and from `Card`. Size must be different to 0. `Address` set to 0 produces a broadcast write.

```
u_char* SPACReadFIFO (SPACMaster* Card, u_char Address,
u_char SubAddress, u_short Size);
```

SPACReadFIFO returns an allocated buffer, resulting of the `Size` byte(s) read from the FIFO pointed by `SubAddress`, on the slave board `Address`, and from `Card`. Size must be different to 0. This function does not support a broadcast read,so 0 is a forbidden address.

```
void SPACWriteRAM (SPACMaster* Card, u_char Address, u_char
SubAddress, u_char* Data, u_short Size, u_short
RAMStartAddress);
```

SPACWriteRAM writes `Size` byte(s) of the buffer `Data` from `RAMStartAddress` to `RAMStartAddress+Size-1` in the RAM pointed by `SubAddress`, on the slave board `Address`, and from `Card`. Size must be different to 0. Address set to 0 produces a broadcast write.

```
u_char* SPACReadRAM (SPACMaster* Card, u_char Address,
u_char SubAddress, u_short Size, u_short RAMStartAddress);
```
SPACReadRAM returns an allocated buffer, resulting of the `Size` byte(s) read from the RAM pointed by `SubAddress`, from the RAM address `RAMStartAddress` to `RAMStartAddress+Size-1`, on the slave board `Address`, and from `Card`. Size must be different to 0. This function does not support a broadcast read,so 0 is a forbidden address.

```
void SPACRun (SPACMaster* Card, u_char State);
```
SPACRun modifies the run bit of `Card`. `State` can be :

- `OFF` : to stop the emission.
- `ON` : to allow the emission.

```
void SPACReset (SPACMaster* Card);
```
SPACReset resets the `Card`.

```
u_short SPACIdentifier (SPACMaster* Card);
```
SPACIdentifier returns the identifier of `Card`. Its value : `0xA110`.

```
u_char SPACMasterStatus (SPACMaster* Card);
```
SPACMasterStatus returns the Master status value. The result may be used as follow :

- (result & RUN) != 0 : the master can emitt.
- (result & EMITTER_FIFO_FULL) != 0 : Emitter FIFO is full
- (result & EMITTER_FIFO_EMPTY) != 0 : Emitter FIFO is empty
- (result & RECEIVER_FIFO_FULL) != 0 : Receiver FIFO is full
- (result & RECEIVER_FIFO_EMPTY) != 0 : Receiver FIFO is empty
- (result & READY_TO_SEND) != 0 : the Master is ready to send
- (result & READY_TO_RECEIVE) != 0 : the Master is ready to receive
- (result & INTERRUPT) != 0 :an interrupt signal has been transmitted from a slave
- (result & TIMEOUT) != 0 : a Timeout has occurred
- (result & MASTER_CHECKSUM_ERROR) != 0 : a Master Checksum error has occurred

```
void SPACReloadEmitterFIFO (SPACMaster* Card);
```
SPACReloadEmitterFIFO moves the reading pointer of the Emitter FIFO to the first word. The FIFO is ready to be read again. This function should be used carefully.

```
void SPACClearAlarms (SPACMaster* Card)
```
SPACClearAlarms resets InterruptTimeout and Master checksum error.

3.2. The implicit VME Library : `vme.h`

This library is composed of 5 functions :

```
void VMEInitialize (SPACMaster* Card);
```
VMEInitialize(`Card`) defines a reserved system address to realise the VME accesses.

```
void VMEWrite (u_short* Address, u_short Value);
```
VMEWrite (`Address, Value`) writes `Value` at the VME address `Address`.

```
u_short VMERead (u_short* Address);
```
VMERead (`Address`) returns the value read at `Address`.

```
void VMEWriteBlock (u_short* Address, u_short* Buffer,
u_short Size);
```

VMEWriteBlock (Address, Buffer, Size) writes the block Buffer of Size word(s) to the offset Address.

```
void VMEReadBlock (u_short* Address, u_short* Buffer,
u_short Size);
```

VMEReadBlock (Address, Buffer, Size) reads a block of Size word(s) from the base address Address into Buffer.

In order to adapt the SPAC Library to an other platform, 3 functions must be changed:

- `void VMEInitialize (SPACMaster* Card)`
- `void VMEWriteBlock (u_short* Address, u_short* Buffer, u_short Size)`
- `void VMEReadBlock (u_short* Address, u_short* Buffer, u_short Size)`

The initialization of the VME requires 3 values :

- VME Base address = BoardAddress << 16
- AddressModifier = 0x39 or 0x3D
- CrateNumber

3.3. The constants

The VME offsets of the SPAC master board :

IDENTIFIER	= 0x1000
MASTER_STATUS	= 0x1200
EMITTER_FIFO	= 0x1E00
RECEIVER_FIFO	= 0x1800
SPAC_RESET	= 0x1000
RUN_ADD	= 0x1200
CLEAR_ALARMS	= 0x1400
RELOAD_EMITTER_FIFO	= 0x1C00

The state called by SPACRun

ON	= 1
OFF	= 0

The bits of the status register :

MASTER_CHECKSUM_ERROR	= 0x0200
TIMEOUT	= 0x0100
INTERRUPT	= 0x0080
READY_TO_RECEIVE_MESSAGE	= 0x0040
READY_TO_SEND_MESSAGE	= 0x0020
RECEIVER_FIFO_EMPTY	= 0x0010
RECEIVER_FIFO_FULL	= 0x0008
EMITTER_FIFO_EMPTY	= 0x0004
EMITTER_FIFO_FULL	= 0x0002
RUN	= 0x0001

3.4. Glossary

void	**SPACClearAlarms**	(SPACMaster* Card);
SPACMaster*	**SPACDeclareBoard**	(u_short CrateNumber, u_short AddressModifier, u_short BoardAddress);
u_short	**SPACIdentifier**	(SPACMaster* Card);
u_short	**SPACMasterStatus**	(SPACMaster* Card);
u_char*	**SPACReadFIFO**	(SPACMaster* Card, u_char Address, u_char SubAddress, u_short Size);
u_char*	**SPACReadRAM**	(SPACMaster* Card, u_char Address, u_char SubAddress, u_short Size, u_short RAMStartAddress);
u_long	**SPACReadRegister**	(SPACMaster* Card, u_char Address, u_char SubAddress, u_char Size);
void	**SPACReloadEmitterFIFO**	(SPACMaster* Card);
void	**SPACReset**	(SPACMaster* Card);
void	**SPACRun**	(SPACMaster* Card, u_char State);
void	**SPACWriteFIFO**	(SPACmaster* Card, u_char Address, u_char SubAddress, u_char* Data, u_short Size);
void	**SPACWriteRAM**	(SPACMaster* Card, u_char Address, u_char SubAddress, u_char* Data, u_short Size, u_short RAMStartAddress);
void	**SPACWriteRegister**	(SPACMaster* Card, u_char Address, u_char SubAddress, u_long Data, u_short Size);
void	**VMEInitialize**	(SPACMaster * Card);
u_short	**VMERead**	(SPACMaster *Card, u_short Address);
void	**VMEReadBlock**	(SPACMaster *Card, u_short Address, u_short *Buffer, u_short Size);
void	**VMEWrite**	(SPACMaster *Card, u_short Address, u_short Value);
void	**VMEWriteBlock**	(SPACMaster *Card, u_short Address, u_short *Buffer, u_short Size);

3.5. A typical program

```
#include "stdio.h"
#include "stdlib.h"
#include "spac.h"

#define CRATE_NUM      0x00
#define AM             0x39
#define CARD_ADD       0x01
#define PAGE_NUM       15
#define SLAVE_ADD      0x15
#define NTA_SUBADD     0x00
#define FIFO_SUBADD    0x02
#define RAM_SUBADD     0x03

void main()
{
      u_long regGet, regPut = 0xBABA;
      u_char* fifoPut, fifoGet, ramPut, ramGet;
      u_long i, size=0x50;
      SPACMaster* Master;
      fifoPut = (u_char *) calloc (size, sizeof(u_char));
      fifoGet = (u_char *) calloc (size, sizeof(u_char));
      ramPut = (u_char *) calloc (size, sizeof(u_char));
      ramGet = (u_char *) calloc (size, sizeof(u_char));
```

```
	Master = SPACDeclareBoard (CRATE_NUM, AM, CARD_ADD);
	SPACInitialize(Master);

	SPACWriteRegister(Master,SLAVE_ADD , NTA_SUBADD, regPut, 2);
	regGet = SPACReadRegister(Master, SLAVE_ADD, NTA_SUBADD, 2);
	printf("NTA = 0x%X\n", regGet);

	for (i=0;i<size;i++)
		fifoPut[i]= (u_char) i;
	SPACWriteFIFO (Master, SLAVE_ADD, FIFO_SUBADD, fifoPut, size);
	fifoGet=SPACReadFIFO (Master, SLAVE_ADD, FIFO_SUBADD, size);

	SPACWriteRAM (Master, SLAVE_ADD, RAM_SUBADD, ramPut, size, 0x0000);
	printf("ecrite= ");
	for (i=0;i<size;i++)
	{
		ramPut[i]= (u_char) i;
		printf("%X ",ramPut[i]);
	}
	printf("\n");
	ramGet=SPACReadRAM (Master, SLAVE_ADD, RAM_SUBADD, size, 0x0000);
	printf("relue= ");
	for (i=0;i<size;i++)
		if (ramPut[i]==ramGet[i])
			printf("%X ", ramGet[i]);
		else
			printf("*%X* ",ramGet[i]);
	printf("\n");
}
```