Chapter 1

Serial interface

In order to connect and control the Common Readout Unit (CRU) from a user Personal Computer (PC), some sort of serial communication between the user PC and the Field-Programmable Gate Array (FPGA) is needed. The purpose of the serial link is mainly to monitor the Gigabit Transceiver (GBT) and manipulate the control signals. Because of this, the speed requirements of the link is not a crucial matter. The FPGA unit used in this thesis (section **??**) has (although sparse) various forms of communication standards readily available. The following sections starts off with an evaluation of the different communication standards available on the FPGA, and continues with a description of the final implementation of the serial communication setup.

1.1 Readily available standards

When choosing the serial communication between the FPGA and the PC, factors like physical compatibility, implementation, availability and complexity were taken into consideration. The FPGA unit has the following communication capabilities readily available: Peripheral Component Interconnect (PCI)-express, Ethernet, JTAG UART (through the USB Blaster II), and an SDI-transceiver. The following short sections describe advantages and disadvantages of using one of the systems mentioned above in context with the thesis.

PCI-express

The PCI-express connection requires the FPGA to be directly mounted on the motherboard of the PC, which is in this situation impractical and not necessary for a simple communication link. This option also limits the compatibility with some FPGA boards and PCs that do not have a PCI-express connection. It does, however, enable for very fast data transmission and removes potential noise generated by using external cabling.

Ethernet

The ethernet connection is integrated in most FPGA boards, but requires layers of protocols in order to communicate (no direct serial communication). While it is possible to send serial data over ethernet, it would require a serial-to-ethernet adapter [?]; It would just be easier to send serial data directly from the FPGA-pins using an USB-to-RS232 adapter-cable (see section 1.2). The upside is that an ethernet connection offers long distances between the PC and FPGA, either through networking or long cabling. Using ethernet for communication only requires a known IP-address between the devices for connection and transmission.

SDI-transceiver

The SDI-transceiver is meant for audio/video transmission and uses BNC-connectors for this task. It therefore requires special audio/video equipment on the PC side for connection and transmission, which is not necessary for a simple communication link.

Altera JTAG

Serial communication through the Altera JTAG UART IP is possible through the only USB-connection on the board; the USB Blaster II, which is used to program the FPGA. This requires the implementation of the Nios II soft processor. A soft processor is a microprocessor implemented into the FPGA with the help of logic synthesis only. While it is also possible to use a Hard Processor System (HPS), only a few FPGA-boards (SoC-boards) comes with one implemented. There is no support for HPS on the FPGA board used in this thesis. A Nios II-extension for the Eclipse IDE in combination with the C programming language is commonly used to program the soft processor. Using this approach not only limits the user to send and receive data through a dedicated Altera System Console [?], the Nios II also occupies a large portion of the FPGA.

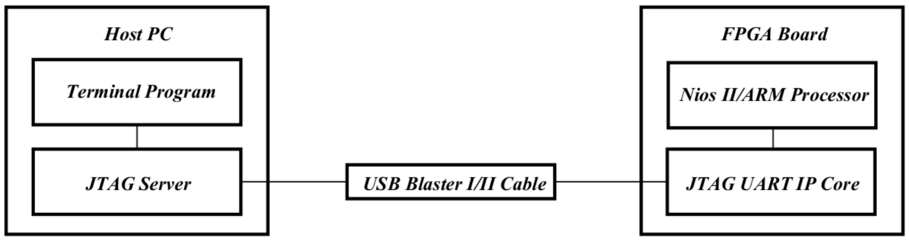

Figure 1.1: JTAG UART communication link between the host PC and the FPGA board [? , Figure 1].

Attempts were made to implement the Nios II and write a simple send- and receive routine in C through the Altera JTAG, but was quickly abandoned due to issues with debugging using the

Eclipse Nios II IDE. The most serious debugging issue was the fact that the compiler continued displaying errors after the errors were corrected. Even after the actual bugged function or line of code were deleted completely, the compiler would still complain on the same errors as before it was deleted. This made the dedicated Nios II compiler unreliable, and the code impossible to debug. Not only did this approach make it a whole lot more complicated, but it was also more error prone and permitted the use of user defined software for communication with the FPGA.

1.2 User defined communication

In addition to the communication standards described above, which in turn requires specific cable- and socket types for connection, it is possible to connect the transmit and receive signals directly to the FPGA pinout. The only requirement is that the given FPGA has unoccupied transmitter and receiver signal pins available for the user; either through a HSMC-port (with the help of an additional GPIO-extension board), or through a prototype area or header.[1]

The FPGA unit used in this thesis has no dedicated prototyping area for external signal connection. It instead comes with an GPIO-extension board that connects to one of the two onboard HSMC-ports.[2] However, if removing the on-mounted LCD-display, it is possible to use the exposed header for signaling. The header (J10 in the board schematic) is connected to a transmitter- and receiver pair (both running on a voltage of 5 V), several 5 V output pins and a ground pin. By using the transmitter- and receiver pair from the available header, it is possible to implement a type of asynchronous serial protocol into the FPGA.[3]

1.3 Duplex systems

Common to all communication systems considered is that they are all duplex systems, which simply means two connected devices that can both transmit and receive signals. While *full-duplex* enables both devices to transmit and receive simultaneously (like a telephone), *half-duplex* only enables one device to transmit at a time (like a walkie-talkie). Common to both the duplex systems is that they have two communication channels.

[1]The user has to assign the available pins to the associated transmitter and receive signals in the *Pin Planner* program, as part of Quartus II suite.

[2]By using both the HSMC-ports on the FPGA, the GPIO-Printed Circuit Board (PCB) could not fit properly side-by-side with the HSMC-to-VLDB PCB (section **??**) because the latter PCB is a bit to wide.

[3]Most FPGA developement boards comes with a built-in Universal Asynchronous Receiver/Transmitter (UART)-to-USB interface, which connects directly to the USB-port of the PC. The particular FPGA used in this thesis had no UART already built in.

1.4 Choosing communication protocol

Perhaps the most well-known and supported serial communications protocol out there is the RS-232 standard. It supports both synchronous and asynchronous transmission, and only requires a single transmit- and receiver pair (if excluding the data control signals). It is compatible over a wide range of voltage levels, and can be connected directly to the serial port of a PC (if one is available) or through a USB-to-RS232 adapter. The RS-232-standard was chosen mainly because it only requires two wires (one for transmitting and the other for receiving). It is supported by most Operating Systems (OSs) used today, and can be easily implemented using available C-libraries.

The standard will be used for asynchronous transmission of data between the PC and FPGA. A simple UART with a byte-decoder will be implemented on the FPGA-side, while a dedicated C-program with access to the COM port will be implemented on the PC-side. A USB-to-RS232 adapter with a voltage converter will be used as the connection between the FPGA and the PC.

1.5 Transmission protocol: RS-232

RS-232 (RS-232) standard is a transmission protocol for full duplex, serial transmitting and receiving of data. The standard defines electrical characteristics and timing of signals, the "meaning" of the signal, and also physical size and pin-out for connectors. For communications to work properly, both the Data Terminal Equipment (DTE) (the PC) and the Data Communication Equipment (DCE) (in this case, the FPGA) needs to agree on the same data-packet setting, i.e the baud rate, the number of data bits, any additional parity bit and the number of stop bits.

A typical RS-232 data-packet consists of a start bit, followed by 5, 7 or 8 data bits; 1 parity bit, for error checking; and 1 or 2 stop bits, indicating that the transmission is done. The start bit is typically a logical low and the stop bit(s) high (this is usually the case, but can be system dependent). The transmission line remains high until the start bit pulls it down low, and the data transfer begins until the stop bit is reached. The line is then kept high until a new start bit pulls it low again for a new byte to be transfered. Data-packets are often described in the form: $19200 - 8 - N - 1$, which simply means 19200 bit/s, 8 *data bits*, *No parity* and 1 *stop bit*. Figure 1.2 demonstrates a typical RS-232 signal.

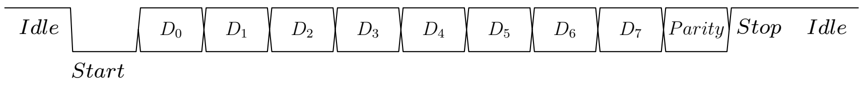

Figure 1.2: Example of an RS-232 signal with 8 data bits, 1 parity bit (Odd, Even or No parity) and 1 stop bit.

Chapter 2

Hardware design on the FPGA side

2.1 Specification

The implemented hardware logic is to contain the following specifications:

- Communication with the user PC using RS-232.
  - Receive requests sent from the user PC. The requests are reserved byte-codes in which the byte-decoder translates to read or write commands.
  - Send out data from the GBT control-register to the user PC when told. The address of the data must be specified and sent from the user PC.
  - Manage to send and receive data at a reasonable speed.
- Read or write to the GBT control-register.

2.2 Hardware Components

The below sections gives a brief description of each part of the design.
The UART itself is based on the UART-design by Pong P. Chu, found in chapter 7 from his book *FPGA Prototyping By VHDL Examples* [**?**]. The Baud Rate Generator is borrowed from the *Uart2Bus* VHDL-design by Arild Velure. The UART-decoder was written in conjunction with the C-program on the PC side, and is the one component which ties the communication together.

The end result forms a design that uses an UART to receive and transmit $19200-8-N-1$ RS-232 data-bytes. The received bytes are stored in a FIFO until treated by the UART-decoder. The "decoder" translates the received bytes into requests, i.e read or write requests, and manipulates or sends out data-bits from the GBT-register according to the received requests. The UART itself is optimized for 19200 bit/s, but has been tested to work at speeds up to 57600 bit/s.

2.2.1 UART

For communication with the user PC, a simple UART was implemented into the FPGA. Simply put, an UART is a circuit that transmits and receives parallel data through a serial line [?]. It uses the concept of oversampling to synchronize with the incoming data. This involves using a sample-clock which is 16 times faster than the transmitted/received data.

The UART-design is divided into five parts:

- A Receiver that receives the serial data and reassembles it into parallel data.
- A Transmitter that sends parallel data bit by bit out the serial line.
- A Baud Rate Generator that generates the right amount of ticks relative to the baud rate and global clock.
- Two FIFO-registers connected to the transmitter and receiver to shift the data in or out.

2.2.2 UART oversampling and the Baud Rate Generator

To obtain an accurate sampling of the received signal, an asynchronous system like the UART uses what is referred to as oversampling.

With a typical rate of 16 times the baud rate, the receiver listens for the line to go from idle to the first start bit. When pulled low or high (depending on the system), a counter starts counting from 0 to 7. When the counter reaches 7, the received signal is roughly in the middle of the start bit. By sampling the bit in the middle of its time frame, the receiver avoids the noise and ringing that are generated whenever a serial bit changes [?]. When in the middle of the start bit, the counter needs to tick 16 times before it reaches the middle of the first data bit, the Least Significant Bit (LSB). The LSB can now be sampled, and the procedure is repeated $N - 1$ times until reaching the last data bit, the MSB. If there is a parity bit, the same procedure is repeated one more time to retrieve it. After retrieving all the data bits, the same procedure is used one last time to sample the M stop bits at the end of the signal. After this, the line is held high until a new start bit arrives.
To achieve a sampling rate of 16 times the baud rate, a Baud Rate Generator module is implemented into the design. The module generates a one-clock-cycle tick once every $\frac{clock}{16\times baud\ rate}$ clock cycles. For a baud rate of 19200 $\mathrm{bit/s}$ and a clock of 50 MHz, there must be a one-clock-cycle tick once every 163 clock cycle. This is achieved by counting in certain steps given by the formula below:

$$Baud\ frequency = \frac{16 \times baud\ rate}{GCD(clock, 16 \times baud\ rate)} \tag{2.1}$$

, where baud frequency is the count steps, GCD is the greatest common divisor between the global clock and the baud rate times 16 [?].

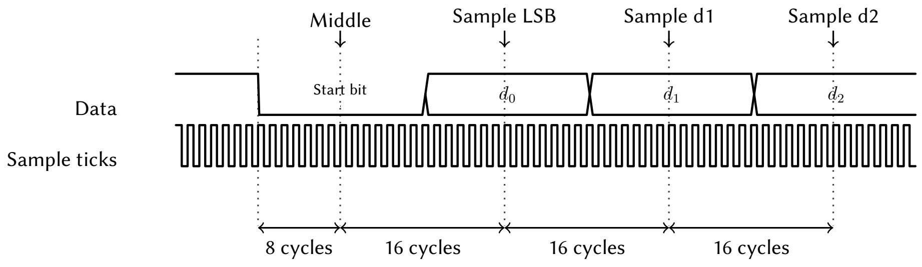

Figure 2.1: UART receive synchronisation and data sampling points with 16 times the sampling rate.

For a baud rate of 19200 $\mathrm{bit/s}$ and a clock of 50 MHz, the counter must count in steps of 576 per clock cycle.
Once the counter reaches a given baud limit, the counter resets and the tick-signal is pulled high. After one clock cycle, the tick-signal is pulled low and the counter starts to count upwards again. The baud limit is given by:

$$Baud\ limit = \frac{clock}{GCD(clock, 16 \times baud\ rate)} - baud\ frequency \tag{2.2}$$

, where clock is the global clock of the system and GCD is the greatest common divisor between the global clock and the baud rate times 16 [**?**].
For a baud rate of 115200 $\mathrm{bit/s}$ and a clock of 50 MHz, the counter must count in steps of 576 up to the baud limit of 15049, before pulling the tick-signal high and start over again.

2.2.3 UART Receiver

The receiver is essentially a finite state machine, divided into four states: the idle-, start-, data- and stop state. It uses the Start and Stop bits to reset the state machine in an attempt to synchronize the clock phase to the incoming signal. For this, the receiver contains three registers: the s- and n registers for counting, and a b register for data storing. The s-register keeps track of the sample ticks and n-register the number of data bits sampled. There are two constants defined for the receiver: the C_DBIT constant, which indicates the number of data bits; and the C_SB_TICK constant, which indicates how many ticks that is required for the stop bit(s) (16 ticks for 1 stop bit).

Idle state

Starting with the idle state: given that there is not already a signal being received and sampled (i.e $rx\_done\_tick =' 1'$), the receiver waits for the rx-signal to go low (i.e detecting a start signal). The s-register then resets and the state machine goes to the next state: the start state.

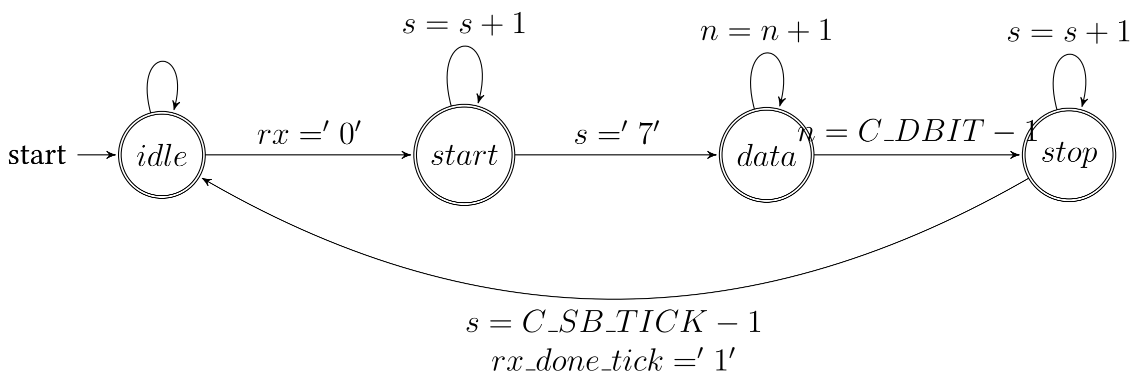

Figure 2.2: UART receiver state machine.

Start state

When in the start state, the ticks generated by the baud rate generator clocks the s-registers and waits for it to count up to 7 (i.e in the middle of the start-bit). It then resets the s- and n-registers and sets the next state to the data state.

Data state

Since the received signal so far is the middle of the start bit, the s-register must count up to 15 before reaching the middle of the data bit so that the b-register can sample the data. Each time the s-register reaches 15, the b-register shifts in the rx by 1 bit while the n-register increments by 1, keeping track of the bit width. When the n-register reaches a count equal to the C_DBIT constant minus 1, the transmission is at its end, and the state machine shift to the last state: the stop state.

Stop state

The stop state uses the sample ticks in conjunction with the s-register to count the stop bit(s) duration by using the $C\_SB\_TICK - 1$ as the upper count limit. When done, the rx_done_tick signal is set to 1 and the state machine shifts back to the idle state. We have now successfully received a serially transmitted byte.

2.2.4 UART Transmitter

The UART transmitter has a similar design to that of the receiver; it uses the same state machine structure, but for the purpose of shifting out data. In addition to the s-, n-, and b-registers used for counting, the transmitter contains a din-register for input parallel data and a 1 bit tx-register for shifting out the data. Connected to the tx-register is a tx output signal. To prevent multiple clocks, the baud rate generator is also used to clock the transmitter. For the transmitter to be properly synchronized with the receiver, it instead uses the counter registers to slow down the operation 16 times. This is because there is no oversampling involved in transmitting a signal.

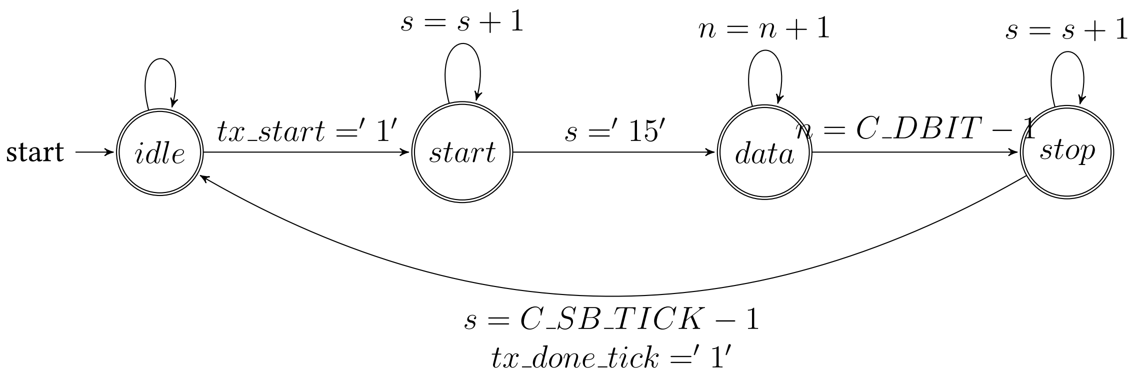

Figure 2.3: UART transmitter state machine, similar to that of the receiver.

Idle state

When in the idle state, the tx-register is held high for idle line and the b-register is connected to the din register for data input. The state machine remains in the idle state until the tx_start signal changes from low to high. It then shift to the start state to begin the transmission of the data stored in the b-register.

Start state

When in the start state, the tx-register is held low for the start bit to be transmitted out. When the s-register reached the count of 15, the start bit is transmitted and the state machine shifts to the data state to further transmit the actual data bits.

Data state

When in the data state, the tx-register is set to the first bit of the b-register, while the b-register shift its data using a simple right shift operation every 16 clock ticks (s-register reaches 15). To keep track of how many data bits that have been transmitted, the n-register increments by 1 at the same rate as the b-register shifts the data (same as for the receiver). When reaching $C\_DBIT - 1$, the data bits is finished transmitting out and the state machine shifts to the stop state.

Stop state

When in the stop state, the tx-register is held high indicating a stop bit is being transmitted. When the s-register reaches a count equal $C\_SBIT - 1$, the stop bits are finished transmitting. The tx_done_tick signal is set to high and the state machine shifts back to the idle state, ready to transmit the next data byte.

2.2.5 FIFO buffers

Both the UART transmitter and receiver has FIFO-buffers that stores the incoming data sequentially in a "First In, First Out" manner. On the receiver side, the incoming data is stored until it gets the read-out signal (rd_uart goes high). It then places the first stored byte on the (r_data)-line for one clock cycle. As long as the read-out signal remains high and there is bytes stored, the FIFO will spew out data on the (r_data)-line with the rising edge of the clock. The rx_empty signal indicates whether there is one or more bytes stored in the FIFO. On the transmitter side, when data from the FPGA is written into the FIFO, it sends a signal to the transmitter to start shifting out the data stored in the buffer, oldest byte first. Having FIFOs to store data between the UART and the rest of the FPGA logic is necessary to prevent data loss, as the FPGA logic operates at a much higher clock speeds than the UART can transmit and receive data. If a FIFO gets filled up, no new data will be written to it until data is read out of it, freeing one slot.

2.2.6 UART decoder

The UART decoder is a state machine connected to the other end of the UART receiver- and transmitter FIFOs. It reads out the received bytes and does tasks according to the order and value of the bytes. Starting at the idle state, the decoder waits for a request byte from the fifo followed by an address byte. The request byte can for instance be a read or a write. The following small sections describes the different decoder states.

Idle state

Being initially in the idle state, the decoder is triggered by the rx_empty signal coming from the rx-fifo, which is held low if there is at least one byte stored in the rx-fifo. If rx_empty is low, the decoder sends a read signal (rd_uart), telling the rx-fifo to place the next in-line byte on the read-bus (r_data), and in the same delta-time assigns the read-bus to a new register ($b\_reg(0)$) for temporary storage. The decoder then goes to a read state.

Read1 state

In the read1 state, the first temporary stored byte is analyzed. The byte must have a value that is equal to one of three predefined constants, called requests (See $uart\_gbt\_pkg.vhdl$). This first byte is interpreted as a read- or write-request, sent from the client PC through the serial connection. If the request-byte is equal to a ”read” request, a ”write 0” request or a ”write 1” request, the decoder goes on to a wait state. If the byte is not a request-value, the state machine returns back to idle and reset the registers.

Wait1 state

In the wait1 state, the decoder waits for the next arriving byte after the first one. The wait state does the same as the idle state, i.e waits for rx_empty to become low before reading out the next byte from the rx-fifo. To prevent the state machine from getting stuck in the wait state (if an address byte does not arrive in time), an additional count-register, triggered by the baud generator ticks, counts up to a predefined value, $C\_TIMEOUT$. $C\_TIMEOUT$ is defined to be a reasonable larger value than the pre-estimated time it takes for the next byte to arrive, i.e $16\ ticks/\mathrm{bit} \times 10\ bits$. If rx_empty is not triggered low before the count-register has finished counting, the state machine resets to idle. If the next byte arrives before timeout, the decoder stores it and goes to the final state.

Read2 state

In the read2 state, the value of the first byte, the request-byte, decides if the decoder should perform a read or a write operation on the gbt-register. The value of the last received byte is treated as an address to the data of the gbt-register the client wants to read or write to.

Chapter 3

Software on the PC side

The program was written with the goal of one day replacing the Quartus-bound In-System-Source-And-Probe Editor, which is used today to access the GBT control-signals, and instead introduce a cross-platform, open-source solution.
To access the serial port on the PC, the software uses free and cross-platform C-libraries.
The C programming language was chosen for this task mainly because of previous experience with the language, but also because of eager to learn to use the language more professionally.

3.1 Specification

The program is to contain the following specifications:

- The ability to send and receive bytes from the UART on the FPGA.
- An user interface that makes it easy to control and monitor the FPGA, i.e a command console.
- Cross-platform. This was not a critical requirement, but was added later because of the language the program was written in and also the already cross-platform libraries used.

3.2 Non-standard libraries

The serial communications software was developed with the help of a few non-standard libraries; non-standard meaning that the libraries are not a part of the C standard libraries. In addition, a library dedicated to hold the GBT control-signal information was written. The following sections gives brief descriptions of each of these libraries and the associated functions used in the serial communications software:

3.2.1 RS232

The RS232 library (*rs232.h* and *rs232.c*) is a cross-platform C-library for sending and receiving bytes from the COM port of a PC. It was written by Teunis van Beelen, and is licensed under the ”GPL version 3” licence [?]. The library compiles with GCC on Linux and Mingw-w64 on Windows. By specifying baud rate, the name of the relevant COM port, and the transmission mode (8N1 is standard), the library provides access to the COM port and allows for both reading and writing to it. For more information about the library and functions, visit `http://www.teuniz.net/RS-232/#`.

Associated functions

```
int RS232_OpenComport(int comport_number, int baudrate, const char * mode)
```

Opens the COM port. The user must specify a COM port number, a baudrate number and the transmission mode (see list of valid inputs in *rs232.c*). It is important that the user grants superuser/administration priveliges to the program, or it might not be able to open the COM port or change the baudrate correctly. The function returns 1 if it does not succeed in opening the COM port.

```
int RS232_PollComport(int comport_number, unsigned char *buf, int size)
```

Returns the amount of bytes (in integers) received from the COM port. The received bytes are stored in a buffer and pointed to by *buf*. One must specify the *size* of the buf pointer. It is recommended to call this function routinely from a timer.

```
int RS232_SendBuf(int comport_number, unsigned char *buf,int size)
```

Sends a buffer of bytes via the COM port. The *buf* pointer must point to an array of bytes, and *size* must specify the correct size of the array pointed to by the *buf*.

```
void RS232_CloseComport(int comport_number)
```

Closes the COM port. To prevent errors, it is important to disable any timers that calls COM port related functions before closing the COM port.

3.2.2 Timer

The Timer library (*timer.h* and *timer.c*) is a library for handling timers in a C environment. It was written by Teunis van Beelen, and is licensed under the ”GPL version 3” licence [?]. The accuracy of the timer is system dependent. It supports a resolution down to $1\ \mathrm{ms}$ on the Windows platform and a resolution down to $1\ \mu\mathrm{s}$ on the Linux platform (though, in real-life situations, the timer might not be as accurate). The timer function is used with the *RS232_PollComport*-function in mind (see function definition above). For more information about the library and functions, visit `http://www.teuniz.net/RS-232/#`.

Associated functions

```
int start_timer(int milliSeconds, void (*)(void))
```

Starts the timer. At a given time interval of *milliSeconds* (milliseconds in Windows, microseconds in Linux), it calls a given function. The call-function must be a void function with no inputs (just like a standard main function without the argument calls). The *start_timer* function are called only ones in main, and repeats calling the given function every *milliSeconds* interval until the *stop_timer* is called.

```
int stop_timer( void )
```

Stops the timer if *start_timer* has previously been called. If the timer routinely calls RS232-functions, *stop_timer* must be called before closing the COM port at the end of the program to prevent errors.

3.2.3 Signals

The Signals library (*signals.h* and *signals.c*) was written with the GBT control-signals in mind. It features methods for storing and manipulating the control-signal information, and in some extent encapsulates the information by partly hiding the information from the user. This is done by defining the main information-structure inside the *signals.c*-file, instead of defining it in the header file (and thus exposing it to the main program). A dedicated pointer is used to point to the structures namespace, and the functions defined in the header file calls the pointer as input instead of the structure itself. The main program only have access to what is defined in the header, thus limiting the user to only have access to the structure through dedicated library functions.

Associated structures

```
typedef struct {
  char *name;
  char *index;
  Byte type : 1;
} Type;

 struct _Signal {
  Type type;
  int i;
  Byte data;
};
```

The *Type*-structure contains the signal name and index in string-form, and a 1-bit type variable that defines the signal as either a probe (0) or a switch (1). *_Signal*-structure is the main structure and contains the *Type*-structure along with a ”real” index, i, which is the actual data-address used to access the correct register-address on the FPGA; and the data byte, data, which stores the data bit of the signal. If the structure is defined as a probe, the data can only be read from the FPGA. If it is defined as a switch, on the other hand, it can be both read from the FPGA or the main program can write data to the FPGA.

Associated functions

```
Signal Signal_New (void)
```

Assigns a new pointer to the *_Signal*-structure and allocates enough memory.

```
Signal Signal_Init(char *index, char *name, Byte data)
```

Uses *Signal_New* to define a new signal, and in addition assigns values to the signal variables. The *index* string must contain either an *’S’* or a *’P’* character followed by two number-characters. This is to indicate that the signal is either a probe or a switch with a given register address, for example: ”P00”, ”S35”.
The first letter is used to assign the *type*-variable and the two number-characters are converted to an integer and assigned to the ”real” index, i, of the structure. The *name* string should contain a descriptive name of the signal, for example: ”TX_FRAMECLK PHASE ALIGNER - PLL LOCKED”.

```
void Signal_InitFromFile(Signal s[], int width, char *filename)
```

Uses *Signal_Init* to define a signal-array using information given from a file. The array has a width given by the *width*-variable. Refer to *signal_probe.txt* or *signal_switch.txt* for examples on how to set up a text-file. In the program, the *Signal_InitFromFile*-function is used twice to define two signal-arrays; one for the *Switches* and one for the *Probes* of the GBT control-signals. Defining two signal-arrays makes it more convenient when separating what is writeable and what is read-only.

```
void Signal_Free(Signal s)
```

Frees the given signal-pointer and all the associated variables.

```
void Signal_FreeArray(Signal *s, int n)
```

Uses *Signal_Free* in a loop to free up an array of signal-pointers.

```
int Signal_PrintSet(WINDOW *win, int *gy, Signal s[], int width)
```

Prints out all printable signal information of a signal-array to a *curses*-window (see section about ncurses) with a row-position given by *gy*. *gy* is defined as a pointer because it reports back to the program on which line in the window the information is printed on.

```
void Signal_setData(Signal s, Byte value)
Byte Signal_getData(Signal s)
```

Functions related to reading and writing to the data-variable, data, of a given signal.

```
void Signal_setIndex(Signal s, int i)
int Signal_getIndex(Signal s)
```

Functions related to reading and writing to the ”real” index-variable, *i*, of a given signal.

```
void Signal_setType(Signal s, Byte type, char *name, char *index)
void Signal_getType(Signal s, Byte *type, char *name, char *index)
```

Functions related to reading and writing to the *Type*-structure of a given signal.

```
void Signal_setTypeType(Signal s, Byte type)
Byte Signal_getTypeType(Signal s)
```

Functions related to reading and writing to the type-variable, *type*, of the *Type*-structure of a given signal.

```
void Signal_setTypeName(Signal s, char *name)
char *Signal_getTypeName(Signal s)
```

Functions related to reading and writing to the name string, *name*, of the *Type*-structure of a given signal.

```
void Signal_setTypeIndex(Signal s, char *name)
char *Signal_getTypeIndex(Signal s)
```

Functions related to reading and writing to the index string, *index*, of the *Type*-structure of a given signal.