# Chapter 1

# PC to CRU control interface

In order to connect and control the Common Readout Unit (CRU) from a user 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 in this thesis is mainly to monitor and manipulate the Gigabit Transceiver (GBT) control signals. Because of this, the speed requirements of the link is not a crucial matter.
Figure 1.1 illustrates the different blocks that make up the serial interface: The interface on the PC side consists of a terminal-like interface that lets the user type in requests that enables the user to read and/or write to the GBT control signals via the link. These requests then gets translated into byte-codes and sent out via the transmitter of the COM port to the Universal Asynchronous Receiver/Transmitter (UART) on the FPGA side. Here, the bytes are stored in a buffer before they get interpreted by a decoder. Depending on the user-request, the decoder can write to a GBT-switch or read from a GBT-probe.

## 1.1 Readily Available Standards

The FPGA board used in this thesis (section **??**) has various forms of communication standards readily available. The following sections starts off with an evaluation of the

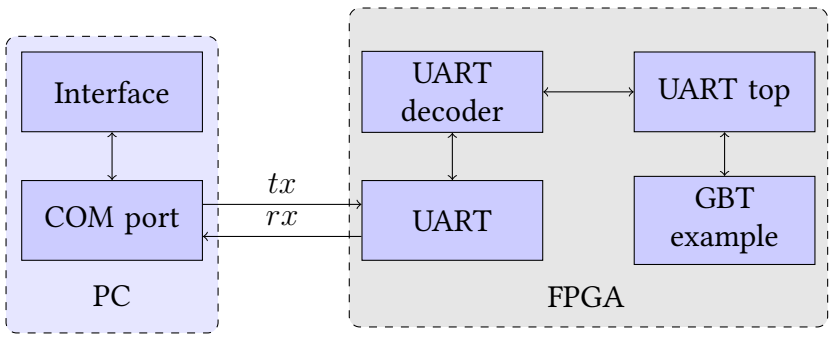


**Figure 1.1:** Simplified diagram over the PC to CRU serial interface.

different communication standards available on the FPGA, and continues with a description of the final implementation of the serial communication software and hardware.

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 board 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 standards 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 available. 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 primary used for programming 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.

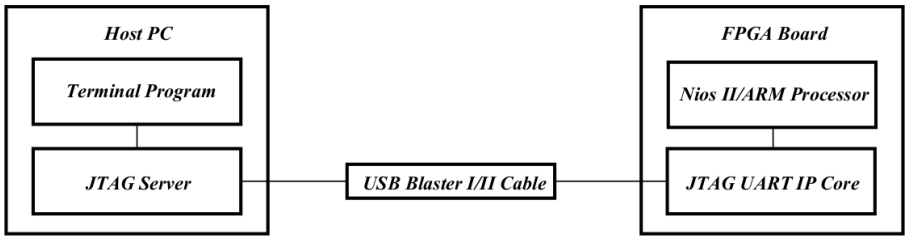


**Figure 1.2:** JTAG UART communication link between the host PC and the FPGA board [**?** , Figure 1].

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 board 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 on-board 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

[1]The user has to assign the available pins to the associated transmitter and receive signals in the *Pin Planner* program, as part of the 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 (see chapter **??**) because the latter PCB is a bit to wide.

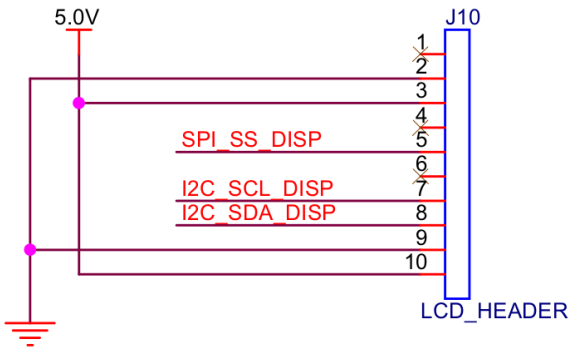


**Figure 1.3:** J10 header from the Cyclone V GT board schematic. The SCL and SDA signals can be used as single-ended transmitter and receiver signals.

board schematic, as shown in figure 1.3) 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 on the FPGA, it is possible to implement a type of asynchronous serial protocol. Most FPGA developement boards, including the Terasic Cyclone GX board (see section **??**), comes with a built-in UART-to-USB interface. This allows you to connect the board directly to the USB-port of the PC as a serial connection. The FPGA board used in this thesis has no integrated UART interface, so this has to be implemented manually.

## 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.

## 1.4 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, which are not crucial). 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. The following chapters describe the implementation of the serial interface, first on the FPGA side (1.5) and then on the PC side (1.6).

## 1.5 Hardware Design on the FPGA Side

The hardware design in the FPGA is responsible for treating incoming requests made by the user PC. It will essentially become a module to connect with the GBT example design, were it replaces the In-System Source And Probe Editor (ISSP)-module that is used today. The module must have access to the GBT control signal register, and this is done by re-connecting the probe and source signals, that is now connected to the ISSP, to the module. The module is mostly completed; what remains is to implement it in the GBT example design, connecting the probe and source signals and effectively replacing the ISSP module.

### 1.5.1 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.

### 1.5.2 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 a read-request if $0xDD$ or a write-request if $0xEE$ (write value 0) or $0xFF$ (write value 1), and manipulates or sends out data-bits from the GBT-register according to the received requests. The UART itself is optimized for $19200\ \mathrm{bit/s}$, but has been tested to work at speeds up to $57600\ \mathrm{bit/s}$.

### 1.5.3 UART

For communication with the user PC, a simple UART was implemented in 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.

### 1.5.4 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. This is achieved by counting in certain steps given by the formula below:

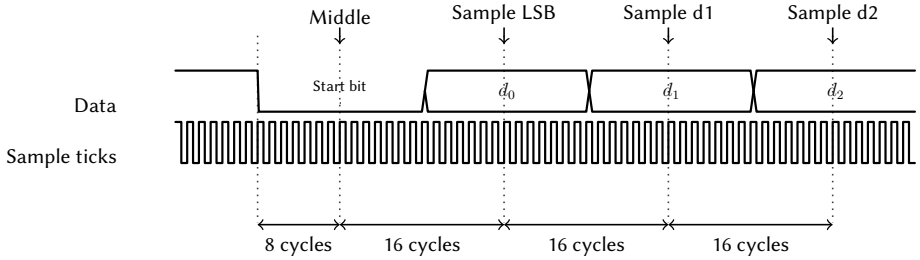


**Figure 1.4:** UART receive synchronisation and data sampling points with 16 times the sampling rate.

$$Baud\ frequency = \frac{16 \times baud\ rate}{GCD(clock, 16 \times baud\ rate)} \tag{1.1}$$

, where baud frequency is the count steps, and *GCD* is the greatest common divisor between the global clock and the baud rate times 16 [**?** ].
For a baud rate of $19200\ \mathrm{bit/s}$ and a clock of $50\ \mathrm{MHz}$, the counter must count in steps of $96$ 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:

$$Baud\ limit = \frac{clock}{GCD(clock, 16 \times baud\ rate)} - baud\ frequency \tag{1.2}$$

, 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 $19200\ \mathrm{bit/s}$ and a clock of $50\ \mathrm{MHz}$, the counter must count in steps of $96$ up to the baud limit of $15565$, before pulling the tick-signal high and start over again.

### 1.5.5 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).

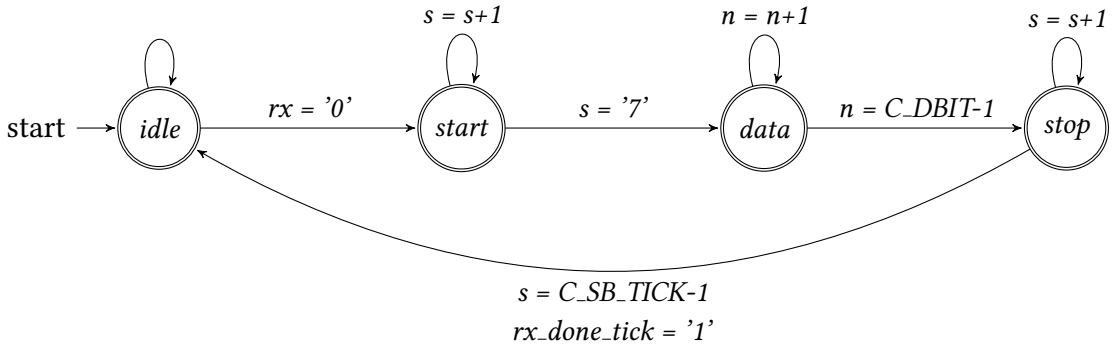


**Figure 1.5:** UART receiver state machine.

### 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.

### 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.

### 1.5.6 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.

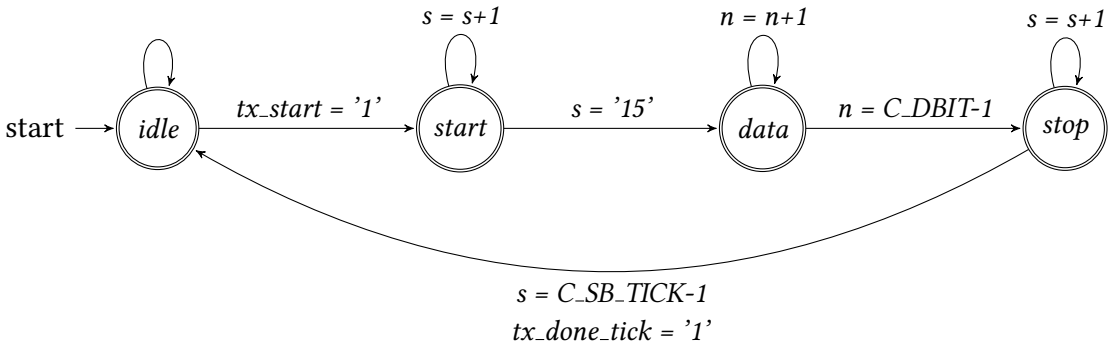


**Figure 1.6:** UART transmitter state machine, similar to that of the receiver.

#### 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.

### 1.5.7 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.

### 1.5.8 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.

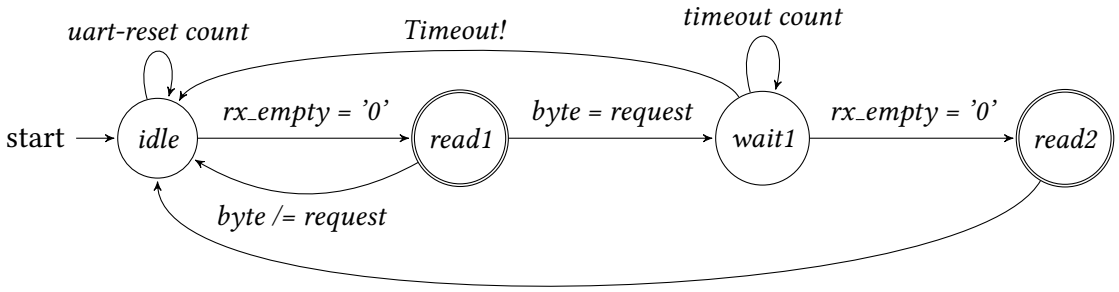


**Figure 1.7:** UART decoder state machine.

### 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 $16\ ticks/\text{bit} \times 10\ bits$. 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.

## 1.6 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 (see sections below).

The C programming language was chosen for this task mainly because of previous experience with the language, but also because of desire to learn to use the language more professionally.

### 1.6.1 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.

### 1.6.2 Software Structure and Flowchart

The software is divided into two modules; one module for sending and receiving data, to and from the FPGA; and one module that acts as the actual software interface. Both these modules were intended to be merged together into one program, both because of time constraints were left separated. The below sections describes the inner workings of each of the two modules, and provides flowcharts to further illustrate the behaviour.

#### Interface module

Figure 1.8 illustrates the behaviour of the interface module. The main loop start by checking for any changes in screen size of the main terminal window. This allows the user to freely change the window size, although full screen is preferred. If a resize has occurred, the program stores the size-parameter and uses it to further resize and properly align all the internal windows (”Command History”, ”Commandline” and ”Signals”) using *wresize*, and clears the content using *wclear*. The content will later be redrawn with coordinates aligned according to the new size.

With ncurses, by using *getch* in conjunction with *nodelay*, one can check for keypresses without having to pause the entire program (see **??**). If a key is pressed, the

program analyses the input: If a character is typed, it is appended to a command-string and displayed in the ”Commandline” window.

If it is a return-character ('\n'), the program processes the command-string and compares it up against a table of legal commands. The legal commands has associated functions that executes if there is a match between the input and an entry in the table. For it to be a match, the first word of the input-string must be a legal command followed by legal arguments. To see the usage of a command, one can just type in the command without any arguments. It is also possible to type in just a letter or a partial word to print out possible commands that contain the letter or partial word typed in. Legal commands include: *write*, which writes a bit value to one or more of the switch-signals; *read*, which sends read requests for one or more signals to the FPGA; *open*, which opens the COM port if it is not already open; *close*, which closes the comport if it is open; *status*, which prints out information about the COM port and its current status; and *exit*, which closes the COM port and exits the program.

If either return or a character has been typed in, the program checks the input for a special key. Special keys include: ”Page Up” and ”Page Down”, that selects which signals to display in the ”Signals” window (probes, switches or both); and ”Arrow up” and ”Arrow down”, that browse previous typed commands and displays them in the ”Commandline” window.

After checking for key-presses, the program redraws the windows using *wrefresh* and sleeps with a small delay (to prevent screen flickering and unnecessary cpu loads). It then returns to the top of the main loop and the process starts over again.

### Send/Receive module

Figures 1.9 and 1.10 illustrates the behaviour of the main transmitter and receiver functions of the send/receive module. The basic operation goes as follows: The program sends a data-pattern from a 2d-array via the COM port to the GBT-registers on the FPGA side. It then sends a read request to read all values of the GBT-register back to confirm that the data pattern was transmittet properly. There are five different patterns that are sent in sequence to the registers, and each time a pattern is sent it is read back to the program on the PC side and displayed. This is to confirm that the communication link is working properly.

There are four variables that controls the transmitter: *txStatus*, *txReq*, *txData[2]* and *txAdr*. The *txStatus* variable controls the behaviour of both transmitter and receiver function. For the program to be able to send a new byte out the COM port, the *txStatus* variable must either be flagged as idle (0x00) or as repeat (0xFD). When flagged idle, the program is not waiting for any received address-byte and is ready to transmit a new request-byte to the COM port. When set to repeat, the program has already sent a request, but has not received the desired data and address byte, and must send the same read-request again. The *txReq* variable determines what request is to be sent to

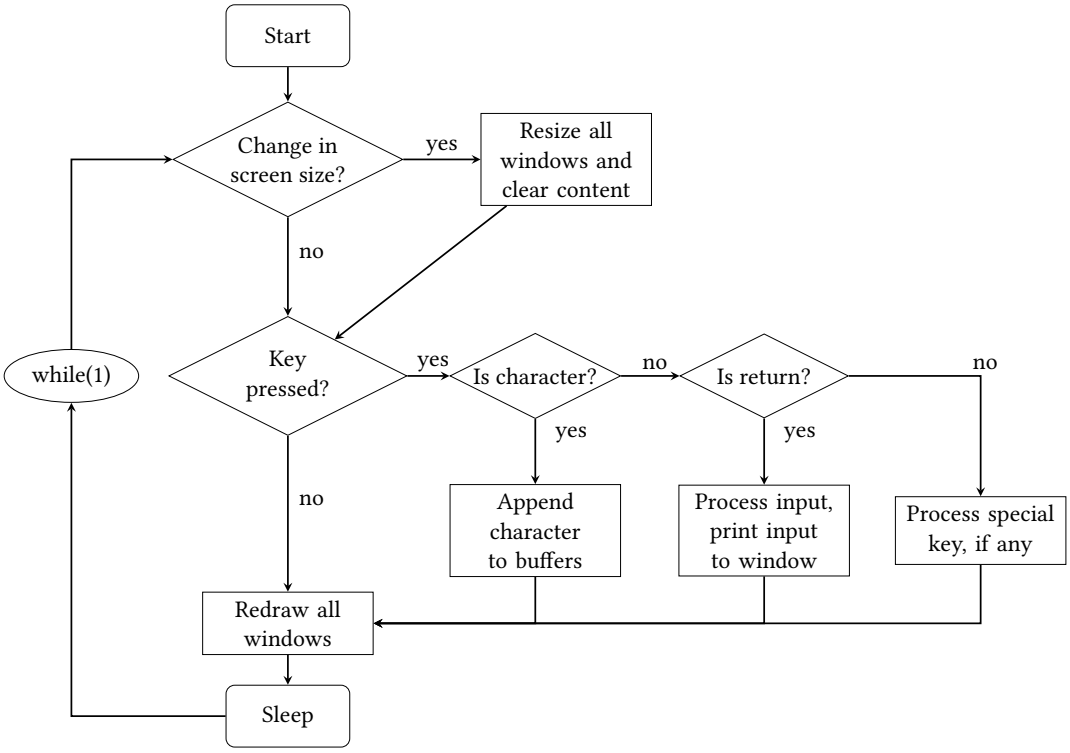


**Figure 1.8:** Flowchart over the main loop of the interface module.

the FPGA. When equal to a read-request (0xDD), both *txStatus* and *txData[0]* is set to 0xDD. If *txReq* is equal to a write-request (0xEE), however, data from the *GBT data table* is copied into *txData[0]* using *txAdr* as index. *txData[1]* is set always equal to *txAdr*, which is now the desired address to the GBT-register in which to read from or write to. Sending data to the COM port involves using the *RS232_Sendbuf*-function (see **??**). *txAdr* is then incremented by one or reset to 0 if it exceeds the width of the GBT data-register. If the latter is the case, it means that the read/write operation is finished, and *txReq* is set to the opposite operation. If a write operation is done, the pattern index is also incremented, so that the next write operation sends out a different pattern. The transmitter function is always called before the receiver function, with a specified delay in between the two function calls.

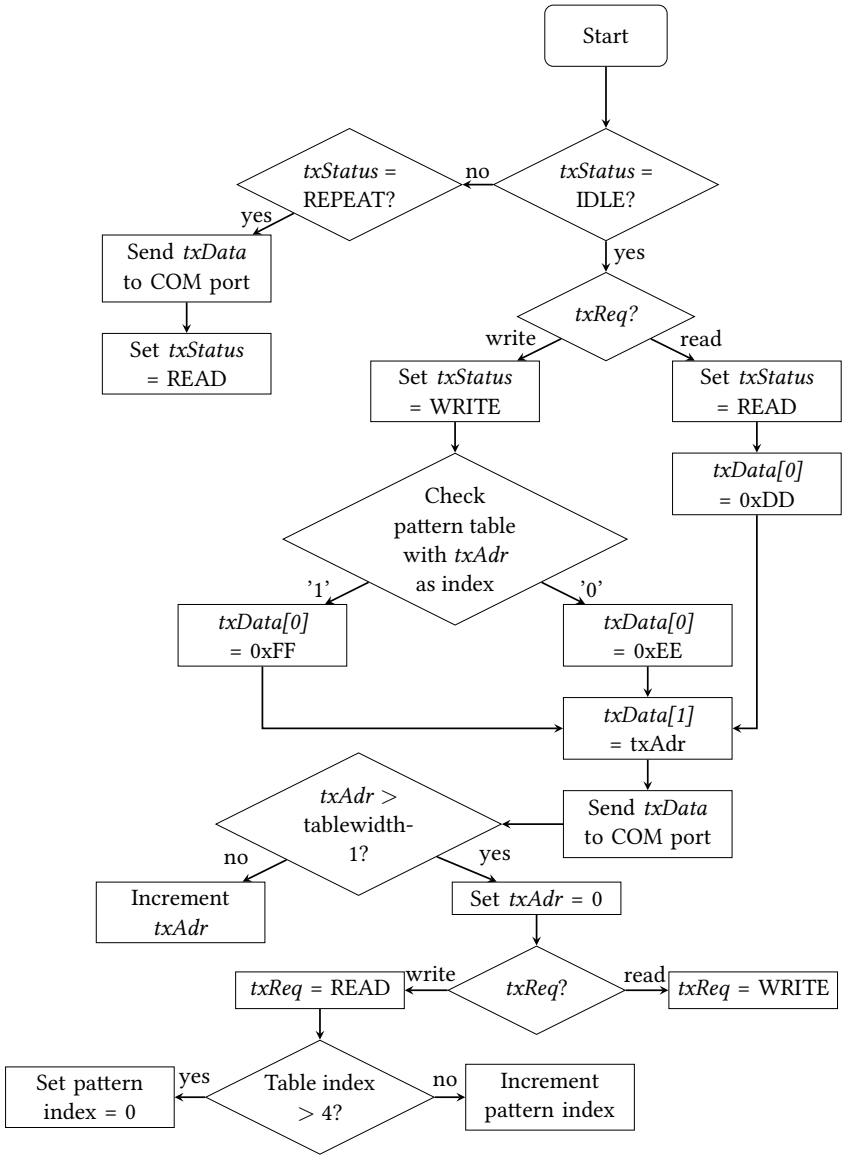


**Figure 1.9:** Flowchart over the transmitter function.

For the receiver function to read out what is stored in the COM port buffer, *txStatus* must be set to read (0xDD). The process of reading out a byte from the COM port involves using the *RS232_PollComport*-function (see **??**). The *n* variable is equal to the number of bytes read from the COM port. If no bytes are present, this means that the previous read-request has not been received or processed properly by the FPGA UART state machine. *txStatus* is therefore set equal to repeat (0xFD), signaling the transmitter function to send the same previous read-request to the COM port. If *n* is larger than 0, the data and address values are filtered out of the byte(s); the data bit being the Most Significant Bit (MSB) and the address the remaining bits.

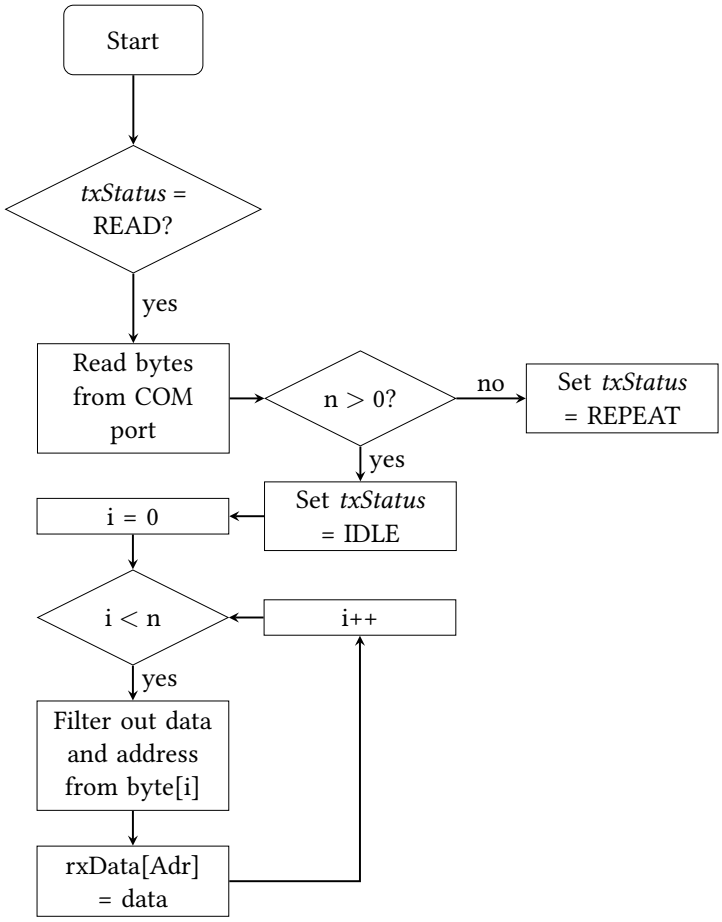


**Figure 1.10:** Flowchart over the receiver function.

## 1.7 Conclusion and Discussion

As for now, the serial interface software consists of two modules: the sending and receiving module, and the ncurses command interface. Both modules work more or less as they should: The send/receive module writes a pattern to the GBT-register in the FPGA and reads it back to the terminal, and the interface module allows you to perform write- and read-commands and monitor the GBT control signals. What remains is to fully integrate the send/receive module into the interface module. The module is partly integrated in terms of merging the two programs. What is left undone is to rewrite the main transmit and receive routines from the send/receive module, which are not completed. In addition, some adaptations needs to be made to the write- and read command-entries in order for them to invoke and direct the send/receive module. Also, a final test of the software as a whole remains.