Chapter 1

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 Gigabit Transceiver (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:

1.1 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/#.

1.1.1 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 super-user/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.

1.2 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 ms on the Windows platform and a resolution down to 1 µs 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/#.

1.2.1 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.

1.3 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.

1.3.1 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 Field-Programmable Gate Array (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.

1.3.2 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 1.4) 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.

1.4 ncurses

The ncurses library is a terminal control library commonly used with Unix-systems. It is ”freely redistributable in source form” [**?**], and enables you to construct interfaces using the terminal as the base. The main purpose of using ncurses for the GBT software interface was to enable multiple windows; one window to continuously update and display the GBT control signals, and another window for the user to write commands. To compile ncurses-based programs in Windows, one can use the *pdcurses* library. The *pdcurses* library has the same function names and overall functionality, so no alterations of code is needed. The following library and function information has been taken from http://linux.die.net/man/, which is a website collection of Linux-related documentation (ncurses is integrated into the Linux system).

1.4.1 Associated functions - Initialization

```
WINDOW *initscr(void);
```

Initialization function that sets up a new ncurses window. This needs to be called once, and before any other ncurses related function.

```
int noecho(void)
```

Disables the user input from being printed back on screen, i.e being echoed.

```
int cbreak(void)
```

Disables line buffering, making characters typed by the user available to the program one-by-one.

```
int keypad(WINDOW *win, bool bf)
```

Function that, when **TRUE**, allows the program to capture special keys using the function *getch()*, like *backspace* and the arrow keys. In the GBT serial interface, *win* is defined as *stdscr*, which is the main terminal screen.

```
int nodelay(WINDOW *win, bool bf)
```

Function that keeps the input capture function, *getch()*, from blocking the program when no input has been received. When *bf* is **TRUE**, getch() will return **ERR** instead of waiting for a key to be pressed. This function was necessary to allow one window to update freely while another window handled input. *win* is defined as *stdscr*.

```
void getmaxyx(WINDOW *win, int y, int x)
```

Function that store the current beginning coordinates and size of the specified window. In the case of the GBT software interface, *win* is specified as the main terminal screen, *stdscr*. These coordinates are needed when adding individual windows inside *stdscr*.

```
WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x)
```

Function used to initiate a new window within the main terminal screen, *stdscr*, where *nlines* a *ncols* is the number of lines and columns the window is to contain, and *begin_y* and *begin_x* is the upper left hand corner of the window. This function is used to initiate the three windows, ”Command History”, ”Commandline” and ”Signals” in the GBT serial interface.

```
int scrollok(WINDOW *win, bool bf)
```

Function that, if **TRUE**, enables scrolling of typed in characters, either by using the additional *scroll*-function, or when the cursor reached the last column of the last line in the specified window *win*. This function is only used with the ”Command History” window to scroll commands that are read in.

```
int leaveok(WINDOW *win, bool bf)
```

Function to disable the cursor at the specified window. It is used to disable the cursor in the ”Command History” and ”Signals” windows in the GBT serial interface.

1.4.2 Associated functions - Various

```
int wresize(WINDOW *win, int lines, int columns)
```

Enables for resizing of the initiated windows.

```
int wclear(WINDOW *win)
```

Clears the specified window of its content.

```
int mvwprintw(WINDOW *win, int y, int x, const char *fmt, ...)
```

Analogous to the printf routine, found in the C standard library. However, this allows to print output in a specified window with coordinates. *mvwprintw* is used when printing information out on the different windows.

```
int wrefresh(WINDOW *win)
```

Function to update the individual windows. This function must be called to get actual output to the terminal, preferably at the end of the main loop.

```
int wmove(WINDOW *win, int y, int x)
```

Function that allows for repositioning of the cursor at the specified window. This function is called to reposition the cursor in the ”Commandline” at the end of an input string.