Game application:	Points:	_____ / 7
Bluetooth/UART:	Points:	_____ / 5
Music Playback (MP3 and Volume):	Points:	_____ / 4
GLCD:	Points:	_____ / 5
Rand,ADC:	Points:	_____ / 4
Overall:	Points:	_____ / 25

Microcontroller VU

Application Protocol

Jan Nausner, MatrNr. 01614835
e01614835@student.tuwien.ac.at

November 28, 2018

Declaration of Academic Honesty

I hereby declare that this protocol (text and code) is my own original work written in my own words, that I have completed this work using only the sources cited in the text, and that neither this protocol nor parts of it have ever before been submitted to this or any other course.

_____ _____
(Date) (Signature of Student)

Admission to Publish

☐ I explicitly **allow** the publication of my solution (protocol and sourcecode) on the course webpage.

☐ I **do not allow** the publication of my solution (default if nothing is checked).

_____
(Signature of Student)

Contents

1 Overview

1.1 Connections, External Pullups/Pulldowns

Pin Assignment

What	
J12	Connected to VCC
J14	EXT
J15	PF0
LEDs	Off
GLCD backlight	On
PORT switches	Off
PORT jumpers	Pull down

The mp3-SD module needs to be connected as per specification and the bluetooth module needs to be connected to PORTJ.

1.2 Design Decisions

For abstraction and compartementalization purposes, the application is split up into many independant submodules, as proposed in the specification. Furthermore it was decided to implement a standalone timer module, which abstracts the timer hardware and allows to start timers by just specifiying the period in ms. Also the main application was not put in the main file, but rather in a seperate game module, connecting all the necessary driver modules, handling the game logic and task switching. This leads to a clean main.c file, without many includes or ISRs and just two function calls, one for setup and one for running the main application loop. Another major design decision was to precompute a number of random walls with a python script (randomWalls.py) for the game and store them in the flash to later load them randomly in order to trade storage for computation time.

1.3 Specialities

The application is very reactive to user input and runs smoothly. Also the user interface controls can be considered as intuitive or are appropriately explained with text. By partitioning the big background task into many subtasks it is prevented that one task monopolizes the CPU, leading to clear music playback, responsive controls and smooth gameplay. Also on the screen showing an ongoing connection attempt between MCU and wiimote an animation was included to signifiy the user that the application is still progressing (this could be a place to award bonus points). The former mentioned timer library can also be listed as a speciality, as it handles the

hardware in an intuitive way for the programmer and also checks if hardware is not allocated multiple times (another place for potentially awarding bonus points).

The gameplay itself can be considered a bit boring, as it might be a littlebit too easy for some people's taste. This could be improved by integrating harder platforms (less or smaller gaps) and supporting even higher scroll speeds. But this would be the task of a game UX specialist rather than that of a computer engineer to be.

2 Main Application

As described above, the code for the main application is not located in the main.c file but in a seperate game.c file. When the application is started, music starts playing, the program checks for a present bluetooth connection and if the remote is disconnted it initates a new connection. Provided the correct MAC-address was entered in the mac.h file, after several secondes the program switches to the start screen, implying a successful connection to the wiimote. On the start screen the user is presented with two options: either start playing the game or view the highscore table. If the highscore table is selected, it will be empty as no persistent storage was implemented.

When the "Play" option is selected, the user is presented a screen, where a player can be selected, using the up and down arrows on the wiimote. At this point, the user can choose to go back to the start screen or to start playing the game.

Upon entering the game, the screen is filled with random walls, the ball is placed on the bottom and the accelerometer in the wiimote is enabled in order to allow steering the ball. The level immediately starts scrolling and placing new random walls. The ball can be moved in x direction by holding the wiimote in an orientation similar to how one would hold a TV remote and by tilting it right or left. Gameplay is as described in the specification, the ball cannot move out of the visible field, falls down if not blocked by platforms or the bottom and the game is lost if the ball touches the top. The game either ends when the game-over condition is met or when the home button is pressed, which leads back to the start screen. In both cases the current score is placed in the highscore table if it is sufficiently high and the accelerometer is disabled again. In case the player was defeated, a game-over screen with the achieved score is shown. The user can chose to go back to the start screen or to directly view the highscore table, sorted in descending order.

The main application is driven by a gametick timer which fires every 30 ms. On every gametick, the user input from the wiimote is checked in order to determine if the app needs to move to another screen. In the game a few additional tasks need to be executed. First, the screen is updated by moving the ball according to the user input after performing collision detection and checking if the game has been lost. By using timer divisor counters, the screen scrolls after several gameticks and after shifting out a wall, a new random wall is drawn on the screen. The score counter and the difficulty (scrolling speed) are also updated by using timer divisor counters.

3 Music Playback

3.1 SPI

The SPI module was implemented mostly by following the examples given on pages 198-199 of the ATmega1280 manual. The module uses busy-waiting rather than interrupts and the SPI is configured to be in master mode.

3.2 Playback

Music playback is implemented in music.c by reading a 32 bit block of data from the SD card and sending it to the mp3 module. The function immediately returns, so that the game task can proceed in the meanwhile, but is called repeatedly until the buffer of th mp3 module is full. After the end of the selected song has been reached, it starts playing all over again. The music module also provides a function to set the volume on the mp3 board, which linearizes the input value and can be used as a callback for the ADC module. Note that the volume is only set in case the value has actually changed and if the SPI is not used by the music playback task.

4 ADC

The ADC is required to fullfill two tasks: reading the the onboard potentiometer to set the music volume and providing entropy for the PRNG module. This is achived by setting the ADC to auto trigger mode, triggering it every 200 Hz, and alternatingly reading the potentiometer and the floating pins, which produces an update of the respective values approximately every 10 ms. When the trigger timer interrupt arrives the ADC is enabled (required if differential mode is used in combination with auto triggering according to the manual). After finishing the conversion (ADC interrupt) the result is read, the ADC disabled, the ADC MUX gets set to read from the next source and the ADC result is passed to the appropriate callback.

5 GLCD–Display

5.1 GLCD

The GLCD module provides high-level functions for drawing and writing on the display. It relies on functions for setting, clearing and inverting single pixels. This functions need to first read the respective memory page before performing a pixel operation, as the state of the other pixels in the page must be preserved. The functions for drawing lines and circles implement the Bresenham Line Algorithm (https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm)

and the Midpoint Circle Algorithm (https://en.wikipedia.org/wiki/Midpoint_circle_algorithm). Empty and filled rectangles are composed of single lines, where several cases must be considered so that no pixel gets drawn twice in order for the invert operation to function correctly. In order to draw characters, it is required to set a pointer to the correct entry in the font array and then draw the respective pixels forming the character. The text writing functions call the character drawing function for every character in a string and discard any symbols not fitting on a given line.

5.2 HAL GLCD

The HAL GLCD module handles communication with the GLCD controller and implements four basic operations: reading and writing data, sending commands and reading the status bits. Before carrying out the first three cases, it must be ensured that the GLCD is reset and not busy, which is ensured by busy waiting until the corresponding status bits are set. After writing data or commands and before reading data, the respective control bits are set, the correct controller is selected and the enable flag is toggled as per timing diagram in the datasheet, which is realized by waiting with a few NOPs. The high level read, write and address-setting functions abstract away the fact that the GLCD in fact uses two seperate controllers by maintaining an internal state which controller needs to be used. This state is updated upon crossing memory boundaries. The y-shift is set with a dedicated command and the current value is tracked by an internal variable. The read function needs to perform two read operations where the first result is discarded, as the controllers require a dummy read.

6 PRNG

The pseudorandom number generator implements the linear feedback shift register algorithm suggested by the PRNG manual using volatile inline assembler. This assembler block needs to be atomic, as it makes heavy use of the carry bit and requires 16 bit arithmetic. It provides functions for feeding entropy and for extracting pseudorandom numbers from the LFSR.

7 Timer abstraction

The timer moduel provides abstraction for timers 1, 3, 4 and 5. The user can specify how long the timer should count in milliseconds and if the timer should only be fired once or if it should run periodically. The output compare registers and prescaler values are set accordingly. The module also offers the option to stop a periodic timer, which can for example be used for implementing animations. The module also provides feedback if the requested timer is available or already in use.

8 UART

The UART's baudrate is set to 1 M using double transmission speed. If a new byte is received, the RX ISR puts it in the ringbuffer and calls the processRingbuffer procedure (only once, ensured by lock flags), which empties the buffer by calling the receive callback in the background. If the buffer fill level reaches a certain limit, the CTS line is set to signify the bluetooth module that transmission should be halted. If a byte should be sent, there are three cases which need to be checked before sending. First, reset must be completed. If this is not the case, the byte to be sent gets buffered and sending is tried again upon reset completion. Second, it must be assured that the RTS hardware flow control flag is not set by the blutooth module. Otherwise, sending is postponed again and a pin change interrupt gets activated on the RTS pin. If the corresponding ISR is executed, sending of the buffered byte is tried again and the PCINT disabled. Finally, the UART data register must be empty. If it is not, the UART data register interrupt is enabled and upon triggering, sending is tried again. If nothing blocks the send procedure, the byte is sent and the UDRINT enabled. On execution of the corresponding ISR the send callback is called.

9 Problems

One problem that was encountered during the development of the application was the faulty function wiiUserSetAccel. In the specification it was stated that using this function it would be possible to either turn the wiimote accelerometer on or off. Unfortunately this behavior was not reflected in the implementation, which only turned the accelerometer on, regardless of the value provided to the enable parameter.

Another difficulty concerned the interpretation of the accelerometer data. It took some time and thinking to derive a simple and fast algorithm for wiimote tilt detection which did not require the use of complex numerics or even floating point arithmetic. To understand the principal behavior of an accelerometer, the following article was consulted: http://bildr.org/2011/04/sensing-orientation-with-the-adxl335-arduino/.

A lot of time had to be in invested in figuring out how to communicate with the GLCD module. The exact timing and pin level sequence had to be found out by trial-and-error, as the datasheets for the hardware were not completely accurate.

Finally, just before uploading the solution a bug was discovered, where the program started to freeze or reset the board after running a considerably long time. After many hours of debugging the error was found to be hidden in a for loop counting from high to low numbers. The problem was that one part of the loop condition was checking for the index to be greater or equal to zero and with the index being an unsigned integer, this sometimes resulted in an endless loop.

10 Work

Task	Assumption (IP)	Reality
reading manuals, datasheets	5 h	5 h
program design	10 h	6 h
programming	20 h	16 h
debugging	20 h	60 h
protocol	5 h	4 h
Total	60 h	91 h