Listings

A Listings

Include EVERY source file of your Application (including headers)!!! And EVERY file provided by us which you have modified!

A.1 Application

Listing 1: ../Application/main.c

```
#include <game.h>

int main(void)
{
    game_init();

    game_run();

    return 0;
}
```

Listing 2: ../Application/mac.h

```
#include <stdint.h>

#ifndef __MAC__
#define __MAC__

const uint8_t mac_address[6] = { 0x58, 0xbd, 0xa3, 0x4b, 0xf6, 0x80 };

#endif
```

Listing 3: ../Application/task.h

```
/**
 *
 * @file task.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-16
 *
 * Provide a task function return type.
 *
 */

#ifndef __TASK__
#define __TASK__

typedef enum {DONE, BUSY} task_state_t;

#endif
```

Listing 4: ../Application/game/data.h

```
/**
 *
 * @file data.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-08
 *
 * Game data stored in the program memory (flash).
 *
 */

#ifndef __DATA__
```

```
#define __DATA__

#include <avr/pgmspace.h>
#include <stdint.h>

/* Common strings */
const char data_menu_b[] PROGMEM = "Home: Menu";
const char data_player[] PROGMEM = "Player ";
const char data_game_name[] PROGMEM = "Falling down ball";
const char data_play_b[] PROGMEM = "1: Play";
const char data_highscore_b[] PROGMEM = "2: Highscore";
const char data_connecting[] PROGMEM = "Connecting";
const char data_towiimote[] PROGMEM = "to wiimote ...";
const char data_select_b[] PROGMEM = "A: Play B: Menu";
const char data_gameover[] PROGMEM = "Game over!";
const char data_score[] PROGMEM = "Score: ";

/* Connection screen animation */
struct connectFrame
{
    xy_point l0p0;
    xy_point l0p1;
    xy_point l1p0;
    xy_point l1p1;
};
const unsigned char data_connectFrames[2][4][2] PROGMEM =
{
    {{53, 40}, {73, 40}, {63, 35}, {63, 45}},
    {{53, 35}, {73, 45}, {73, 35}, {53, 45}}
};

/* A set of randomly generated walls for the game */
#define WALL_POINTS 5
typedef unsigned char wall_points_t[WALL_POINTS];
#define WALLS_AVAILABLE 128
const wall_points_t data_walls[WALLS_AVAILABLE] PROGMEM =
{
        {2,23,47,97,127},
        {0,30,80,104,125},
        {1,16,48,87,127},
        {0,40,79,111,126},
        {4,20,45,109,127},
        {0,18,82,107,123},
        {9,22,51,112,127},
        {0,15,76,105,118},
        {2,18,30,84,127},
        {0,43,97,109,125},
        {15,42,53,115,127},
        {0,12,74,85,112},
        {21,37,45,89,127},
        {0,38,82,90,106},
        {11,26,60,79,127},
        {0,48,67,101,116},
        {20,41,56,99,127},
        {0,28,71,86,107},
        {17,45,81,110,127},
        {0,17,46,82,110},
        {15,38,74,93,127},
        {0,34,53,89,112},
        {11,37,58,92,127},
        {0,35,69,90,116},
        {12,27,35,76,127},
        {0,51,92,100,115},
        {9,43,53,90,127},
        {0,37,74,84,118},
        {0,15,42,92,127},
        {0,35,85,112,127},
        {19,37,72,105,127},
        {0,22,55,90,108},
        {20,39,49,73,127},
```

```
{0,54,78,88,107},
{12,44,71,112,127},
{0,15,56,83,115},
{12,38,50,60,127},
{0,67,77,89,115},
{16,51,75,110,127},
{0,17,52,76,111},
{5,18,53,68,127},
{0,59,74,109,122},
{0,33,58,71,127},
{0,56,69,94,127},
{20,36,62,100,127},
{0,27,65,91,107},
{1,11,42,106,127},
{0,21,85,116,126},
{18,31,50,68,127},
{0,59,77,96,109},
{18,32,53,79,127},
{0,48,74,95,109},
{17,30,46,99,127},
{0,28,81,97,110},
{5,36,47,115,127},
{0,12,80,91,122},
{14,35,71,94,127},
{0,33,56,92,113},
{14,31,49,66,127},
{0,61,78,96,113},
{1,14,42,65,127},
{0,62,85,113,126},
{11,31,40,103,127},
{0,24,87,96,116},
{6,41,64,74,127},
{0,53,63,86,121},
{10,28,50,116,127},
{0,11,77,99,117},
{15,31,65,106,127},
{0,21,62,96,112},
{13,34,69,82,127},
{0,45,58,93,114},
{12,41,59,90,127},
{0,37,68,86,115},
{21,45,82,102,127},
{0,25,45,82,106},
{3,22,59,110,127},
{0,17,68,105,124},
{9,19,35,104,127},
{0,23,92,108,118},
{7,24,40,102,127},
{0,25,87,103,120},
{17,52,75,96,127},
{0,31,52,75,110},
{3,25,38,63,127},
{0,64,89,102,124},
{7,30,49,88,127},
{0,39,78,97,120},
{21,35,67,112,127},
{0,15,60,92,106},
{10,47,71,105,127},
{0,22,56,80,117},
{4,41,73,94,127},
{0,33,54,86,123},
{8,29,60,75,127},
{0,52,67,98,119},
{13,32,52,103,127},
{0,24,75,95,114},
{9,44,75,101,127},
{0,26,52,83,118},
{3,31,48,85,127},
{0,42,79,96,124},
{4,24,43,101,127},
```

```
        {0,26,84,103,123},
        {12,49,78,90,127},
        {0,37,49,78,115},
        {11,24,35,67,127},
        {0,60,92,103,116},
        {0,35,49,95,127},
        {0,32,78,92,127},
        {20,33,54,75,127},
        {0,52,73,94,107},
        {10,42,50,95,127},
        {0,32,77,85,117},
        {8,25,61,72,127},
        {0,55,66,102,119},
        {11,23,45,115,127},
        {0,12,82,104,116},
        {9,38,53,85,127},
        {0,42,74,89,118},
        {9,45,77,103,127},
        {0,24,50,82,118},
        {20,43,52,92,127},
        {0,35,75,84,107},
        {13,41,49,69,127},
        {0,58,78,86,114},
        {7,17,32,68,127},
        {0,59,95,110,120}
};

#endif
```

Listing 5: ../Application/game/game.h

```
/**
 *
 * @file game.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-13
 *
 * Header file for the game module.
 *
 */

#ifndef __GAME__
#define __GAME__

#include <stdint.h>

/**
 * @brief       Initialize the game user interface.
 */
void game_init(void);

void game_run(void);

#endif
```

Listing 6: ../Application/game/game.c

```
/**
 *
 * @file game.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-13
 *
 * Implementation of the falling down ball game.
 *
 */

#include <avr/io.h>
#include <avr/sleep.h>
```

```
#include <avr/pgmspace.h>
#include <avr/interrupt.h>

#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include <glcd.h>
#include <Standard5x7.h>
#include <wii_user.h>
#include <adc.h>
#include <rand.h>
#include <timer.h>
#include <music.h>
#include <data.h>
#include <mac.h>
#include <task.h>
#include <game.h>

#define X_WIDTH         128
#define Y_HEIGHT        64
#define TOP             0
#define BOTTOM          Y_HEIGHT-1

/* Wii button encoding */
#define BUTTON_2_WII    0x01
#define BUTTON_1_WII    0x02
#define BUTTON_B_WII    0x04
#define BUTTON_A_WII    0x08
#define BUTTON_H_WII    0x80
#define BUTTON_L_WII    (0x01<<8)
#define BUTTON_R_WII    (0x02<<8)
#define BUTTON_D_WII    (0x04<<8)
#define BUTTON_U_WII    (0x08<<8)

/* Local button encoding */
#define BUTTON_1        0x01
#define BUTTON_2        0x02
#define BUTTON_A        0x04
#define BUTTON_B        0x08
#define BUTTON_H        0x10
#define BUTTON_D        0x20
#define BUTTON_U        0x40

/* Accelerometer corner values */
#define X_MIN           0x66
#define X_MID           0x80
#define X_MAX           0x99
#define Y_MIN           0x52
#define Y_MAX           0xa7
#define Z_MIN           0x66
#define Z_MID           0x80
#define Z_MAX           0x99
#define ACC_DELTA       10

/* Game parameters */
#define TICK_PERIOD_MS          30
#define TICKS_PER_SCROLL        9
#define TICKS_PER_SCORE         5
#define TICKS_PER_DIFF          100
#define MAX_DIFFICULTY          2
#define WALL_GAP                15
#define BALL_RADIUS             2
#define GRAVITY                 1
#define PLAYERNUM               5
#define SELECTOR_RADIUS         2
#define SELECTOR_Y_START        6
#define LINE_SPACING            10
#define LINE_MARGIN             10
```

```
typedef enum {START, CONNECT, SELECTPLAYER, PLAY, GAMEOVER, HIGHSCORE} game_state_t;
typedef enum {INIT, WAIT} static_state_t;
typedef enum {SETUP, UPDATE, SCROLL, LEVEL, NEXT} tick_state_t;

/* Interrupt flags */
static struct
{
    volatile task_state_t game;
    volatile task_state_t music;
} workLeft;

/* State variables */
static struct
{
    game_state_t next;
    static_state_t start:1;
    static_state_t connect:1;
    static_state_t selectPlayer:1;
    static_state_t gameOver:1;
    static_state_t highScore:1;
    tick_state_t play;
} gameStates;

/* Wiimote status flags */
static struct
{
    uint8_t triedConnect:1;
    connection_status_t status:1;
    uint8_t triedSetAcc:1;
    uint8_t accStatus:1;
} wiimote;

/* Wiimote sensor values */
static struct
{
    uint8_t buttons;
    uint8_t accX;
} input;

/* Counter variables */
static struct
{
    uint8_t scrollDiv;
    uint8_t scroll;
    uint8_t score;
    uint8_t level;
    uint16_t diffDiv;
    uint16_t diff;
} tickCnt;

/* Current screen image */
#define WALLS_ON_SCREEN 4
struct wall
{
    uint8_t yPos;
    wall_points_t points;
};
static struct
{
    uint8_t topWall;
    xy_point ball;
    struct wall walls[WALLS_ON_SCREEN];
} screenImage;

/* Screen dynamic values */
struct ball_dyn
{
    int8_t xAcc;
    int8_t yAcc;
};
```

```
static struct
{
    uint8_t yShift;
    struct ball_dyn ballDynamics;
} screenDynamics;

/* Player data */
struct highScoreEntry
{
    int8_t player;
    uint16_t score;
};
static struct
{
    uint8_t currPlayer;
    uint16_t currScore;
    struct highScoreEntry highScore[PLAYERNUM];
} playerData;

/* Connect screen animation figure */
#define CONNECT_FRAMES 2
#define CONNECT_FRAME_MS 350
uint8_t currFrame;

/* Callback functions */
static void gameTimerCB(void);
static void musicCB(void);
static void buttonCB(uint8_t wii, uint16_t buttonStates);
static void accelCB(uint8_t wii, uint16_t x, uint16_t y, uint16_t z);
static void connectCB(uint8_t wii, connection_status_t status);
static void setAccelCB(uint8_t wii, error_t status);
static void connectAnimCB(void);

/* Local functions */
static task_state_t start(game_state_t *game_state);
static task_state_t connect(game_state_t *game_state);
static task_state_t selectPlayer(game_state_t *game_state);
static task_state_t play(game_state_t *game_state);
static task_state_t gameOver(game_state_t *game_state);
static task_state_t highScore(game_state_t *game_state);
static void displayStartText(uint8_t y_top);
static void displayConnectText(uint8_t y_top);
static void displaySelectPlayerText(uint8_t y_top);
static void displayGameOverText(uint8_t y_top);
static void displayHighScoreText(uint8_t y_top);
static void initLevel(void);
static void moveSelector(uint8_t curr, uint8_t next);
static void playUpdate(void);
static void playScroll(void);
static void displayNewWall(uint8_t yOff);
static void drawWall(uint8_t wall);
static void clearWall(uint8_t wall);
static void calcBallAcc(void);
static uint8_t updateBallPos(void);
static void drawBall(void);
static void clearBall(void);
static void enterHighScore(void);

/**
 * @brief       Initialize the game user interface.
 */
void game_init(void)
{
    glcdInit();
    music_init(&musicCB);
    adc_setCallbacks(&rand_feed, &music_setVolume);
    adc_init();
    while (wiiUserInit(&buttonCB, &accelCB) != SUCCESS);

    /* Initialize the structs */
```

```
    wiimote.triedConnect = 0;
    wiimote.status = DISCONNECTED;
    wiimote.triedSetAcc = 0;
    wiimote.accStatus = 0;
    screenDynamics.yShift = glcdGetYShift();

    for (uint8_t p = 0; p < PLAYERNUM; p++)
    {
        playerData.highScore[p].player = -1;
        playerData.highScore[p].score = 0;
    }

    workLeft.game = 0;
    workLeft.music = 0;

    sei();
}

/*
 * @brief       Run the game. This procedure switches between the music and game tasks.
 */
void game_run(void)
{
    game_state_t game_state = START;

    set_sleep_mode(SLEEP_MODE_IDLE);
    sleep_enable();

    timer_startTimer1(TICK_PERIOD_MS, TIMER_REPEAT, &gameTimerCB);

    for (;;)
    {
        do
        {
            if (workLeft.game != DONE)
            {
                if (START == game_state)
                    workLeft.game = start(&game_state);
                else if (CONNECT == game_state)
                    workLeft.game = connect(&game_state);
                else if (SELECTPLAYER == game_state)
                    workLeft.game = selectPlayer(&game_state);
                else if (PLAY == game_state)
                    workLeft.game = play(&game_state);
                else if (GAMEOVER == game_state)
                    workLeft.game = gameOver(&game_state);
                else if (HIGHSCORE == game_state)
                    workLeft.game = highScore(&game_state);

            }
            if (workLeft.music != DONE)
                workLeft.music = music_play();

        } while (workLeft.game != DONE || workLeft.music != DONE);

        sleep_cpu();
    }
}

/*
 * @brief               Display the start screen of the game.
 * @param game_state    Contains the next state after the procedure call.
 * @return              The task state is returned, either DONE or BUSY.
 */
static task_state_t start(game_state_t *game_state)
{
    if (INIT == gameStates.start)
    {
        glcdFillScreen(GLCD_CLEAR);
        displayStartText(LINE_SPACING);
```

```
        gameStates.start = WAIT;
        return BUSY;
    }
    if (WAIT == gameStates.start)
    {
        if (DISCONNECTED == wiimote.status)
            *game_state = CONNECT;
        else if (BUTTON_1 & input.buttons)
            *game_state = SELECTPLAYER;
        else if (BUTTON_2 & input.buttons)
            *game_state = HIGHSCORE;

        if (START != *game_state)
        {
            gameStates.start = INIT;
            input.buttons = 0;
        }
    }

    return DONE;
}

/*
 * @brief               Display the connect screen of the game and waits
 *                      for a successful wiimote connection.
 * @param game_state    Contains the next state after the procedure call.
 * @return              Returns DONE.
 */
static task_state_t connect(game_state_t *game_state)
{
    if (INIT == gameStates.connect)
    {
        glcdFillScreen(GLCD_CLEAR);
        displayConnectText(LINE_SPACING);
        gameStates.connect = WAIT;
        wiimote.accStatus = 0;
        wiimote.triedSetAcc = 0;
        timer_startTimer3(CONNECT_FRAME_MS, TIMER_REPEAT, &connectAnimCB);
        return BUSY;
    }
    if (WAIT == gameStates.connect)
    {
        if (wiimote.triedConnect == 0)
        {
            if(wiiUserConnect(0, mac_address, &connectCB) == SUCCESS)
                wiimote.triedConnect = 1;
        }

        if (CONNECTED == wiimote.status)
        {
            timer_stopTimer3();
            *game_state = START;
            gameStates.connect = INIT;
            input.buttons = 0;
        }
    }

    return DONE;
}

/*
 * @brief               Lets the user select a player.
 * @param game_state    Contains the next state after the procedure call.
 * @return              The task state is returned, either DONE or BUSY.
 */
static task_state_t selectPlayer(game_state_t *game_state)
{
    uint8_t lastPlayer;

    if (INIT == gameStates.selectPlayer)
```

```
    {
        glcdFillScreen(GLCD_CLEAR);
        displaySelectPlayerText(LINE_SPACING);
        moveSelector(0, 0);
        playerData.currPlayer = 0;
        gameStates.selectPlayer = WAIT;
        return BUSY;
    }
    if (WAIT == gameStates.selectPlayer)
    {
        if (DISCONNECTED == wiimote.status)
            *game_state = CONNECT;
        else if (BUTTON_A & input.buttons)
            *game_state = PLAY;
        else if (BUTTON_B & input.buttons)
            *game_state = START;
        /* Move cursor up */
        else if (BUTTON_U & input.buttons)
        {
            lastPlayer = playerData.currPlayer;
            if (playerData.currPlayer == 0)
                playerData.currPlayer = PLAYERNUM-1;
            else
                playerData.currPlayer--;
            moveSelector(lastPlayer, playerData.currPlayer);
            input.buttons &= ~BUTTON_U;
        }
        /* Move cursor down */
        else if (BUTTON_D & input.buttons)
        {
            lastPlayer = playerData.currPlayer;
            if (playerData.currPlayer == PLAYERNUM-1)
                playerData.currPlayer = 0;
            else
                playerData.currPlayer++;
            moveSelector(lastPlayer, playerData.currPlayer);
            input.buttons &= ~BUTTON_D;
        }

        if (SELECTPLAYER != *game_state)
        {
            gameStates.selectPlayer = INIT;
            input.buttons = 0;
        }
    }

    return DONE;
}

/*
 * @brief               The main game function. Takes user input and updates
 *                      the screen accordingly.
 * @param game_state    Contains the next state after the procedure call.
 * @return              The task state is returned, either DONE or BUSY.
 */
static task_state_t play(game_state_t *game_state)
{
    if (SETUP == gameStates.play)
    {
        gameStates.next = PLAY;
        if (wiimote.accStatus == 1)
        {
            glcdFillScreen(GLCD_CLEAR);
            initLevel();
            input.buttons = 0;
            gameStates.play = UPDATE;
        }
        /* Enable accelerometer */
        else if (wiimote.triedSetAcc == 0)
        {
```

```
            if (wiiUserSetAccel(0, 1, &setAccelCB) == SUCCESS)
                wiimote.triedSetAcc = 1;
        }
        return BUSY;
    }
    if (UPDATE == gameStates.play)
    {
        playUpdate();
        return BUSY;
    }
    if (SCROLL == gameStates.play)
    {
        playScroll();
        return BUSY;
    }
    if (LEVEL == gameStates.play)
    {
        gameStates.play = NEXT;
        clearWall(screenImage.topWall);
        displayNewWall(BOTTOM);
        return BUSY;
    }
    if (NEXT == gameStates.play)
    {
        if (PLAY == gameStates.next)
        {
            if (DISCONNECTED == wiimote.status)
                gameStates.next = CONNECT;
            else if (BUTTON_H & input.buttons)
                gameStates.next = START;
        }
        if (PLAY != gameStates.next)
        {
            if (wiimote.accStatus == 0 || CONNECT == gameStates.next)
            {
                /* New highscore entry */
                enterHighScore();

                *game_state = gameStates.next;
                gameStates.play = SETUP;
                input.buttons = 0;
            }
            /* Disable accelerometer */
            else if (wiimote.triedSetAcc == 0)
            {
                if (wiiUserSetAccel(0, 0, &setAccelCB) == SUCCESS)
                    wiimote.triedSetAcc = 1;
            }
        }
        else
            gameStates.play = UPDATE;
    }

    return DONE;
}

/*
 * @brief               Displays the game over message.
 * @param game_state    Contains the next state after the procedure call.
 * @return              The task state is returned, either DONE or BUSY.
 */
static task_state_t gameOver(game_state_t *game_state)
{
    if (INIT == gameStates.gameOver)
    {
        glcdFillScreen(GLCD_CLEAR);
        displayGameOverText(LINE_SPACING);
        gameStates.gameOver = WAIT;
        return BUSY;
    }
```

```
    if (WAIT == gameStates.gameOver)
    {
        if (DISCONNECTED == wiimote.status)
            *game_state = CONNECT;
        else if (BUTTON_H & input.buttons)
            *game_state = START;
        else if (BUTTON_2 & input.buttons)
            *game_state = HIGHSCORE;

        if (GAMEOVER != *game_state)
        {
            gameStates.gameOver = INIT;
            input.buttons = 0;
        }
    }

    return DONE;
}

/*
 * @brief               Diplays the current highscore table (max. 5 entries).
 * @param game_state    Contains the next state after the procedure call.
 * @return              The task state is returned, either DONE or BUSY.
 */
static task_state_t highScore(game_state_t *game_state)
{
    if (INIT == gameStates.highScore)
    {
        glcdFillScreen(GLCD_CLEAR);
        displayHighScoreText(LINE_SPACING);
        gameStates.highScore = WAIT;
        return BUSY;
    }
    if (WAIT == gameStates.highScore)
    {
        if (DISCONNECTED == wiimote.status)
            *game_state = CONNECT;
        else if (BUTTON_H & input.buttons)
            *game_state = START;

        if (HIGHSCORE != *game_state)
        {
            gameStates.highScore = INIT;
            input.buttons = 0;
        }
    }

    return DONE;
}

/*
 * @brief   Callback function for the main game timer. Set a flag on interrupt.
 */
static void gameTimerCB(void)
{
    workLeft.game = 1;
}

/*
 * @brief   Callback function for the mp3 module. Set a flag on interrupt.
 */
static void musicCB(void)
{
    workLeft.music = 1;
}

/*
 * @brief   Callback function for the wiimote buttons.
 */
static void buttonCB(uint8_t wii, uint16_t buttonStates)
```

```
{
    (void) wii;

    if (buttonStates & BUTTON_A_WII)
        input.buttons |= BUTTON_A;
    if (buttonStates & BUTTON_B_WII)
        input.buttons |= BUTTON_B;
    if (buttonStates & BUTTON_1_WII)
        input.buttons |= BUTTON_1;
    if (buttonStates & BUTTON_2_WII)
        input.buttons |= BUTTON_2;
    if (buttonStates & BUTTON_H_WII)
        input.buttons |= BUTTON_H;
    if (buttonStates & BUTTON_D_WII)
        input.buttons |= BUTTON_D;
    if (buttonStates & BUTTON_U_WII)
        input.buttons |= BUTTON_U;
}

/*
 * @brief   Callback function for the wiimote accelerometer.
 */
static void accelCB(uint8_t wii, uint16_t x, uint16_t y, uint16_t z)
{
    (void) wii;
    (void) y;
    (void) z;

    input.accX = x>>2;
}

/*
 * @brief   Callback function for enabling/disabling the wiimote accelerometer.
 */
static void connectCB(uint8_t wii, connection_status_t status)
{
    (void) wii;

    wiimote.status = status;
    wiimote.triedConnect = 0;
}

/*
 * @brief   Callback function for enabling/disabling the wiimote accelerometer.
 */
static void setAccelCB(uint8_t wii, error_t status)
{
    (void) wii;
    (void) status;

    wiimote.accStatus = !wiimote.accStatus;
    wiimote.triedSetAcc = 0;
}

/*
 * @brief   Display a little animation on the connect screen.
 */
static void connectAnimCB(void)
{
    struct connectFrame frame;
    memcpy_P(&frame, &data_connectFrames[currFrame], sizeof(struct connectFrame));
    glcdDrawLine(frame.l0p0,
                 frame.l0p1,
                 &glcdClearPixel);
    glcdDrawLine(frame.l1p0,
                 frame.l1p1,
                 &glcdClearPixel);
    currFrame = !currFrame;
    memcpy_P(&frame, &data_connectFrames[currFrame], sizeof(struct connectFrame));
    glcdDrawLine(frame.l0p0,
```

```
                frame.l0p1,
                &glcdSetPixel);
    glcdDrawLine(frame.l1p0,
                frame.l1p1,
                &glcdSetPixel);
}

/*
 * @brief   Display the text for the start screen.
 * @param y The y coordinate of the top line.
 */
static void displayStartText(uint8_t yTop)
{
    xy_point startPoint;
    startPoint.y = (screenDynamics.yShift+yTop) & (Y_HEIGHT-1);
    startPoint.x = LINE_MARGIN;

    glcdDrawTextPgm(data_game_name, startPoint, &Standard5x7, &glcdSetPixel);
    startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    glcdDrawTextPgm(data_play_b, startPoint, &Standard5x7, &glcdSetPixel);
    startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    glcdDrawTextPgm(data_highscore_b, startPoint, &Standard5x7, &glcdSetPixel);
}

/*
 * @brief   Display the text for the connect screen.
 * @param y The y coordinate of the top line.
 */
static void displayConnectText(uint8_t yTop)
{
    xy_point startPoint;
    startPoint.y = (screenDynamics.yShift+yTop) & (Y_HEIGHT-1);
    startPoint.x = LINE_MARGIN;

    glcdDrawTextPgm(data_connecting, startPoint, &Standard5x7, &glcdSetPixel);
    startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    glcdDrawTextPgm(data_towiimote, startPoint, &Standard5x7, &glcdSetPixel);
}

/*
 * @brief   Display the text for the select player screen.
 * @param y The y coordinate of the top line.
 */
static void displaySelectPlayerText(uint8_t yTop)
{
    uint8_t slen = strlen_P(data_player);
    char plStr[slen+2];

    xy_point startPoint;
    startPoint.y = (screenDynamics.yShift+yTop) & (Y_HEIGHT-1);
    startPoint.x = LINE_MARGIN;

    for (uint8_t p = 0; p < PLAYERNUM; p++)
    {
        memset(plStr, 0, slen+2);
        strcpy_P(plStr, data_player);
        sprintf(plStr+slen, "%u", p+1);
        glcdDrawText(plStr, startPoint, &Standard5x7, &glcdSetPixel);
        startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    }

    glcdDrawTextPgm(data_select_b, startPoint, &Standard5x7, &glcdSetPixel);
}

/*
 * @brief   Display the text for the game over screen.
 * @param y The y coordinate of the top line.
 */
static void displayGameOverText(uint8_t yTop)
{
```

```
    uint8_t slen = strlen_P(data_score);
    char goStr[slen+6];

    xy_point startPoint;
    startPoint.y = (screenDynamics.yShift+yTop) & (Y_HEIGHT-1);
    startPoint.x = LINE_MARGIN;

    glcdDrawTextPgm(data_gameover, startPoint, &Standard5x7, &glcdSetPixel);
    startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    memset(goStr, 0, slen+6);
    strcpy_P(goStr, data_score);
    sprintf(goStr+slen, "%u", playerData.currScore);
    glcdDrawText(goStr, startPoint, &Standard5x7, &glcdSetPixel);
    startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    glcdDrawTextPgm(data_menu_b, startPoint, &Standard5x7, &glcdSetPixel);
    startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    glcdDrawTextPgm(data_highscore_b, startPoint, &Standard5x7, &glcdSetPixel);
}

/*
 * @brief   Display the current highscore table.
 * @param y The y coordinate of the top line.
 */
static void displayHighScoreText(uint8_t yTop)
{
    uint8_t slen = strlen_P(data_player);
    char hsStr[slen+9];

    xy_point startPoint;
    startPoint.y = (screenDynamics.yShift+yTop) & (Y_HEIGHT-1);
    startPoint.x = LINE_MARGIN;

    for (uint8_t p = 0; p < PLAYERNUM; p++)
    {
        if (playerData.highScore[p].player < 0)
            break;

        memset(hsStr, 0, slen+9);
        strcpy_P(hsStr, data_player);
        sprintf(hsStr+slen, "%d: %u", playerData.highScore[p].player+1, playerData.highScore[p
    ↪ ].score);
        glcdDrawText(hsStr, startPoint, &Standard5x7, &glcdSetPixel);
        startPoint.y = (startPoint.y+LINE_SPACING) & (Y_HEIGHT-1);
    }

    startPoint.y = ((screenDynamics.yShift+yTop) & (Y_HEIGHT-1)) + LINE_SPACING*PLAYERNUM;
    glcdDrawTextPgm(data_menu_b, startPoint, &Standard5x7, &glcdSetPixel);
}

/*
 * @brief   Procedure to initialise the screen with the ball and a new random level.
 */
static void initLevel(void)
{
    uint8_t newWall;
    uint8_t yPos = 0;
    screenImage.topWall = 0;

    for (uint8_t w = 0; w < WALLS_ON_SCREEN; w++)
    {
        newWall = rand16() & (WALLS_AVAILABLE-1);
        /* Load new wall from PROGMEM */
        memcpy_P(&screenImage.walls[w].points,
                 &data_walls[newWall], WALL_POINTS);
        screenImage.walls[w].yPos = yPos;
        drawWall(w);
        yPos += WALL_GAP+1;
    }

    screenDynamics.ballDynamics.xAcc = 0;
```

```
    screenDynamics.ballDynamics.yAcc = GRAVITY;
    screenImage.ball.x = (X_WIDTH/2)-1;
    screenImage.ball.y = BOTTOM-BALL_RADIUS;
    drawBall();

    tickCnt.scrollDiv = TICKS_PER_SCROLL-1;
    tickCnt.scroll = TICKS_PER_SCROLL-1;
    tickCnt.diffDiv = TICKS_PER_DIFF-1;
    tickCnt.diff = 0;
    tickCnt.score = TICKS_PER_SCORE-1;
    tickCnt.level = WALL_GAP;

    playerData.currScore = 0;
}

/*
 * @brief       This function moves the selector on the select player screen.
 * @param curr  Currently selected player.
 * @param next  Next selected player.
 */
static void moveSelector(uint8_t curr, uint8_t next)
{
    xy_point selector;

    selector.x = 4;
    selector.y = SELECTOR_Y_START+screenDynamics.yShift+LINE_SPACING*curr;
    glcdDrawCircle(selector, SELECTOR_RADIUS, &glcdClearPixel);
    selector.y = SELECTOR_Y_START+screenDynamics.yShift+LINE_SPACING*next;
    glcdDrawCircle(selector, SELECTOR_RADIUS, &glcdSetPixel);
}

/*
 * @brief   This procedure performs the necessary updates of a game tick.
 *          The scroll, score and difficulty counters are updated and the
 *          ball is moved to its new position.
 */
static void playUpdate(void)
{
    if (tickCnt.scroll >= tickCnt.scrollDiv)
    {
        gameStates.play = SCROLL;
        tickCnt.scroll = 0;
    }
    else
    {
        gameStates.play = NEXT;
        tickCnt.scroll++;
    }

    /* Update player score */
    if (tickCnt.score == TICKS_PER_SCORE-1)
    {
        playerData.currScore++;
        tickCnt.score = 0;
    }
    else
        tickCnt.score++;

    /* Increase game difficulty */
    if (tickCnt.diff == tickCnt.diffDiv && tickCnt.scrollDiv >= MAX_DIFFICULTY)
    {
        tickCnt.scrollDiv--;
        tickCnt.diffDiv = TICKS_PER_DIFF;
        tickCnt.diff = 0;
    }
    else
        tickCnt.diff++;

    calcBallAcc();
    clearBall();
```

```
    if (updateBallPos() == 1)
    {
        gameStates.play = NEXT;
        gameStates.next = GAMEOVER;
    }
    drawBall();
}

/*
 * @brief   This procedure performs the scrolling of the screen. The
 *          internally stored ball an wall positions are updated accordingly.
 */
static void playScroll(void)
{
    if (tickCnt.level == WALL_GAP)
    {
        gameStates.play = LEVEL;
        tickCnt.level = 0;
    }
    else
    {
        gameStates.play = NEXT;
        tickCnt.level++;
    }

    if (screenDynamics.yShift == Y_HEIGHT-1)
        screenDynamics.yShift = 0;
    else
        screenDynamics.yShift++;

    /* Adapt y position of walls and ball on shift */
    for (uint8_t w = 0; w < WALLS_ON_SCREEN; w++)
    {
        if (screenImage.walls[w].yPos == 0)
            screenImage.walls[w].yPos = BOTTOM;
        else
            screenImage.walls[w].yPos--;
    }

    if (screenImage.ball.y == 0)
        screenImage.ball.y = BOTTOM;
    else
        screenImage.ball.y--;

    glcdSetYShift(screenDynamics.yShift);
}

static void displayNewWall(uint8_t yOff)
{
    uint8_t newWall = rand16() & (WALLS_AVAILABLE-1);

    /* Load new wall from PROGMEM */
    memcpy_P(&screenImage.walls[screenImage.topWall].points,
             &data_walls[newWall], WALL_POINTS);
    screenImage.walls[screenImage.topWall].yPos = yOff;

    drawWall(screenImage.topWall);

    if (screenImage.topWall == WALLS_ON_SCREEN-1)
        screenImage.topWall = 0;
    else
        screenImage.topWall++;
}

/*
 * @brief       Draws a level wall on the screen.
 * @param wall  The index of the wall in the screenImage.
 */
static void drawWall(uint8_t wall)
{
```

```
    xy_point point0, point1;

    point0.y = screenImage.walls[wall].yPos+screenDynamics.yShift;
    point1.y = screenImage.walls[wall].yPos+screenDynamics.yShift;

    for (uint8_t i = 0; i < WALL_POINTS; i += 2)
    {
        if (i == WALL_POINTS-1)
        {
            if (screenImage.walls[wall].points[i] != X_WIDTH-1)
            {
                point0.x = screenImage.walls[wall].points[i];
                point1.x = X_WIDTH-1;
                glcdDrawLine(point0, point1, &glcdSetPixel);
            }
        }
        else
        {
            point0.x = screenImage.walls[wall].points[i];
            point1.x = screenImage.walls[wall].points[i+1];
            glcdDrawLine(point0, point1, &glcdSetPixel);
        }
    }
}

/*
 * @brief       Draws a level wall on the screen.
 * @param wall  The index of the wall in the screenImage.
 */
static void clearWall(uint8_t wall)
{
    xy_point point0, point1;

    point0.y = screenImage.walls[wall].yPos+screenDynamics.yShift;
    point1.y = screenImage.walls[wall].yPos+screenDynamics.yShift;
    point0.x = 0;
    point1.x = X_WIDTH-1;

    glcdDrawLine(point0, point1, &glcdClearPixel);
}

/*
 * @brief   Calculate the ball acceleration according to the accelerometer data.
 */
static void calcBallAcc(void)
{
    if (input.accX >= X_MID-ACC_DELTA &&
        input.accX <= X_MID+ACC_DELTA)
    {
       screenDynamics.ballDynamics.xAcc = 0;
       return;
    }

    if (input.accX > 0x81)
    {
        screenDynamics.ballDynamics.xAcc = 1;
        return;
    }

    screenDynamics.ballDynamics.xAcc = -1;
}

/*
 * @brief   This function performs collision detection for the ball and
 *          sets the balls next position accordingly.
 * @return  On game over 1 is returned, else 0.
 */
static uint8_t updateBallPos(void)
{
    uint8_t xCollisionL = 0;
```

```
uint8_t xCollisionR = 0;
uint8_t yCollision = 0;

/* GAME OVER */
if (screenImage.ball.y-BALL_RADIUS == TOP)
    return 1;

/* Wall collision detection */
for (uint8_t w = 0; w < WALLS_ON_SCREEN; w++)
{
    /* Check if wall is on ball level */
    if (screenImage.walls[w].yPos >= screenImage.ball.y-BALL_RADIUS &&
        screenImage.walls[w].yPos <= screenImage.ball.y+BALL_RADIUS+1)
    {
        for (uint8_t p = 0; p < WALL_POINTS; p += 2)
        {
            /* Detect if a wall is being hit from the top */
            if (screenImage.walls[w].yPos == screenImage.ball.y+BALL_RADIUS+1)
            {
                if (p == WALL_POINTS-1)
                {
                    if (screenImage.walls[w].points[p] != X_WIDTH-1 &&
                        screenImage.walls[w].points[p] <= screenImage.ball.x+BALL_RADIUS)
                    {
                        yCollision = 1;
                        break;
                    }
                }
                else if ((screenImage.walls[w].points[p] < screenImage.ball.x-BALL_RADIUS
↪ &&
                          screenImage.walls[w].points[p+1] > screenImage.ball.x+
↪ BALL_RADIUS) ||
                         (screenImage.walls[w].points[p] >= screenImage.ball.x-BALL_RADIUS
↪  &&
                          screenImage.walls[w].points[p] <= screenImage.ball.x+BALL_RADIUS
↪ ) ||
                         (screenImage.walls[w].points[p+1] >= screenImage.ball.x-
↪ BALL_RADIUS &&
                          screenImage.walls[w].points[p+1] <= screenImage.ball.x+
↪ BALL_RADIUS))
                {
                    yCollision = 1;
                    break;
                }
            }
            /* Detect if a wall is being hit from the side */
            if (p == WALL_POINTS-1)
            {
                if (screenImage.walls[w].points[p] != X_WIDTH-1 &&
                    screenImage.walls[w].points[p] == screenImage.ball.x+BALL_RADIUS+1)
                {
                    xCollisionR = 1;
                    break;
                }
            }
            else if (screenImage.walls[w].points[p] == screenImage.ball.x+BALL_RADIUS+1)
            {
                xCollisionR = 1;
                break;
            }
            else if (screenImage.walls[w].points[p+1] == screenImage.ball.x-BALL_RADIUS-1)
            {
                xCollisionL = 1;
                break;
            }
        }

        break;
    }
}
```

```
    if (screenImage.ball.x-BALL_RADIUS == 0)
        xCollisionL = 1;
    else if (screenImage.ball.x+BALL_RADIUS == X_WIDTH-1)
        xCollisionR = 1;

    if ((!xCollisionL && !xCollisionR) ||
        (xCollisionL && screenDynamics.ballDynamics.xAcc > 0) ||
        (xCollisionR && screenDynamics.ballDynamics.xAcc < 0))
        screenImage.ball.x += screenDynamics.ballDynamics.xAcc;

    if (!yCollision &&
        screenImage.ball.y+BALL_RADIUS < BOTTOM)
        screenImage.ball.y += screenDynamics.ballDynamics.yAcc;

    return 0;
}

/*
 * @brief   Draw the ball to the screen on its current position.
 */
static void drawBall(void)
{
    xy_point ball;
    ball.x = screenImage.ball.x;
    ball.y = screenImage.ball.y+screenDynamics.yShift;

    xy_point p0 = {ball.x-BALL_RADIUS, ball.y};
    xy_point p1 = {ball.x+BALL_RADIUS, ball.y};
    glcdDrawLine(p0, p1, &glcdSetPixel);

    for (uint8_t l = 1; l <= BALL_RADIUS; l++)
    {
        glcdDrawLine((xy_point) {ball.x+l-BALL_RADIUS-1, ball.y-l},
                     (xy_point) {ball.x-l+BALL_RADIUS+1, ball.y-l}, &glcdSetPixel);
        glcdDrawLine((xy_point) {ball.x+l-BALL_RADIUS-1, ball.y+l},
                     (xy_point) {ball.x-l+BALL_RADIUS+1, ball.y+l}, &glcdSetPixel);
    }
}

/*
 * @brief   Erase the ball from the screen.
 */
static void clearBall(void)
{
    xy_point ball;
    ball.x = screenImage.ball.x;
    ball.y = screenImage.ball.y+screenDynamics.yShift;

    xy_point p0 = {ball.x-BALL_RADIUS, ball.y};
    xy_point p1 = {ball.x+BALL_RADIUS, ball.y};
    glcdDrawLine(p0, p1, &glcdClearPixel);

    for (uint8_t l = 1; l <= BALL_RADIUS; l++)
    {
        glcdDrawLine((xy_point) {ball.x+l-BALL_RADIUS-1, ball.y-l},
                     (xy_point) {ball.x-l+BALL_RADIUS+1, ball.y-l}, &glcdClearPixel);
        glcdDrawLine((xy_point) {ball.x+l-BALL_RADIUS-1, ball.y+l},
                     (xy_point) {ball.x-l+BALL_RADIUS+1, ball.y+l}, &glcdClearPixel);
    }
}

/*
 * @brief   Check if the current score belongs in the highscore
 *          table and place it on the appropriate position.
 */
static void enterHighScore(void)
{
    int8_t p = PLAYERNUM-1;
```

```
    while (p >= 0 && playerData.currScore > playerData.highScore[p].score)
    {
        if (p == PLAYERNUM-1)
        {
            playerData.highScore[PLAYERNUM-1].player = playerData.currPlayer;
            playerData.highScore[PLAYERNUM-1].score = playerData.currScore;
        }
        else
        {
            playerData.highScore[p+1].player = playerData.highScore[p].player;
            playerData.highScore[p+1].score = playerData.highScore[p].score;
            playerData.highScore[p].player = playerData.currPlayer;
            playerData.highScore[p].score = playerData.currScore;
        }
        p--;
    }
}
```

A.2 ADC

Listing 7: ../Application/adc/adc.h

```
/**
 *
 * @file adc.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-10-27
 *
 * Header file for the ADC driver.
 *
 */

#ifndef __ADC__
#define __ADC__

#include <stdint.h>

/**
 * @brief       Initialize the ADC driver.
 */
void adc_init(void);

/**
 * @brief           Set the callback functions for the ADC ISR.
 * @param _difCB    The callback function processing the adc value from the differential
    ↪ channel.
 * @param _potCB    The callback function processing the adc value from the potentiometer.
 */
void adc_setCallbacks(void (*_difCB)(uint8_t adc), void (*_potCB)(uint8_t adc));

#endif
```

Listing 8: ../Application/adc/adc.c

```
/**
 *
 * @file adc.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-10-27
 *
 * Implementation of the ADC driver.
 *
 */

#include <avr/io.h>
#include <avr/interrupt.h>
```

```c
#include <stdint.h>

#include <adc.h>

/* Sample ADC3 and ADC2 in differential mode with 200x amplification */
#define ADMUX_DIF    (1<<MUX3)|(1<<MUX2)|(1<<MUX1)|(1<<MUX0)
/* Set timer frequency to ~200Hz if used with a prescaler of 1024, so every channel gets read
    ↪ every ~10ms */
#define OCV          78

enum adc_state {DIF, POT};
static volatile enum adc_state state;

static void (*difCB)(uint8_t adc);
static void (*potCB)(uint8_t adc);

/**
 * @brief       Initialize the ADC driver.
 */
void adc_init(void)
{
    /*******************
     * Setup timer 0 A *
     *******************/

    OCR0A = OCV;
    TCNT0 = 0;
    /* Enable output compare interrupt B */
    TIMSK0 |= (1<<OCIE0A);
    /* Set timer to CTC mode and set prescaler to 1024 */
    TCCR0A |= (1<<WGM01);
    TCCR0B |= (1<<CS02)|(1<<CS00);

    /*****************
     * Setup the ADC *
     *****************/

    /* Set voltage reference to AVCC */
    ADMUX |= (1<<REFS0);
    /* Enable auto triggering, enable ADC interrupt and use 128 as prescaler */
    ADCSRA |= (1<<ADATE)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
    /* Select timer0 compare match A as auto trigger source  */
    ADCSRB |= (1<<ADTS1)|(1<<ADTS0);

    state = DIF;
}

/**
 * @brief           Set the callback functions for the ADC ISR.
 * @param _difCB    The callback function processing the adc value from the differential
    ↪ channel.
 * @param _potCB    The callback function processing the adc value from the potentiometer.
 */
void adc_setCallbacks(void (*_difCB)(uint8_t adc), void (*_potCB)(uint8_t adc))
{
    difCB = _difCB;
    potCB = _potCB;
}

/**
 * @brief  Read the ADC value, pass it to the appropriate callback, disable the ADC and
    ↪ switch to the next channel.
 */
ISR(ADC_vect, ISR_BLOCK)
{
    uint16_t adc_res = ADC;
    ADCSRA &= ~(1<<ADEN);

    if (DIF == state)
    {
```

```
        ADMUX &= ~(ADMUX_DIF);
        state = POT;
        sei();
        difCB(adc_res>>2);
    }
    else if (POT == state)
    {
        ADMUX |= ADMUX_DIF;
        state = DIF;
        sei();
        potCB(adc_res>>2);
    }
}

/**
 * @brief   Enable the ADC.
 */
ISR(TIMER0_COMPA_vect, ISR_BLOCK)
{
    ADCSRA |= (1<<ADEN);
}
```

A.3 GLCD

Listing 9: ../Application/glcd/glcd.h

```
/**
 *
 * @file glcd.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-13
 *
 * Header file for the glcd module.
 *
 */

#ifndef __GLCD__
#define __GLCD__

#include <avr/pgmspace.h>
#include <stdint.h>
#include <font.h>

#define GLCD_FILL   0xff
#define GLCD_CLEAR  0x00

/**
 * @brief       Initialize the glcd module.
 */
void glcdInit(void);

/**
 * @brief   Set the specified pixel.
 * @param x X coordinate of the pixel.
 * @param y Y coordinate of the pixel.
 */
void glcdSetPixel(const uint8_t x, const uint8_t y);

/**
 * @brief   Clear the specified pixel.
 * @param x X coordinate of the pixel.
 * @param y Y coordinate of the pixel.
 */
void glcdClearPixel(const uint8_t x, const uint8_t y);

/**
 * @brief   Invert the specified pixel.
```

```
 * @param x X coordinate of the pixel.
 * @param y Y coordinate of the pixel.
 */
void glcdInvertPixel(const uint8_t x, const uint8_t y);

typedef struct xy_point_t
{
    uint8_t x, y;
} xy_point;

/**
 * @brief           Draws a line between two points.
 * @param p1        First endpoint.
 * @param p2        Second endpoint.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawLine(const xy_point p1, const xy_point p2,
                  void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Draws a rectangle, specified by two opposite corners.
 * @param p1        First corner.
 * @param p2        Second corner.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawRect(const xy_point p1, const xy_point p2,
                  void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Fill screen with the specified pattern.
 * @param pattern   The pattern.
 */
void glcdFillScreen(const uint8_t pattern);

/**
 * @brief           Set the y-shift value.
 * @param yshift    The y-shift value.
 */
void glcdSetYShift(uint8_t yshift);

/**
 * @brief           Get the current y-shift value.
 * @return          The current y-shift value.
 */
uint8_t glcdGetYShift(void);

/**
 * @brief           Draws a circle at the given centerpoint with given radius.
 * @param c         The centerpoint.
 * @param radius    The radius.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawCircle(const xy_point c, const uint8_t radius,
                    void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Draws a vertical line.
 * @param x         X postion for the line.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawVertical(const uint8_t x,
                      void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Draws a horizontal line.
 * @param y         Y postion for the line.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawHorizontal(const uint8_t y,
                        void (*drawPx)(const uint8_t, const uint8_t));
```

```
/**
 * @brief           Draws a filled rectangle, specified by two opposite corners.
 * @param p1        First corner.
 * @param p2        Second corner.
 * @param drawPx    Pixel draw function.
 */
void glcdFillRect(const xy_point p1, const xy_point p2,
                  void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Draws a character at a specific position.
 * @param c         The character.
 * @param p         The position.
 * @param f         The font.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawChar(const char c, const xy_point p, const font* f,
                  void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Draws text at a specific position.
 * @param text      The text.
 * @param p         The position.
 * @param f         The font.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawText(const char *text, const xy_point p, const font* f,
                  void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief           Draws text stored in program memory at a specific position.
 * @param text      The text stored in program memory.
 * @param p         The position.
 * @param f         The font.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawTextPgm(PGM_P text, const xy_point p, const font* f,
                     void (*drawPx)(const uint8_t, const uint8_t));

#endif
```

Listing 10: ../Application/glcd/glcd.c

```
/**
 *
 * @file glcd.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-13
 *
 * Implementation of the glcd module.
 *
 */

#include <avr/pgmspace.h>

#include <stdint.h>
#include <stdlib.h>

#include <font.h>
#include <hal_glcd.h>
#include <glcd.h>

#define X_MAX    128
#define Y_MAX    64

/* Static functions */
static void drawLineLow(const xy_point p1, const xy_point p2,
                        void (*drawPx)(const uint8_t, const uint8_t));
```

```
static void drawLineHigh(const xy_point p1, const xy_point p2,
                         void (*drawPx)(const uint8_t, const uint8_t));

/**
 * @brief       Initialize the glcd module.
 */
void glcdInit(void)
{
    halGlcdInit();
}

/**
 * @brief   Set the specified pixel.
 * @param x X coordinate of the pixel.
 * @param y Y coordinate of the pixel.
 */
void glcdSetPixel(const uint8_t x, const uint8_t y)
{
    uint8_t page;

    halGlcdSetAddress(x, y>>3);
    page = halGlcdReadData();
    halGlcdSetAddress(x, y>>3);
    halGlcdWriteData(page | (1 << (y & 7)));
}

/**
 * @brief   Clear the specified pixel.
 * @param x X coordinate of the pixel.
 * @param y Y coordinate of the pixel.
 */
void glcdClearPixel(const uint8_t x, const uint8_t y)
{
    uint8_t page;

    halGlcdSetAddress(x, y>>3);
    page = halGlcdReadData();
    halGlcdSetAddress(x, y>>3);
    halGlcdWriteData(page & ~(1 << (y & 7)));
}

/**
 * @brief   Invert the specified pixel.
 * @param x X coordinate of the pixel.
 * @param y Y coordinate of the pixel.
 */
void glcdInvertPixel(const uint8_t x, const uint8_t y)
{
    uint8_t page;

    halGlcdSetAddress(x, y>>3);
    page = halGlcdReadData();
    halGlcdSetAddress(x, y>>3);
    halGlcdWriteData(page ^ (1 << (y & 7)));
}

/**
 * @brief           Draws a line between two points.
 * @param p1        First endpoint.
 * @param p2        Second endpoint.
 * @param drawPx    Pixel draw function.
 */
/* Bresenham's line algorithm: https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm */
void glcdDrawLine(const xy_point p1, const xy_point p2,
                  void (*drawPx)(const uint8_t, const uint8_t))
{
    if (abs(p2.y - p1.y) < abs(p2.x - p1.x))
    {
        /* Reverse coordinates */
        if (p1.x > p2.x)
```

```
            drawLineLow(p2, p1, drawPx);
        else
            drawLineLow(p1, p2, drawPx);
    }
    else
    {
        /* Reverse coordinates */
        if (p1.y > p2.y)
            drawLineHigh(p2, p1, drawPx);
        else
            drawLineHigh(p1, p2, drawPx);
    }
}

/* Plot lines with low gradient */
static void drawLineLow(const xy_point p1, const xy_point p2,
                        void (*drawPx)(const uint8_t, const uint8_t))
{
    int8_t dx, dy, yi, d, y;
    dx = p2.x - p1.x;
    dy = p2.y - p1.y;
    yi = 1;

    /* Falling line */
    if (dy < 0)
    {
        yi = -1;
        dy = -dy;
    }

    /* How many pixels in a line */
    d = 2*dy - dx;
    y = p1.y;

    for (uint8_t x = p1.x; x <= p2.x; x++)
    {
        drawPx(x, y);
        /* Update y coordinate */
        if (d > 0)
        {
            d -= 2*dx;
            y += yi;
        }
        d += 2*dy;
    }
}

/* Plot lines with steep gradient */
static void drawLineHigh(const xy_point p1, const xy_point p2,
                         void (*drawPx)(const uint8_t, const uint8_t))
{
    int8_t dx, dy, xi, d, x;
    dx = p2.x - p1.x;
    dy = p2.y - p1.y;
    xi = 1;

    /* Falling line */
    if (dy < 0)
    {
        xi = -1;
        dx = -dx;
    }

    /* How many pixels in a line */
    d = 2*dx - dy;
    x = p1.x;

    for (uint8_t y = p1.y; y <= p2.y; y++)
    {
        drawPx(x, y);
```

```
        /* Update x coordinate */
        if (d > 0)
        {
            d -= 2*dy;
            x += xi;
        }
        d += 2*dx;
    }
}

/**
 * @brief           Draws a rectangle, specified by two opposite corners.
 * @param p1        First corner.
 * @param p2        Second corner.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawRect(const xy_point p1, const xy_point p2,
                  void (*drawPx)(const uint8_t, const uint8_t))
{
    glcdDrawLine(p1, (xy_point) {p2.x, p1.y}, drawPx);

    /* Only draw second horizontal line if rect higher than 1 */
    if (p1.y > p2.y || p2.y > p1.y)
        glcdDrawLine((xy_point) {p1.x, p2.y}, p2, drawPx);

    /* Only draw vertical edges if rect higher than 2px */
    if (p1.y > p2.y && (p1.y - p2.y) > 2)
    {
        glcdDrawLine((xy_point) {p1.x, p1.y-1}, (xy_point) {p1.x, p2.y+1}, drawPx);
        glcdDrawLine((xy_point) {p2.x, p1.y-1}, (xy_point) {p2.x, p2.y+1}, drawPx);
    }
    else if (p2.y > p1.y && (p2.y - p1.y) > 2)
    {
        glcdDrawLine((xy_point) {p1.x, p1.y+1}, (xy_point) {p1.x, p2.y-1}, drawPx);
        glcdDrawLine((xy_point) {p2.x, p1.y+1}, (xy_point) {p2.x, p2.y-1}, drawPx);
    }
}

/**
 * @brief           Fill screen with the specified pattern.
 * @param pattern   The pattern.
 */
void glcdFillScreen(const uint8_t pattern)
{
    halGlcdFillScreen(pattern);
}

/**
 * @brief           Set the y-shift value.
 * @param yshift    The y-shift value.
 */
void glcdSetYShift(uint8_t yshift)
{
    halGlcdSetYShift(yshift);
}

/**
 * @brief           Get the current y-shift value.
 * @return          The current y-shift value.
 */
uint8_t glcdGetYShift(void)
{
    return halGlcdGetYShift();
}

/**
 * @brief           Draws a circle at the given centerpoint with given radius.
 * @param c         The centerpoint.
 * @param radius    The radius.
 * @param drawPx    Pixel draw function.
```

```
 */
/* Midpoint circle algorithm: https://en.wikipedia.org/wiki/Midpoint_circle_algorithm */
void glcdDrawCircle(const xy_point c, const uint8_t radius,
                    void (*drawPx)(const uint8_t, const uint8_t))
{
    uint8_t x = radius;
    uint8_t y = 0;
    int8_t dx = 1;
    int8_t dy = 1;
    int8_t error = dx - (radius << 1);

    /* Draw circle segments in counterclockwise order starting from (r, 0) */
    while (x >= y)
    {
        /* Draw circle outline */
        drawPx(c.x + x, c.y + y);
        drawPx(c.x + y, c.y + x);
        drawPx(c.x - y, c.y + x);
        drawPx(c.x - x, c.y + y);
        drawPx(c.x - x, c.y - y);
        drawPx(c.x - y, c.y - x);
        drawPx(c.x + y, c.y - x);
        drawPx(c.x + x, c.y - y);

        /* Calculate error function and decide if x/y need to be updated */
        if (error <= 0)
        {
            y++;
            error += dy;
            dy += 2;
        }
        if (error > 0)
        {
            x--;
            dx += 2;
            error += dx - (radius << 1);
        }
    }
}

/**
 * @brief           Draws a vertical line.
 * @param x         X postion for the line.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawVertical(const uint8_t x,
                      void (*drawPx)(const uint8_t, const uint8_t))
{
    glcdDrawLine((xy_point) {x, 0}, (xy_point) {x, Y_MAX-1}, drawPx);
}

/**
 * @brief           Draws a horizontal line.
 * @param y         Y postion for the line.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawHorizontal(const uint8_t y,
                        void (*drawPx)(const uint8_t, const uint8_t))
{
    glcdDrawLine((xy_point) {0, y}, (xy_point) {X_MAX-1, y}, drawPx);
}

/**
 * @brief           Draws a filled rectangle, specified by two opposite corners.
 * @param p1        First corner.
 * @param p2        Second corner.
 * @param drawPx    Pixel draw function.
 */
void glcdFillRect(const xy_point p1, const xy_point p2,
                  void (*drawPx)(const uint8_t, const uint8_t))
```

```
{
    if (p1.y < p2.y)
    {
        for (uint8_t y = p1.y; y <= p2.y; y++)
            glcdDrawLine((xy_point) {p1.x, y}, (xy_point) {p2.x, y}, drawPx);
    }
    else
    {
        for (uint8_t y = p2.y; y <= p1.y; y++)
            glcdDrawLine((xy_point) {p1.x, y}, (xy_point) {p2.x, y}, drawPx);
    }
}

/**
 * @brief           Draws a character at a specific position.
 * @param c         The character.
 * @param p         The position.
 * @param f         The font.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawChar(const char c, const xy_point p, const font* f,
                  void (*drawPx)(const uint8_t, const uint8_t))
{
    if (c < f->startChar || c > f->endChar)
        return;

    /* Calculate charcter offset for font table */
    uint16_t charIndex = ((c - f->startChar) * f->width);
    for (uint8_t pn = 0; pn < f->width; pn++)
    {
        /* Read character from PROGMEM */
        char page = pgm_read_byte(&(f->font[charIndex+pn]));
        for (uint8_t y = 0; y < 8; y++)
        {
            if (page & (1<<y))
                drawPx(p.x+pn, p.y+y-7);
        }
    }
}

/**
 * @brief           Draws text at a specific position.
 * @param text      The text, must be null terminated.
 * @param p         The position.
 * @param f         The font.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawText(const char *text, const xy_point p, const font* f,
                  void (*drawPx)(const uint8_t, const uint8_t))
{
    uint8_t x = p.x;

    for (uint8_t c = 0; c < X_MAX/f->charSpacing; c++)
    {
        if (text[c] == '\0')
            return;

        glcdDrawChar(text[c], (xy_point) {x, p.y}, f, drawPx);
        x += f->charSpacing;
    }
}

/**
 * @brief           Draws text stored in program memory at a specific position.
 * @param text      The text stored in program memory, must be null terminated.
 * @param p         The position.
 * @param f         The font.
 * @param drawPx    Pixel draw function.
 */
void glcdDrawTextPgm(PGM_P text, const xy_point p, const font* f,
```

```
                    void (*drawPx)(const uint8_t, const uint8_t))
{
    uint8_t x = p.x;

    for (uint8_t c = 0; c < X_MAX/f->charSpacing; c++)
    {
        char character = pgm_read_byte(&text[c]);
        if (character == '\0')
            return;

        glcdDrawChar(character, (xy_point) {x, p.y}, f, drawPx);
        x += f->charSpacing;
    }
}
```

Listing 11: hal_glcd.h

```
/**
 *
 * @file hal_glcd.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-13
 *
 * Header file for the glcd driver.
 *
 */

#ifndef __HAL_GLCD__
#define __HAL_GLCD__

#include <stdint.h>

/**
 * @brief       Initialize the glcd driver.
 */
uint8_t halGlcdInit(void);

/**
 * @brief       Set the internal address.
 * @param xCol  X column.
 * @param yPage Y page.
 */
uint8_t halGlcdSetAddress(const uint8_t xCol,
                          const uint8_t yPage);

/**
 * @brief       Write data to the RAM at the currently set address.
 * @param data  The data.
 */
uint8_t halGlcdWriteData(const uint8_t data);

/**
 * @brief       Read data from the RAM at the currently set address.
 * @return      The data.
 */
uint8_t halGlcdReadData(void);

/**
 * @brief           Set the display row address displayed at the top of the screen.
 * @param yShift    The y-shift address.
 */
uint8_t halGlcdSetYShift(uint8_t yShift);

/**
 * @brief       Get the display row address displayed at the top of the screen.
 * @return      The y-shift address.
 */
uint8_t halGlcdGetYShift(void);
```

```
/*
 * @brief           Fills the whole screen with the desired pattern.
 * @param pattern   The pattern for filling the screen.
 */
uint8_t halGlcdFillScreen(uint8_t pattern);

#endif
```

Listing 12: hal_glcd.c

```
/**
 *
 * @file hal_glcd.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-13
 *
 * Implementation of the glcd driver.
 *
 */

#include <avr/io.h>
#include <stdint.h>

#include <hal_glcd.h>

#define GLCD_CTRL_PORT        PORTE
#define GLCD_CTRL_DDR         DDRE
#define GLCD_CTRL_RS          PE4
#define GLCD_CTRL_RW          PE5
#define GLCD_CTRL_EN          PE6
#define GLCD_CTRL_CS0         PE2
#define GLCD_CTRL_CS1         PE3
#define GLCD_CTRL_RESET       PE7

#define GLCD_DATA_PORT        PORTA
#define GLCD_DATA_DDR         DDRA
#define GLCD_DATA_PIN         PINA

#define GLCD_STATUS_BUSY      PA7
#define GLCD_STATUS_DISP      PA5
#define GLCD_STATUS_RESET     PA4

#define GLCD_CMD_ON           0x3f
#define GLCD_CMD_OFF          0x3e
#define GLCD_CMD_SET_ADDR     0x40
#define GLCD_CMD_SET_PAGE     0xb8
#define GLCD_CMD_DISP_START   0xc0

#define MAX_X_CHIP   64
#define MAX_X        128
#define MAX_Y        64

#define busy_wait()             \
        __asm__ __volatile__(   \
                "nop\n\t"       \
                "nop\n\t"       \
                "nop\n\t"       \
                "nop\n\t" ::)

typedef enum {
    CONTROLLER_0 = (1<<GLCD_CTRL_CS1),
    CONTROLLER_1 = (1<<GLCD_CTRL_CS0),
    CONTROLLER_B = (1<<GLCD_CTRL_CS1)|(1<<GLCD_CTRL_CS0)
} controller_t;

static struct
{
    uint8_t x;
    uint8_t y;
```

```
    uint8_t controller;
    uint8_t yShift;
} internalAddr;

static void halGlcdCtrlWriteData(const controller_t controller,
                                 const controller_t data);
static uint8_t halGlcdCtrlReadData(const controller_t controller);
static uint8_t halGlcdCtrlReadStatus(const controller_t controller);
static void halGlcdCtrlWriteCmd(const controller_t controller,
                                const uint8_t data);
static void halGlcdCtrlSetAddress(const controller_t controller,
                                  const uint8_t x,
                                  const uint8_t y);
static void halGlcdCtrlBusyWait(const controller_t controller);
static void halGlcdCtrlSetRAM(const controller_t controller, const uint8_t pattern);

/**
 * @brief       Initialize the glcd driver.
 */
uint8_t halGlcdInit(void)
{
    GLCD_DATA_PORT = 0;
    GLCD_DATA_DDR = 0xff;
    GLCD_CTRL_PORT &= ~0xfc;
    GLCD_CTRL_DDR |= 0xfc;

    /* Perform reset */
    GLCD_CTRL_PORT |= (1<<GLCD_CTRL_RESET);
    GLCD_CTRL_PORT &= ~(CONTROLLER_B);

    halGlcdCtrlWriteCmd(CONTROLLER_0, GLCD_CMD_ON);
    halGlcdCtrlWriteCmd(CONTROLLER_1, GLCD_CMD_ON);
    halGlcdCtrlWriteCmd(CONTROLLER_0, GLCD_CMD_DISP_START);
    halGlcdCtrlWriteCmd(CONTROLLER_1, GLCD_CMD_DISP_START);
    halGlcdFillScreen(0x00);
    halGlcdSetYShift(0);
    halGlcdSetAddress(0, 0);

    return 0;
}

/**
 * @brief       Set the internal address.
 * @param xCol  X column.
 * @param yPage Y page.
 */
uint8_t halGlcdSetAddress(const uint8_t xCol,
                          const uint8_t yPage)
{
    internalAddr.x = xCol & (MAX_X-1);
    internalAddr.y = yPage & 7;

    if (xCol < MAX_X_CHIP)
        internalAddr.controller = CONTROLLER_0;
    else
        internalAddr.controller = CONTROLLER_1;


    halGlcdCtrlSetAddress(internalAddr.controller,
                          internalAddr.x & (MAX_X_CHIP-1), internalAddr.y);

    return 0;
}

/**
 * @brief       Write data to the RAM at the currently set address. The x address is post-
    ↪ incremented.
 * @param data  The data.
 */
uint8_t halGlcdWriteData(const uint8_t data)
```

```
{
    halGlcdCtrlWriteData(internalAddr.controller, data);

    if (internalAddr.x == MAX_X-1)
    {
        internalAddr.controller = CONTROLLER_0;
        halGlcdCtrlSetAddress(CONTROLLER_0, 0, internalAddr.y);
    }
    else if (internalAddr.x == MAX_X_CHIP-1)
    {
        internalAddr.controller = CONTROLLER_1;
        halGlcdCtrlSetAddress(CONTROLLER_1, 0, internalAddr.y);
    }

    internalAddr.x = (internalAddr.x+1) & (MAX_X-1);

    return 0;
}

/**
 * @brief       Read data from the RAM at the currently set address. The x address is post
    ↪ incremented.
 * @return      The data.
 */
uint8_t halGlcdReadData(void)
{
    /* Dummy read necessary */
    halGlcdCtrlReadData(internalAddr.controller);
    halGlcdCtrlSetAddress(internalAddr.controller, internalAddr.x, internalAddr.y);
    uint8_t data = halGlcdCtrlReadData(internalAddr.controller);

    if (internalAddr.x == MAX_X-1)
    {
        internalAddr.controller = CONTROLLER_0;
        halGlcdCtrlSetAddress(CONTROLLER_0, 0, internalAddr.y);
    }
    else if (internalAddr.x == MAX_X_CHIP-1)
    {
        internalAddr.controller = CONTROLLER_1;
        halGlcdCtrlSetAddress(CONTROLLER_1, 0, internalAddr.y);
    }

    internalAddr.x = (internalAddr.x+1) & (MAX_X-1);

    return data;
}

/**
 * @brief           Set the display row address displayed at the top of the screen.
 * @param yShift    The y-shift address.
 */
uint8_t halGlcdSetYShift(uint8_t yShift)
{
    halGlcdCtrlWriteCmd(CONTROLLER_0, GLCD_CMD_DISP_START | (yShift & (MAX_Y-1)));
    halGlcdCtrlWriteCmd(CONTROLLER_1, GLCD_CMD_DISP_START | (yShift & (MAX_Y-1)));
    internalAddr.yShift = yShift & (MAX_Y-1);

    return 0;
}

/**
 * @brief       Get the display row address displayed at the top of the screen.
 * @return      The y-shift address.
 */
uint8_t halGlcdGetYShift(void)
{
    return internalAddr.yShift;
}

/*
```

```
 * @brief           Fills the whole screen with the desired pattern.
 * @param pattern   The pattern for filling the screen.
 */
uint8_t halGlcdFillScreen(uint8_t pattern)
{
    halGlcdCtrlSetRAM(CONTROLLER_0, pattern);
    halGlcdCtrlSetRAM(CONTROLLER_1, pattern);

    return 0;
}

/*
 * @brief               Writes one byte of data to the selected RAM controller(s).
 * @param controller    The selected controller(s).
 * @param data          The data byte to write.
 */
static void halGlcdCtrlWriteData(const controller_t controller,
                                 const uint8_t data)
{
    halGlcdCtrlBusyWait(controller);

    /* Prepare for data write access */
    GLCD_DATA_PORT = data;
    GLCD_CTRL_PORT = (GLCD_CTRL_PORT & (1<<GLCD_CTRL_RESET)) | (1<<GLCD_CTRL_RS) | controller;
    busy_wait();
    GLCD_CTRL_PORT |= (1<<GLCD_CTRL_EN);
    busy_wait();

    GLCD_CTRL_PORT &= ~((1<<GLCD_CTRL_EN)|CONTROLLER_B);
}

/*
 * @brief               Read one byte of data.
 * @param controller    The selected controller.
 * @return              The read byte.
 */
static uint8_t halGlcdCtrlReadData(const controller_t controller)
{
    uint8_t data;

    halGlcdCtrlBusyWait(controller);

    /* Set data port to input */
    GLCD_DATA_DDR = 0;

    /* Prepare for data read access */
    GLCD_CTRL_PORT = (GLCD_CTRL_PORT & (1<<GLCD_CTRL_RESET)) | (1<<GLCD_CTRL_RW) | (1<<
    ↪ GLCD_CTRL_RS) | controller;
    busy_wait();
    GLCD_CTRL_PORT |= (1<<GLCD_CTRL_EN);
    busy_wait();

    data = GLCD_DATA_PIN;

    GLCD_CTRL_PORT &= ~((1<<GLCD_CTRL_EN)|CONTROLLER_B);

    /* Restore initial pin states */
    GLCD_DATA_DDR = 0xff;

    return data;
}

/*
 * @brief               Read the status byte.
 * @param controller    The selected controller.
 * @return              The read byte.
 */
static uint8_t halGlcdCtrlReadStatus(const controller_t controller)
{
    uint8_t status;
```

```
    /* Set data port to input */
    GLCD_DATA_DDR &= ~((1<<GLCD_STATUS_BUSY)|(1<<GLCD_STATUS_DISP)|(1<<GLCD_STATUS_RESET));

    /* Prepare for status read access */
    GLCD_CTRL_PORT = (GLCD_CTRL_PORT & (1<<GLCD_CTRL_RESET)) | (1<<GLCD_CTRL_RW) | controller;
    GLCD_CTRL_PORT |= (1<<GLCD_CTRL_EN);

    status = GLCD_DATA_PIN & ((1<<GLCD_STATUS_BUSY)|(1<<GLCD_STATUS_DISP)|(1<<
    ↪ GLCD_STATUS_RESET));

    GLCD_CTRL_PORT &= ~((1<<GLCD_CTRL_EN)|CONTROLLER_B);

    /* Restore initial pin states */
    GLCD_DATA_DDR |= (1<<GLCD_STATUS_BUSY)|(1<<GLCD_STATUS_DISP)|(1<<GLCD_STATUS_RESET);

    return status;
}

/*
 * @brief               Write a command byte.
 * @param controller    The selected controller.
 * @param data          The command to write.
 */
static void halGlcdCtrlWriteCmd(const controller_t controller,
                                const uint8_t data)
{
    halGlcdCtrlBusyWait(controller);

    /* Prepare for data write access */
    GLCD_DATA_PORT = data;
    GLCD_CTRL_PORT = (GLCD_CTRL_PORT & (1<<GLCD_CTRL_RESET)) | controller;
    busy_wait();
    GLCD_CTRL_PORT |= (1<<GLCD_CTRL_EN);
    busy_wait();

    GLCD_CTRL_PORT &= ~((1<<GLCD_CTRL_EN)|CONTROLLER_B);
}

/*
 * @brief               Set x and y RAM address on the selected controller(s).
 * @param controller    The controller(s) to check.
 * @param x             The column adress.
 * @param y             The page number.
 */
static void halGlcdCtrlSetAddress(const controller_t controller,
                                  const uint8_t x,
                                  const uint8_t y)
{
    halGlcdCtrlWriteCmd(controller, GLCD_CMD_SET_ADDR | x);
    halGlcdCtrlWriteCmd(controller, GLCD_CMD_SET_PAGE | y);
}

/*
 * @brief               Check if the controller is busy and wait until it is ready.
 * @param controller    The controller to check.
 */
static void halGlcdCtrlBusyWait(const controller_t controller)
{
    uint8_t status;
    do
    {
        status = halGlcdCtrlReadStatus(controller);
    }
    while((status & ((1<<GLCD_STATUS_BUSY)|(1<<GLCD_STATUS_RESET))) != 0);
}

/*
 * @brief               Sets all pages of the given controller to the provided pattern.
 * @param controller    The controller to clear.
```

```
 * @param pattern       The pattern to write to the RAM pages.
 */
void halGlcdCtrlSetRAM(const controller_t controller, const uint8_t pattern)
{
    for (uint8_t y = 0; y < 8; y++)
    {
        halGlcdCtrlSetAddress(controller, 0, y);

        for (uint8_t x = 0; x < MAX_X_CHIP; x++)
            halGlcdCtrlWriteData(controller, pattern);
    }
}
```

A.4 UART

Listing 13: hal_wt41_fc_uart.h

```
/**
 *
 * @file hal_wt41_fc_uart.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-10-31
 *
 * Header file for the WT41 HAL module.
 *
 */

#ifndef __HAL_WT41_FC_UART__
#define __HAL_WT41_FC_UART__

#include <stdint.h>
#include <util.h>

/**
 * @brief               Initialize the WT41 HAL module.
 * @param sndCallback   This callback gets called when a character is sent to the WT41.
 * @param rcvCallback   This callback gets called for every character received from the WT41.
 */
error_t halWT41FcUartInit(
        void (*sndCallback)(void),
        void (*rcvCallback)(uint8_t)
    );

/**
 * @brief       Sends a byte to the WT41 bluetooth module.
 * @param byte  The byte to be sent.
 */
error_t halWT41FcUartSend(uint8_t byte);

#endif
```

Listing 14: hal_et41_fc_uart.c

```
/**
 *
 * @file hal_wt41_fc_uart.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-10-31
 *
 * Implementation of the WT41 HAL module.
 *
 */

#include <avr/io.h>
#include <avr/sleep.h>
#include <avr/interrupt.h>
```

```c
#include <util/atomic.h>
#include <stdint.h>

/* Needed for error_t definition */
#include <util.h>
#include <timer.h>
#include <hal_wt41_fc_uart.h>

#define RESET_TIME  5

#define CTS_PIN PJ3
#define RTS_PIN PJ2
#define RST_PIN PJ5
#define RTS_INT PCINT11

#define RBUF_SZ     32  //must be a power of 2!!!
#define RBUF_HIGH   5
#define RBUF_LOW    RBUF_SZ/2

/* Ringbuffer */
struct ringbuffer
{
    uint8_t start;
    uint8_t end;
    uint8_t len;
    uint8_t data[RBUF_SZ];
};
static volatile struct ringbuffer rbuf = { .start = 0, .end = 0, .len = 0 };

/* Transmission buffer */
static uint8_t tx_byte_buf;

/* Callback functions */
static void (*sendCallback)(void);
static void (*recvCallback)(uint8_t);

/* State variables */
enum sendstate {IDLE, SEND, RES_BLOCK, UDR_BLOCK, HW_BLOCK};

/* Interrupt flags */
static struct
{
    volatile uint8_t wt41_reset_complete:1;
    volatile uint8_t ringbuffer_lock:1;
    volatile uint8_t CTS_state:1;
    volatile enum sendstate send_state;
} flags;

/* Local functions */
static void processRingbuffer(void);
static void resetCompleted(void);

/**
 * @brief               Initialize the WT41 HAL module.
 * @param sndCallback   This callback gets called when a character is sent to the WT41.
 * @param rcvCallback   This callback gets called for every character received from the WT41.
 */
error_t halWT41FcUartInit(
        void (*sndCallback)(void),
        void (*rcvCallback)(uint8_t)
    )
{
    sendCallback = sndCallback;
    recvCallback = rcvCallback;

    flags.send_state = IDLE;

    /***************
     * Setup USART3 *
     ***************/
```

```
    /* Set baudrate to 1M */
    UBRR3 = 1;
    /* Double transmission speed */
    UCSR3A |= (1<<U2X3);
    /* Enable RX & TX interrupts and enable RX & TX */
    UCSR3B |= (1<<RXCIE3)|(1<<RXEN3)|(1<<TXEN3);
    /* Disable user data register interrupt */
    UCSR3B &= ~(1<<UDRIE3);
    /* Frame format: 8 databits, 1 stopbit, no parity */
    UCSR3C |= (1<<UCSZ31)|(1<<UCSZ30);

    /************************
     * Setup HW flow control *
     ************************/

    /* Enable output for CTS and RST and input for RTS */
    PORTJ &= ~((1<<CTS_PIN)|(1<<RST_PIN));
    DDRJ |= (1<<CTS_PIN)|(1<<RST_PIN);
    DDRJ &= ~(1<<RTS_PIN);

    /* Configure PCint for RTS */
    /* Enable PCint 15:8 */
    PCICR |= (1<<PCIE1);
    /* Disable RTS PCint */
    PCMSK1 &= ~(1<<RTS_INT);

    /******************
     * Reset the WT41 *
     ******************/

    /* Configure timer 5 for the reset interval */
    timer_startTimer5(RESET_TIME, TIMER_SINGLE, &resetCompleted);

    return SUCCESS;
}

/**
 * @brief       Sends a byte to the WT41 bluetooth module.
 * @param byte  The byte to be sent.
 */
error_t halWT41FcUartSend(uint8_t byte)
{
    if (IDLE == flags.send_state)
        tx_byte_buf = byte;

    /* Buffer the byte until the wt41 reset has finished */
    if (flags.wt41_reset_complete == 0)
    {
        flags.send_state = RES_BLOCK;
        return ERROR;
    }
    /* High RTS inidcates flow control by WT41 */
    if ((PINJ & (1<<RTS_PIN)) != 0)
    {
        /* Enable pin change interrupt for RTS */
        flags.send_state = HW_BLOCK;
        PCMSK1 |= (1<<RTS_INT);
        return ERROR;
    }
    /* UDR not empty */
    if ((UCSR3A & (1<<UDRE3)) == 0)
    {
        /* Enable user data register interrupt */
        flags.send_state = UDR_BLOCK;
        UCSR3B |= (1<<UDRIE3);
        return ERROR;
    }

    flags.send_state = SEND;
```

```
    /* Copy byte into UART register */
    UDR3 = byte;
    /* Enable user data register interrupt */
    UCSR3B |= (1<<UDRIE3);

    return SUCCESS;
}

/**
 * @brief   Empty the ringbuffer by calling the specified callback on every byte.
 *          This function has to be called in an atomic context.
 */
static void processRingbuffer(void)
{
    uint8_t data;

    do
    {
        data = rbuf.data[rbuf.end];
        rbuf.end = (rbuf.end + 1) & (RBUF_SZ - 1);

        rbuf.len--;
        if (flags.CTS_state == 1 &&
            rbuf.len < RBUF_LOW)
        {
            PORTJ &= ~(1<<CTS_PIN);
            flags.CTS_state = 0;
        }

        sei();
        recvCallback(data);
        cli();
    } while (rbuf.len > 0);
}

/**
 * @brief   Signify the end of the wt41 reset period and send one byte via
 *          if the send function has been called during reset.
 */
static void resetCompleted(void)
{
    cli();
    /* Disable reset */
    PORTJ |= (1<<RST_PIN);
    flags.wt41_reset_complete = 1;
    if (RES_BLOCK == flags.send_state)
    {
        sei();
        halWT41FcUartSend(tx_byte_buf);
        return;
    }
    sei();
}

/**
 * @brief   Handle an incoming byte on the UART by putting it in the ringbuffer.
 */
ISR(USART3_RX_vect, ISR_BLOCK)
{
    rbuf.data[rbuf.start] = UDR3;
    /* Increment the start pointer mod buffer size */
    rbuf.start = (rbuf.start + 1) & (RBUF_SZ - 1);
    rbuf.len++;
    /* Set CTS if buffer capacity low */
    if (flags.CTS_state == 0 &&
        RBUF_SZ - rbuf.len < RBUF_HIGH)
    {
        PORTJ |= (1<<CTS_PIN);
        flags.CTS_state = 1;
    }
```

```
    if (flags.ringbuffer_lock == 0)
    {
        flags.ringbuffer_lock = 1;
        processRingbuffer();
        flags.ringbuffer_lock = 0;
    }
}

/**
 * @brief   Try sending the byte which has been held back by a full buffer.
 */
ISR(USART3_UDRE_vect, ISR_BLOCK)
{
    /* Disable the UDR interrupt */
    UCSR3B &= ~(1<<UDRIE3);
    if (SEND == flags.send_state)
    {
        flags.send_state = IDLE;
        sei();
        sendCallback();
    }
    else if (UDR_BLOCK == flags.send_state)
    {
        sei();
        halWT41FcUartSend(tx_byte_buf);
    }
}

/**
 * @brief   Try sending the byte which has been held back by HW flow control.
 */
ISR(PCINT1_vect, ISR_BLOCK)
{
    /* Disable the PC interrupt */
    PCMSK1 &= ~(1<<RTS_INT);
    sei();
    halWT41FcUartSend(tx_byte_buf);
}
```

A.5 Music

Listing 15: ../Application/music/music.h

```
/**
 *
 * @file music.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-08
 *
 * Header file for the music module.
 *
 */

#ifndef __MUSIC__
#define __MUSIC__

#include <stdint.h>
#include <task.h>

/**
 * @brief   Initialize the music module.
 */
void music_init(void (*mp3DataReqCB)(void));

/**
 * @brief   Play some music from the SD card on the mp3 module.
```

```
 * @return  Return non zero if there is still work to do and 0 if everything is done.
 */
task_state_t music_play(void);

/**
 * @brief           Pass a raw volume value (e.g from a pot) to the module.
 * @param volumeRaw The raw volume value, straight from the ADC.
 */
void music_setVolume(uint8_t volumeRaw);

#endif
```

Listing 16: ../Application/music/music.c

```
/**
 *
 * @file music.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-08
 *
 * Implementation of the music module.
 *
 */

#include <stdint.h>

#include <spi.h>
#include <mp3.h>
#include <sdcard.h>
#include <task.h>

/* dt_himalayas.mp3 */
#define SONG_START  4385760
#define SONG_LENGTH 289872

#define DELTA_VOLUME    50

static uint32_t sdcardBlockAddress = SONG_START;
static uint8_t spiLock = 0;
static uint8_t oldVolume = 3;

static uint8_t scaleVolume(uint8_t volume);

/**
 * @brief   Initialize the music module.
 */
void music_init(void (*mp3DataReqCB)(void))
{
    spiInit();
    while (sdcardInit() != SUCCESS);
    mp3Init(mp3DataReqCB);
}

/**
 * @brief   Play some music from the SD card on the mp3 module.
 * @return  Return non zero if there is still work to do and 0 if everything is done.
 */
task_state_t music_play(void)
{
    sdcard_block_t musicBuffer;

    if (!mp3Busy())
    {
        spiLock = 1;
        if (sdcardReadBlock(sdcardBlockAddress, musicBuffer) == SUCCESS)
        {
            mp3SendMusic(musicBuffer);
            spiLock = 0;
            if (sdcardBlockAddress < SONG_START + SONG_LENGTH)
```

```
                sdcardBlockAddress += BLOCK_SIZE;
            else
                sdcardBlockAddress = SONG_START;
        }
        spiLock = 0;
        return BUSY;
    }
    return DONE;
}

/**
 * @brief           Pass a raw volume value (e.g from a pot) to the module.
 * @param volumeRaw The raw volume value, straight from the ADC.
 */
void music_setVolume(uint8_t volumeRaw)
{
    uint8_t newVolume = scaleVolume(volumeRaw);
    /* Only set the volume if it has changed and if the spi is not used by other functions */
    if (newVolume != oldVolume && spiLock == 0)
    {
        mp3SetVolume(newVolume);
        oldVolume = newVolume;
    }
}

/**
 * @brief           Scale the volume value to an approximate logarithmic scale.
 * @param volume    The raw volume value to scale.
 * @return          The scaled volume value.
 */
static uint8_t scaleVolume(uint8_t volume)
{
    /* Implementation of the log-approximation 1-(1-x)^4 */
    volume = 0xff - volume;
    volume = (volume * volume) >> 8;
    volume = (volume * volume) >> 8;
    return 0xff - volume;
}
```

A.6 Rand

Listing 17: ../Application/rand/rand.h

```
/**
 *
 * @file rand.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-10-26
 *
 * Header file for the PRNG module.
 *
 */

#ifndef __RAND__
#define __RAND__

#include <stdint.h>

#define POLYNOMIAL 0x80E3

/**
 * @brief       Shift the LFSR to the right, shifting in the LSB of the parameter. Usually not
    ↪ called directly.
 * @param in    The bit to shift into the LFSR.
 * @return      The bit shifted out of the LFSR.
 */
uint8_t rand_shift(uint8_t in);
```

```
/**
 * @brief       Feed one random bit to the LFSR (reseeding).
 * @param in    The random bit to feed into the LFSR.
 */
void rand_feed(uint8_t in);

/**
 * @brief       Get one bit of random data from the LFSR.
 * @return      A random bit.
 */
uint8_t rand1(void);

/**
 * @brief       Get a random 16-bit number.
 * @return      A random 16-bit number.
 */
uint16_t rand16(void);

#endif
```

Listing 18: ../Application/rand/rand.c

```
/**
 *
 * @file rand.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-10-26
 *
 * Implementation of the PRNG module.
 *
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include <stdint.h>

#include <rand.h>

/* The LFSR used for computing pseudo-random numbers. */
static uint16_t lfsr = 1;

/* The polynomial for the PRNG */
static const uint16_t poly = POLYNOMIAL;

/**
 * @brief       Shift the LFSR to the right, shifting in the LSB of the parameter. Usually not
    ↪ called directly.
 * @param in    The bit to shift into the LFSR.
 * @return      The bit shifted out of the LFSR.
 */
uint8_t rand_shift(uint8_t in)
{
    uint8_t out = 0;

    ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
    {
        asm volatile
        (
            "lsr    %2"                 "\n\t" /* MSB(lfsr) := LSB(in) */
            "ror    %B1"                "\n\t" /* lsfr >> 1 */
            "ror    %A1"                "\n\t"
            "brcc   L_end%="            "\n\t" /* if out */
            "eor    %A1, %A3"           "\n\t" /* lfsr := lfsr xor poly  */
            "eor    %B1, %B3"           "\n\t"
            "L_end%=:"                  "\n\t"
            "rol    %0"                 "\n\t" /* LSB(out) := LSB(lfsr) */
            : "+r" (out), "+r" (lfsr)
```

```
            : "r" (in), "r" (poly)
        );
    }

    return out;
}

/**
 * @brief       Feed one random bit to the LFSR (reseeding).
 * @param in    The random bit to feed into the LFSR.
 */
void rand_feed(uint8_t in)
{
    (void) rand_shift(in);
}

/**
 * @brief       Get one bit of random data from the LFSR.
 * @return      A random bit.
 */
uint8_t rand1()
{
    return rand_shift(0);
}

/**
 * @brief       Get a random 16-bit number.
 * @return      A random 16-bit number.
 */
uint16_t rand16()
{
    uint16_t randnum = 0;

    for (uint8_t i = 0; i < 16; i++)
    {
        randnum |= rand1() << i;
    }

    return randnum;
}
```

A.7 SPI

Listing 19: ../Application/spi/spi.h

```
/**
 *
 * @file spi.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-05
 *
 * Header file for the SPI driver.
 *
 */

#ifndef __SPI__
#define __SPI__

#include <stdint.h>

/* Typedef as in the SD card library */
typedef enum {
    SPI_PRESCALER_128   = 3,
    SPI_PRESCALER_4     = 0,
} spi_prescaler_t;

/**
```

```
 * @brief       Initialize the SPI driver.
 */
void spiInit(void);

/**
 * @brief       Send one byte via SPI.
 * @param data  The byte to send.
 */
void spiSend(uint8_t data);

/**
 * @brief   Receive one byte via SPI.
 * @return  The received byte.
 */
uint8_t spiReceive(void);

/**
 * @brief           Set the SPI prescaler.
 * @param prescaler The chosen prescaler.
 */
void spiSetPrescaler(spi_prescaler_t prescaler);

#endif
```

Listing 20: ../Application/spi/spi.c

```
/**
 *
 * @file spi.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-05
 *
 * Implementation of the SPI driver.
 *
 */

#include <avr/io.h>
#include <stdint.h>

#include <spi.h>

/* SPI pin definitions */
#define DDR_SPI DDRB
#define DD_SS   PB0
#define DD_SCK  PB1
#define DD_MOSI PB2
#define DD_MISO PB3

/**
 * @brief       Initialize the SPI driver.
 */
void spiInit(void)
{
    /* Set MOSI and SCK as output */
    DDR_SPI |= (1<<DD_MOSI)|(1<<DD_SCK);
    /* Set MISO and SS as input */
    DDR_SPI &= ~((1<<DD_MISO)|(1<<DD_SS));
    /* Enable SPI and set master mode */
    SPCR |= (1<<SPE)|(1<<MSTR);
    /* Disable clock polarity and clock phase */
    SPCR &= ~((1<<CPOL)|(1<<CPHA));
}

/**
 * @brief       Send one byte via SPI.
 * @param data  The byte to send.
 */
void spiSend(uint8_t data)
{
```

```
    /* Start sending the byte */
    SPDR = data;
    /* Busy-wait until sending has been finished */
    while (!(SPSR & (1<<SPIF)));
}

/**
 * @brief   Receive one byte via SPI.
 * @return  The received byte.
 */
uint8_t spiReceive(void)
{
    /* Send dummy value and busy-wait for reception to complete */
    SPDR = 0xff;
    while (!(SPSR & (1<<SPIF)));
    /* Read received byte */
    return SPDR;
}

/**
 * @brief           Set the SPI prescaler.
 * @param prescaler The chosen prescaler.
 */
void spiSetPrescaler(spi_prescaler_t prescaler)
{
    SPCR &= ~((1<<SPR1)|(1<<SPR0));
    SPCR |= prescaler;
}
```

A.8 Timer

Listing 21: ../Application/timer/timer.h

```
/**
 *
 * @file timer.h
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-06
 *
 * Header file for the timer module.
 *
 */

#ifndef __TIMER__
#define __TIMER__

#include <stdint.h>

typedef enum {TIMER_SINGLE, TIMER_REPEAT} timer_mode_t;
typedef enum {SUCC, NOT_AVAIL, INVAL} timer_error_t;

/**
 * @brief           Start a timer to run for the specified amount of ms.
 * @param ms        How many milliseconds the timer should run. Must not be bigger than 4194,
    ↪ otherwise INVAL is returned.
 * @param mode      The mode of the timer, wheter it should run once ore periodically.
 * @param _tmrCB    The callback function to be called in the timer ISR. Set to NULL if not
    ↪ needed. This callback can be interrupted at any time.
 * @return          The return value reflects if the setup was successful or if the timer is
    ↪ not available.
 */
timer_error_t timer_startTimer1(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void));
timer_error_t timer_startTimer3(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void));
timer_error_t timer_startTimer4(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void));
timer_error_t timer_startTimer5(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void));

/*
```

```
 * @brief Stops the specified timer to make it available again. If the timer mode was
    ↪ TIMER_SINGLE, it does not have to be stopped.
 */
void timer_stopTimer1(void);
void timer_stopTimer3(void);
void timer_stopTimer4(void);
void timer_stopTimer5(void);

#endif
```

Listing 22: ../Application/timer/timer.c

```
/**
 *
 * @file timer.c
 * @author Jan Nausner <e01614835@student.tuwien.ac.at>
 * @date 2018-11-06
 *
 * Implementation of the timer module.
 *
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/atomic.h>
#include <stdint.h>
#include <stdio.h>

#include <timer.h>

#define F_CPU         (16000000UL)

#define COUNT_1MS     62
#define PRESC_64      3
#define PRESC_256     4
#define PRESC_1024    5

#define MAX_64        262
#define MAX_256       1048
#define MAX_1024      4194

/* Calculate the output compare register value for the desired interval */
#define OCR(T, P)     (((F_CPU/1000)*T)/(P))-1

typedef enum {AVAILABLE, NOT_AVAILABLE} timer_state_t;

typedef struct {
    volatile timer_state_t state;
    timer_mode_t mode;
    void (*callback)(void);
} timer_t;

/* Timer structs */
static timer_t timer1;
static timer_t timer3;
static timer_t timer4;
static timer_t timer5;

/**
 * @brief           Start a timer to run for the specified amount of ms.
 * @param ms        How many milliseconds the timer should run. Must not be bigger than 4194,
    ↪ otherwise INVAL is returned.
 * @param mode      The mode of the timer, wheter it should run once ore periodically.
 * @param _tmrCB    The callback function to be called in the timer ISR. Set to NULL if not
    ↪ needed. This callback can be interrupted at any time.
 * @return          The return value reflects if the setup was successful or if the timer is
    ↪ not available.
 */
timer_error_t timer_startTimer1(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void))
```

```
{
    if (NOT_AVAILABLE == timer1.state)
        return NOT_AVAIL;

    timer1.state = NOT_AVAILABLE;
    timer1.callback = _tmrCB;
    timer1.mode = mode;

    TCNT1 = 0;

    if (ms <= MAX_64)
    {
        OCR1A = OCR(ms, 64);
        TCCR1B = (1<<WGM12)|PRESC_64;
    }
    else if (ms <= MAX_256)
    {
        OCR1A = OCR(ms, 256);
        TCCR1B = (1<<WGM12)|PRESC_256;
    }
    else if (ms <= MAX_1024)
    {
        OCR1A = OCR(ms, 1024);
        TCCR1B = (1<<WGM12)|PRESC_1024;
    }
    else
    {
        timer1.state = AVAILABLE;
        return INVAL;
    }

    TIMSK1 |= (1<<OCIE1A);

    return SUCC;
}

timer_error_t timer_startTimer3(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void))
{
    if (NOT_AVAILABLE == timer3.state)
        return NOT_AVAIL;

    timer3.state = NOT_AVAILABLE;
    timer3.callback = _tmrCB;
    timer3.mode = mode;

    TCNT3 = 0;

    if (ms <= MAX_64)
    {
        OCR3A = OCR(ms, 64);
        TCCR3B = (1<<WGM32)|PRESC_64;
    }
    else if (ms <= MAX_256)
    {
        OCR3A = OCR(ms, 256);
        TCCR3B = (1<<WGM32)|PRESC_256;
    }
    else if (ms <= MAX_1024)
    {
        OCR3A = OCR(ms, 1024);
        TCCR3B = (1<<WGM32)|PRESC_1024;
    }
    else
    {
        timer3.state = AVAILABLE;
        return INVAL;
    }

    TIMSK3 |= (1<<OCIE3A);
```

```
    return SUCC;
}

timer_error_t timer_startTimer4(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void))
{
    if (NOT_AVAILABLE == timer4.state)
        return NOT_AVAIL;

    timer4.state = NOT_AVAILABLE;
    timer4.callback = _tmrCB;
    timer4.mode = mode;

    TCNT4 = 0;

    if (ms <= MAX_64)
    {
        OCR4A = OCR(ms, 64);
        TCCR4B = (1<<WGM42)|PRESC_64;
    }
    else if (ms <= MAX_256)
    {
        OCR4A = OCR(ms, 256);
        TCCR4B = (1<<WGM42)|PRESC_256;
    }
    else if (ms <= MAX_1024)
    {
        OCR4A = OCR(ms, 1024);
        TCCR4B = (1<<WGM42)|PRESC_1024;
    }
    else
    {
        timer4.state = AVAILABLE;
        return INVAL;
    }

    TIMSK4 |= (1<<OCIE4A);

    return SUCC;
}

timer_error_t timer_startTimer5(uint16_t ms, timer_mode_t mode, void (*_tmrCB)(void))
{
    if (NOT_AVAILABLE == timer5.state)
        return NOT_AVAIL;

    timer5.state = NOT_AVAILABLE;
    timer5.callback = _tmrCB;
    timer5.mode = mode;

    TCNT5 = 0;

    if (ms <= MAX_64)
    {
        OCR5A = OCR(ms, 64);
        TCCR5B = (1<<WGM52)|PRESC_64;
    }
    else if (ms <= MAX_256)
    {
        OCR5A = OCR(ms, 256);
        TCCR5B = (1<<WGM52)|PRESC_256;
    }
    else if (ms <= MAX_1024)
    {
        OCR5A = OCR(ms, 1024);
        TCCR5B = (1<<WGM52)|PRESC_1024;
    }
    else
    {
        timer5.state = AVAILABLE;
        return INVAL;
```

```
    }

    TIMSK5 |= (1<<OCIE5A);

    return SUCC;
}

/*
 * @brief Stops the specified timer to make it available again. If the timer mode was
    ↪ TIMER_SINGLE, it does not have to be stopped.
 */
void timer_stopTimer1(void)
{
    TIMSK1 &= ~(1<<OCIE1A);
    timer1.state = AVAILABLE;
}

void timer_stopTimer3(void)
{
    TIMSK3 &= ~(1<<OCIE3A);
    timer3.state = AVAILABLE;
}

void timer_stopTimer4(void)
{
    TIMSK4 &= ~(1<<OCIE4A);
    timer4.state = AVAILABLE;
}

void timer_stopTimer5(void)
{
    TIMSK5 &= ~(1<<OCIE5A);
    timer5.state = AVAILABLE;
}

/*
 * @brief Decide if timer needs to be stopped an call the registered callback.
 */
ISR(TIMER1_COMPA_vect, ISR_BLOCK)
{
    if (TIMER_SINGLE == timer1.mode)
        timer_stopTimer1();

    sei();
    if (timer1.callback != NULL)
        timer1.callback();
}

ISR(TIMER3_COMPA_vect, ISR_BLOCK)
{
    if (TIMER_SINGLE == timer3.mode)
        timer_stopTimer3();

    sei();
    if (timer3.callback != NULL)
        timer3.callback();
}

ISR(TIMER4_COMPA_vect, ISR_BLOCK)
{
    if (TIMER_SINGLE == timer4.mode)
        timer_stopTimer4();

    sei();
    if (timer4.callback != NULL)
        timer4.callback();
}

ISR(TIMER5_COMPA_vect, ISR_BLOCK)
{
```

```
    if (TIMER_SINGLE == timer5.mode)
        timer_stopTimer5();

    sei();
    if (timer5.callback != NULL)
        timer5.callback();
}
```

A.9 Wii User

Listing 23: wii_user.c

```
#include <stdbool.h>
#include <stdlib.h>
#include <util/atomic.h>
#include <wii_user.h>

static uint8_t _state[WII], _leds[WII], _rumbler[WII];

static void (*_rcvButton)(uint8_t, uint16_t);
static void (*_rcvAccel)(uint8_t, uint16_t, uint16_t, uint16_t);

static union
{
        void (*setLedsCallback)(uint8_t, error_t);
        void (*setAccelCallback)(uint8_t, error_t);
        void (*setRumblerCallback)(uint8_t, error_t);
} _union[WII];

static void sndCallback(uint8_t wii)
{
        uint8_t state = _state[wii];
        _state[wii] = 0;
        if (state == 1)         // todo: switch? names for states ???
        {
                if (_union[wii].setLedsCallback)
                        _union[wii].setLedsCallback(wii, SUCCESS);
        }
        else if (state == 2)
        {
                if (_union[wii].setAccelCallback)
                        _union[wii].setAccelCallback(wii, SUCCESS);
        }
        else if (state == 3)
        {
                if (_union[wii].setRumblerCallback)
                        _union[wii].setRumblerCallback(wii, SUCCESS);
        }
}

static void rcvCallback(uint8_t wii, uint8_t length, const uint8_t data[])
{
        if (length > 1)
        {
                if (data[1] == 0x31)
                {
                        if (length != 7)
                                abort();
                        if (_rcvAccel)
                        {
                                uint16_t x = data[4] << 2 | (data[2] & 0x60) >> 5;
                                uint16_t y = data[5] << 1 | (data[3] & 0x20) >> 5;
                                uint16_t z = data[6] << 1 | (data[3] & 0x40) >> 6;
                                _rcvAccel(wii, x, y, z);
                        }
                }
                else
```

```
                {
                        if (data[1] != 0x30)
                                return;
                        if (length != 4)
                                abort();
                }
                if (_rcvButton)
                        _rcvButton(wii, (data[2] & 0x1f) << 8 | (data[3] & 0x9f));
        }
}

error_t wiiUserInit(void (*rcvButton)(uint8_t, uint16_t), void (*rcvAccel)(uint8_t, uint16_t,
    ↪ uint16_t, uint16_t))
{
#ifndef NDEBUG
        ATOMIC_BLOCK(ATOMIC_FORCEON)
        {
                static bool _init;
                if (_init)
                        return ERROR;
                _init = true;
        }
#endif
        _rcvButton = rcvButton;
        _rcvAccel = rcvAccel;
        return wiiBtInit(&sndCallback, &rcvCallback);
}

error_t wiiUserConnect(uint8_t wii, const uint8_t *mac, void (*conCallback)(uint8_t,
    ↪ connection_status_t))
{
        return wiiBtConnect(wii, mac, conCallback);
}

error_t wiiUserSetLeds(uint8_t wii, uint8_t bitmask, void (*setLedsCallback)(uint8_t wii,
    ↪ error_t status))
{
#ifndef NDEBUG
        if (wii >= WII)
                return ERROR;
#endif
        ATOMIC_BLOCK(ATOMIC_FORCEON)
        {
                if (_state[wii])
                        return 1;
                _state[wii] = 1;
        }
        _leds[wii] = bitmask << 4;
        _union[wii].setLedsCallback = setLedsCallback;
        uint8_t data[] = { 0xa2, 0x11, _leds[wii] | _rumbler[wii] };
        uint8_t status = wiiBtSendRaw(wii, sizeof(data), data);
        if (status)
                _state[wii] = 0;
        return status;
}

error_t wiiUserSetAccel(uint8_t wii, uint8_t enable, void (*setAccelCallback)(uint8_t, error_t
    ↪ ))
{
#ifndef NDEBUG
        if (wii >= WII)
                return ERROR;
#endif
        ATOMIC_BLOCK(ATOMIC_FORCEON)
        {
                if (_state[wii])
                        return ERROR;
                _state[wii] = 2;
        }
        _union[wii].setAccelCallback = setAccelCallback;
```

```
        uint8_t data[] = { 0xa2, 0x12, 0x00, 0x31 };
        if (!enable)
            data[3] = 0x30;

        uint8_t status = wiiBtSendRaw(wii, sizeof(data), data);
        if (status)
                _state[wii] = 0;
        return status;
}

error_t wiiUserSetRumbler(uint8_t wii, uint8_t enable, void (*setRumblerCallback)(uint8_t,
    ↪ error_t))
{
#ifndef NDEBUG
        if (wii >= WII)
                return ERROR;
#endif
        ATOMIC_BLOCK(ATOMIC_FORCEON)
        {
                if (_state[wii])
                        return ERROR;
                _state[wii] = 3;
        }
        _union[wii].setRumblerCallback = setRumblerCallback;
        _rumbler[wii] = enable > 0;
        uint8_t data[] = { 0xa2, 0x11, _leds[wii] | _rumbler[wii] };
        uint8_t status = wiiBtSendRaw(wii, sizeof(data), data);
        if (status)
                _state[wii] = 0;
        return status;
}
```