MCVL 2018 – Application 1
Falling Down Ball v1.0

Version	Date	Remark
1.1	Oct. 24	Minor changes to section 2.1.3 and 2.1.4
1.0	Oct. 22	Initial release

Contents

1 General Remarks

Throughout the whole application you must not use busy waiting, except for the $2x16$ LC–Display busy flag polling[1] and SPI transmissions!

Even if you are now programming a "bigger" application keep in mind that you are still working with a microcontroller. Do not waste resources! There is no need to use dynamic memory allocation (malloc/free) in the application and you shall not use it.

No specification is complete! Design decisions have to be documented in the protocol, e.g., what happens if the SD Card is removed during operation? What happens if a string is written to the display that does not fit into the line (or the display)?

If you are unsure if something is left open as a design decisions, or if something is unclear, , ask the staff at mcleitung@ecs.tuwien.ac.at!

Please note that if you want to receive bonus points for additional work, you have to state in the protocol which parts you think are eligible for bonus points and why. Bonus points are awarded on a case by case basis and there is no right for bonus points.

Please recall that bonus point only improve a positive grade and cannot help you pass the course. Therefore, please add fancy stuff only after you have finished the basic assignment.

We highly recommend you to first read the whole specification before starting reading datasheets or even programming! Note that you have to adhere to the Coding Guidelines uploaded to TUWEL.

2 High-Level Specification

This specification document describes a "falling down ball" game that has to be implemented for the bigAVR6 development kit used in the lab. The game has to be controllable with a Wii-Mote, where its accelerometers must be used for direction determination. To provide a better gaming experience, music has to be played in the background, at least throughout the actual gameplay.

2.1 Overview

The external interfaces to the game are shown in Figure 1. These are the "connections to the real world" of the application which consist of the following elements:

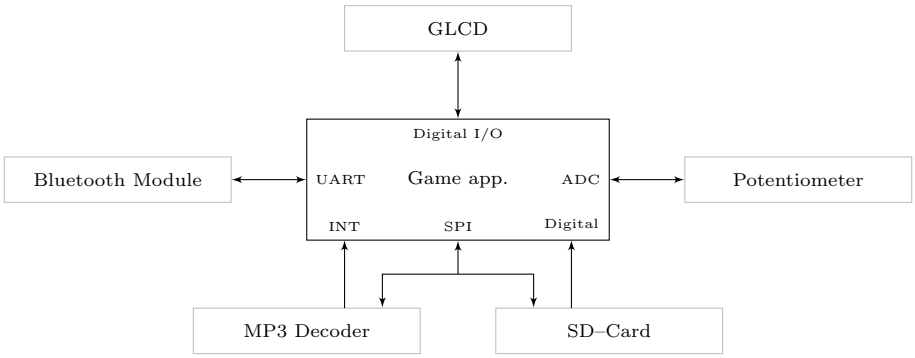

Figure 1: External Interfaces for the application

[1] You are allowed to use *nop* commands and wait for the LCD to get ready after writing a command/data to the LCD that needs less than 50 µs time to execute. You are also allowed to use busy flag polling during the initialization of the LCD.

2.1.1 GLCD

The *GLCD interface* gives feedback to the user via a LC–Display. We do not want to over-specify the UI, as (1) it is not the main focus of this exercise and (2) we think that the freedom gives you the opportunity to be creative and to come up with things we have not thought of.

The GLCD is connected via the designated mounting frame to the bigAVR6.

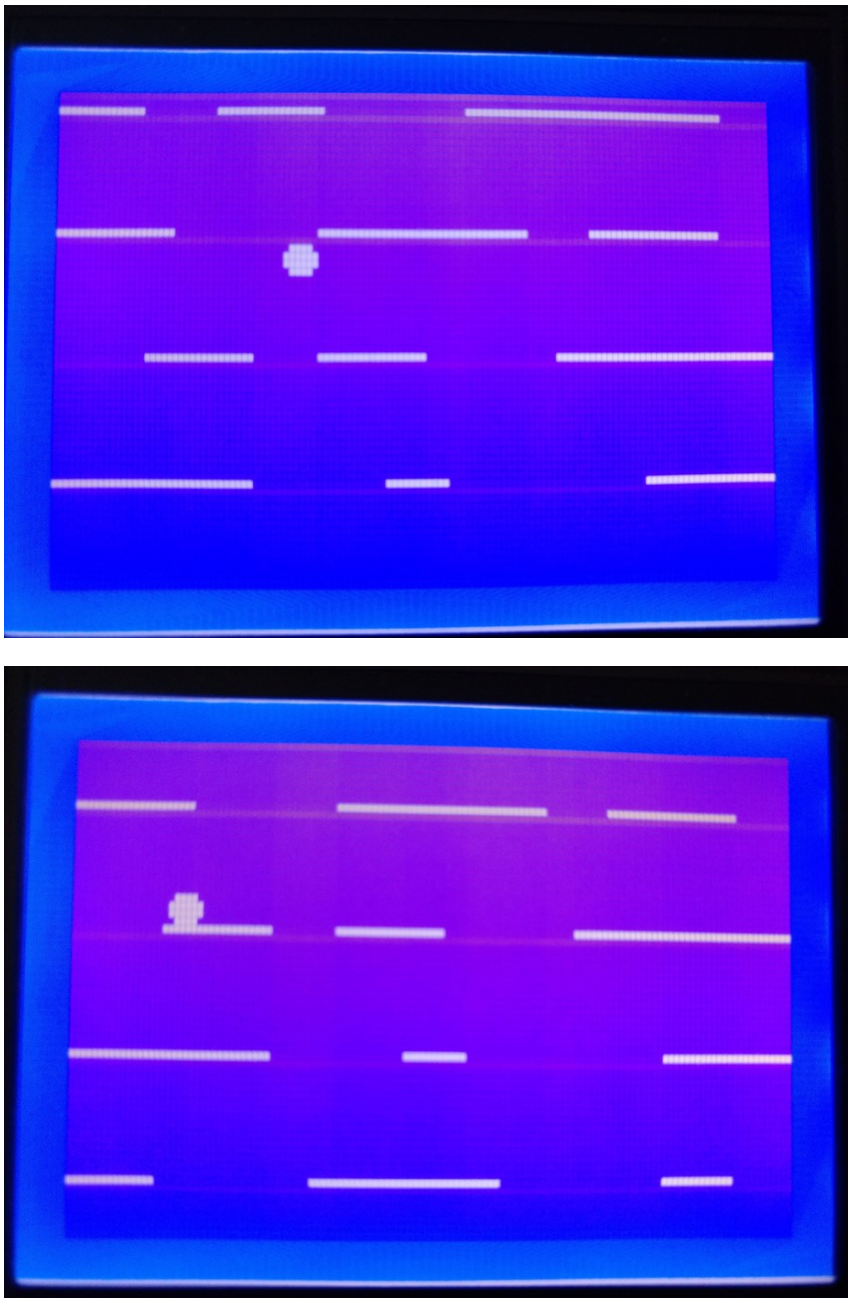

Figure 2: Sample output on the GLCD-Display.

2.1.2 Potentiometer

The potentiometer P5 on the bigAVR6 board should be connected to the ADC on Channel 0, where it will be used to control the volume of the music playback.

Please note that when the volume is turned up very high, the SmartMP3 board can use more power

than certain USB ports can provide. Therefore, if you notice multiple resets of the controller while music playback is active, it might help to reduce the volume (which your lab colleagues will favor) or to use an external power supply.

2.1.3 SD Card

The SD Cards in the lab are filled with a selection of different songs. The sounds are stored as raw data on the SD Card, i.e., there is no file system. A detailed document with the list of sound files, and their alignment on the card, is published in TUWEL under additional material.

You should always use the external SD Card slot on the SmartMP3 add-on board and disconnect the onboard slot via the according DIP switches. The reason for this is that the onboard slot drives the MISO line even if the chip-select is disabled and thus prevents all other communication on the bus.

Read Access to the SD Card is provided as a precompiled library to you.

The SD Card slot on the SmartMP3 extension should be attached to the bigAVR6 board using the pins shown in Table 1. Unfortunately, this cannot be done by a ribbon cable but has to be done by using single connection wires.

	SmartMP3	bigAVR6
SPI	SCK	PB1
	MOSI	PB2
	MISO	PB3
SD Card	$\overline{\text{MMC\_CS}}$	PG1
MP3	$\overline{\text{MP3\_CS}}$	PB0
	$\overline{\text{MP3\_RST}}$	PB4
	BSYNC	PB5
	DREQ	PD0
Power	VCC	VCC
	GND	GND

Table 1: Connection plan of the SmartMP3 extension board

Hint

While all students are using the same pinout, it is always a good idea to check the connections when (re)starting to work in the lab. This can dramatically reduce the debugging needs of a program which "worked the last time".

2.1.4 MP3 Decoder

We use the *VS1011e/VS1053 MP3 Audio Decoder* on the MikroElektronika SmartMP3 add-on board which has a quite extensive datasheet. The datasheets for the chip and the boards can be found in TUWEL in the 'Extension Boards' tab. We provide a library which handles the set-up and communication with the decoder. You only have to feed the raw mp3 data to it, and implement the SPI bus access. It shares the SPI module with the SD Card.

The pinout is shown in Table 1.

2.1.5 Bluetooth

The Bluetooth module is connected to the bigAVR6 board by a ribbon cable attached to the expansion header on *PORTJ*. It provides a serial interface for communication with the Wii-Mote which uses a 1 Mbit UART with hardware flow control for communication. We provide a library which handles the communication, that is transmitted over the serial interface, with the Wii-Mote.

Please ensure that the flow control works, otherwise the module could crash the application.

3 Implementation Remarks

3.1 Support Guide

To help you getting the application done, we provide you with a variety of test programs and libraries. All libraries can be downloaded in TUWEL.

GLCD Library (libglcd): We provide you with a GLCD library to enable a fast start into application programming and easy debugging.

Font Definition (font): To make programming a text-mode on the GLCD easier for you we provide you with a font specification.

SD Card Library (libsdcard): The library for accessing SD Cards. Packed and ready to go.

Source of SD Card Library (srcsdcard): The library for accessing SD Cards. Packed and ready to go. As source files. Feel free to change.

SD Card Track Index (sdcardIndex): This file lists the sound tracks provided on the SD Cards in the lab, giving start position and length (both in bytes).

MP3 Decoder Library (libmp3): The library for handling the MP3 decoder chip. Packed and ready to go.

Source of MP3 Decoder Library (srcmp3): The library for handling the MP3 decoder chip. As source files. Feel free to change.

Wii-Mote Bluetooth Stack (libwiimote): The stack for accessing the Wii-Mote. Packed and ready to go.

Source of Wii-Mote Bluetooth Stack (srcwiimote): The stack for accessing the Wii-Mote. As source files. Feel free to change.

Wii-Mote Bluetooth Stack Demo (wiimoteDemo): A demo program for testing the hardware. Just edit the mac address and compile.

Bluetooth Stack UART Test (wiimoteUartTest): A test program for testing your UART implementation. If everything works it should behave like the Stack Demo. Note that not every flow control scenario will happen in this test.

Testbench for LFSR (librand_test): Testbench to check the implementation of the LFSR used by the random number generator.

3.2 Advice

- Use the provided libraries (GLCD, Bluetooth) to bootstrap the application development.
- Place constant strings in the Flash instead of the RAM (this is required!).
- Keep the ISRs as short as possible.
- Use background tasks. But keep in mind that monopolizing background tasks are forbidden!
- Employing a framebuffer might reduce the time spent for screen updates. This is most likely only needed if you plan to use elaborate animations, otherwise it is a good exercise.

4 Detailed Specification

A proposal for the software stack of the game can be found in Figure 3. The modules filled green are provided by us. The white modules have to be implemented by you and are explained in depth in the following sections.

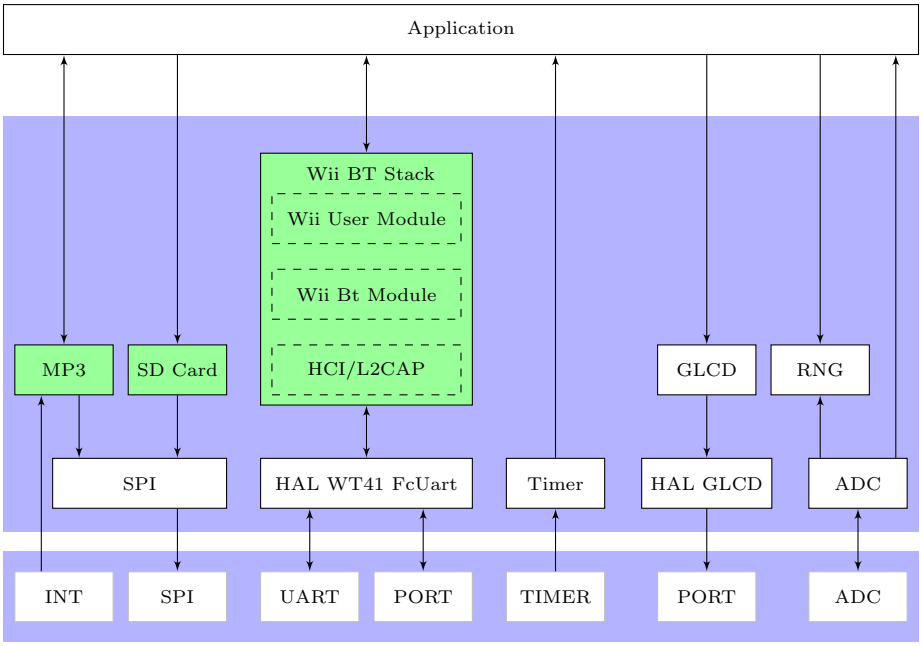

Figure 3: Software stack proposal. The arrows show the control flow between the respective modules.

4.1 Falling down ball

Your task is to build a game in which the movement of the ball must be controllable with the Wii-Mote. The visible playing field of the game is 128×64 pixels, i.e., fills the whole screen.

After the game has started, the ball should be positioned at the edge of the screen that you choose as bottom, in the center of the screen. The game ends when the ball touches the opposite edge, i.e., the top end of the screen.

The playing field has to include platforms, parallel to the top/bottom edge that scroll in from the bottom and move to the top. Furthermore, there has to be at least one gap in every platform such that the ball can drop from one platform to the next. Hence, you have to implement some sort of gravity that allows the ball to move in the direction of the bottom edge without any input if there is no platform between the ball and the bottom edge. We recommend that the scrolling, i.e, movement of the platform from bottom to top happens along the y-axis on a per pixel basis, as scrolling on the x-axis is rather complicated.

Based on this specification of the playing field you have to generate an infinite randomized level. The level should get increasingly harder, e.g., y-scrolling gets faster over time. You can also consider including other challenges like obstacles on the platforms as additional difficulty.

The movement speed of the ball has to dependent on the values you receive from the accelerometer! For a smooth gameplay we recommend game screen updates (ticks) with a frequency of at least 20 Hz. Furthermore, you have to ensure that the amount of pixels the ball can travel per tick is such that it is humanly traceable but also fast enough to prevent instant loss when the level difficulty increases.

The game should have background music. In the lab we will provide you with an SD Cards with tracks “suitable” for this purpose, which should be played via the MP3 decoder board. The volume of the playback must be controllable via the potentiometer, see Section 4.8.1 for details.

After the game has ended, the player must be given the possibility to play another round without performing a hardware reset. Furthermore, an entry in a high score table has to be made. The high score screen should be accessible between games. This screen has to hold at least five entries as long as the board is not reset or powered down (no persistent implementation is needed).

Attention

Be aware that if you implement and test the different modules separately that plugging them together could influences performance of some modules. Especially the USART used for the Wii module has a very low priority interrupt.

4.2 SPI

The SPI module implements a driver for the hardware SPI module of the ATmega1280. Use Mode 0 (CPOL=0, CPHA=0) and send the MSB first.

The module has to provide the following functions:

`void spiInit( );`

Initializes the hardware SPI module of the microcontroller.

`void spiSend(uint8_t data);`

Sends one byte via the SPI module. As the SPI data rate is very high, the overhead for setting up an interrupt is often larger than the actual time required to send the data. Therefore, you are allowed to use busy waiting to wait for a transfer to complete.

`uint8_t spiReceive(void);`

Receives one byte via the SPI module. As with the send function you may use busy waiting here. Note that the SD Card needs the pattern `0xFF` to be sent as dummy value during read.

`void spiSetPrescaler(spi_prescaler_t prescaler);`

Sets the prescaler of the SPI module to a certain value. The SD Card library uses an `enum` to change the prescaler. You have to adhere to the following definition:

```
typedef enum {
        SPI_PRESCALER_128       = 3,
        SPI_PRESCALER_4         = 0,
} spi_prescaler_t;
```

4.3 SD Card

The (Read) Access to the SD Card is done by the SD Card module, which is provided as a precompiled library[2] to you. The access is done in data blocks with a length of 32 bytes. While the library reads the card blockwise, it expects the read offset as block-aligned *byte address*. The library handles the chip-select signals for the SPI access and verifies the presence of a card before trying to access it. Please note that CRC checksums are not verified by the library.

For the SPI access the library uses the functions *spiSend*, *spiReceive*, and *spiSetPrescaler* from the SPI module, details can be found in Section 4.2. Note that the SPI module has to be initialized before using any function of the library. During the initialization of the SD Card module, the SD Card is also initialized. This happens at a lower lower speed than the actual read access. Be sure that *spiSetPrescaler* works correctly, if there are problems with the SD Card access.

Attention

We have tested the library against the SD Cards we use in the lab. If you use other SD Cards we cannot guarantee anything about the functionality. Additionally, please do not use SDHC cards as they are known to be unsupported.

If you make solid modifications to the library, e.g., making it compatible to other SD Cards, we would appreciate early feedback. We would then review the changes and make them available

[2] The source is also available on the course website.

- for other students.

The library provides the following functions:

```
error_t sdcardInit( );
```

The function initializes the SD Card module and the SD Card. The return value is of type *error_t*, listed below, which gives information about the result of the operation.

```
error_t sdcardReadBlock(
     uint32_t byteAddress,
     sdcard_block_t buffer
    );
```

This function reads one 32 byte block from the SD Card. The parameter *byteAddress* specifies the start address of the block to be read and *buffer* is a pointer to a buffer of at least 32 bytes. The return value is of type *error_t* which gives information about the result of the operation. The provided structure and datatype are shown below.

```
typedef enum {
    SUCCESS,
    ERROR,
    E_TIMEOUT,
} error_t;

typedef uint8_t sdcard_block_t[32];
```

4.4 MP3

Like the SD Card module, the MP3 module is provided as a library.

The MP3 module handles the configuration and communication of the MP3 decoder chip. In fact the MP3 decoder chip is a small processor that has some registers for configuration and some memory for buffering the stream. It has two interfaces operated by different chip select signals. The command interface is used to read and write register values and the data interface is used to send MP3 data. Commands and data are sent over SPI using the *spiSend* and *spiReceive* functions implemented in your SPI module.

The library provides the following functions:

```
void mp3Init(void (*dataRequestCallback)(void));
```

This function initializes the hardware and sets the access mode of the decoder chip. It also sets up the external interrupt for the data request line of the decoder chip. The callback function is called whenever the decoder chip requests more data.

Please note that the callback function is called in the context of an interrupt. Therefore, it should be as short as possible.

```
void mp3SetVolume(uint8_t vol);
```

The *SetVolume* function sets the output volume of the decoder chip, where a higher value corresponds to a higher volume. On behalf of the other lab users, we ask you kindly to make use of the function thoughtfully.

```
void mp3StartSineTest();
```

With the *StartSineTest* function you can test your SPI implementation. If communication with the decoder chip works, you should hear a tone after calling the function.

```
void mp3SendMusic(uint8_t *buffer);
```

The *SendMusic* function sends 32 bytes of music data to the decoder chip.

```
bool mp3Busy(void);
```

This functions checks if the decoder chip is ready to receive new/additional data. If `false` is returned, you can directly send the next 32 bytes for decoding via `mp3SendMusic(...)`. Otherwise you should wait for *dataRequestCallback* to be called.

Hint

There are a lot of ways to implement the MP3 playback. We do not want to restrict you to a special way of doing it, but we want to sketch one possible way: Use a background task to copy data from the SD Card to the MP3 chip. Do this until *mp3Busy* tells you to stop.[3] If *dataRequestCallback* is called, set a flag to tell the background task that new data can be copied. Either the background tasks are still executing, or the background tasks will be executed after the return of the ISR. The latter case follows from the fact that the callback is called during an interrupt and thus has woken up the MC.

4.5 Wii Bluetooth Stack

The Wii Bluetooth stack is also provided as a precompiled library, and as source code. It handles everything related to the communication with the Wii-Mote. This includes the implementation of the Bluetooth L2CAP layer, and on top of that the HCI implementation, for communicating with the Bluetooth chip on the Wii-Mote. The library depends on your WT41 HAL module implementation for accessing the WT41 Bluetooth chip.

The library supports up to 7 Wii-Motes. The number of supported motes can be changed by changing the *WII* preprocessor constant either directly in the stack or in the makefile. The default number of supported Wii-Motes is set to 4. To save resources you should change this to 1 if you use the source version of the library.

4.5.1 Wii User Module

The Wii user module provides a high-level abstraction for the Wii-Motes to allow for comfortable application programming.

The following functions are provided:

```
error_t wiiUserInit (
     void (*rcvButton)(uint8_t wii, uint16_t buttonStates),
     void (*rcvAccel)(uint8_t wii,
                      uint16_t x, uint16_t y, uint16_t z)
    );
```

This function initializes the stack and has two function pointers as parameter.

The callback *rcvButton* is called whenever a Wii-Mote sends a datagram containing information about the states of the buttons. The callback's parameters are as follows: `wii` indicated the index of the Wii-Mote that sent the datagram. `buttonStates` holds the actual state for every button on the Wii-Mote.[4]

The callback *rcvAccel* is called whenever a datagram containing accelerometer data is received from a Wii-Mote: The callback's parameters are as follows: `wii` indicated the index of the Wii-Mote that sent the datagram. `x`, `y`, and `z` are the values of the accelerometer.[5]

If the initialization was done successfully, the function returns *SUCCESS*, otherwise it returns *ERROR*. The return values are defined by the following enum:

[3] If you have other background tasks you must pause copying after several blocks to prevent the task of monopolizing the MCU so that the other background tasks can also be executed.

[4] The semantics of the value is listed at http://www.wiibrew.org/wiki/Wiimote#Buttons.

[5] We refer you to http://www.wiibrew.org/wiki/Wiimote#Accelerometer for a description of the values.

```
typedef enum {
    SUCCESS,
    ERROR,
    E_TIMEOUT,
} error_t;
```

```
error_t wiiUserConnect (
      uint8_t wii,
      const uint8_t *mac,
      void (*conCallback)(uint8_t wii, connection_status_t status)
    );
```

The *UserConnect* function tries to establish a connection to the Wii-Mote with the given mac address and binds it on the index *wii*. If provided, the module calls the callback *conCallback* after every change of the connection state, e.g., after the connection was successfully established or after a disconnect. Parameters provided to the callback are the corresponding Wii-Mote index and the new status of the connection.

Attention

You should not call *UserConnect* again before the callback function has been called by the library.

Note

Please be aware that the *UserConnect* function tries to connect to a specific Wii-Mote and uses a timeout of **several seconds** to wait for a response. You may want to inform the player about the ongoing connection attempt and that he should press the sync button on the back of the Wii-Mote. Using a timer, you could also do nice animations on the display.

Note

UserSetAccel, *UserSetLeds*, *UserSetRumbler* return *ERROR* if the Bluetooth stack or the corresponding Wii-Mote were not ready for the command, otherwise they return *SUCCESS*.

```
error_t wiiUserSetAccel (
     uint8_t wii,
     uint8_t enable,
     void (*setAccelCallback)(uint8_t wii, error_t status)
    );
```

This function enables or disables the accelerometer built into the Wii-Mote with the index `wii`, depending on the value of the parameter `enable`. After the corresponding command is sent to the Wii-Mote, the callback *setAccelCallback* is called with the corresponding Wii-Mote index as well as a status parameter. Currently, the status parameter of the callback will always be *SUCCESS*, as the callback is only called if the command was sent successfully.

```
error_t wiiUserSetLeds (
     uint8_t wii,
     uint8_t bitmask,
     void (*setLedsCallback)(uint8_t wii, error_t status)
    );
```

This function sets the LEDs on the Wii-Mote with the index `wii` depending on the value of the parameter `bitmask`. Every bit of the lower nibble in the bitmask corresponds to a LED on the Wii-Mote. The LSB corresponds to LED 1. After the corresponding command is sent to the Wii-Mote, the callback *setLedsCallback* is called, with the corresponding Wii-Mote index as parameter, as well as the corresponding status. Currently, the status parameter of the callback will always be *SUCCESS*, as the callback is only called if the command was sent successfully.

```
error_t wiiUserSetRumbler (
     uint8_t wii,
     uint8_t enable,
     void (*setRumblerCallback)(uint8_t wii, error_t status)
    );
```

This function enables or disables the rumbler built in the Wii-Mote with the index `wii` depending on the value of the parameter `enable`. After the corresponding command is sent to the Wii-Mote, the callback *setRumblerCallback* is called, with the corresponding Wii-Mote index as parameter, as well as the corresponding status. Currently, the status parameter of the callback will always be *SUCCESS*, as the callback is only called if the command was sent successfully.

4.5.2 Wii Bluetooth Module

The Wii Bluetooth module handles the datagram communication with the Wii-Motes. It provides functions to directly communicate with the Wii-Motes and can be used to implement additional functionality to the stack, such as enabling extension modules or using other datagram modes. Note that it should not be necessary to use any of the functions directly as the user module should provide enough functionality.

4.5.3 HCI Module

The *HCI* module handles the low-level communication between the firmware of the *WT_41* Bluetooth module and the microcontroller as well as the logical links to the Wii-Motes via *L2CAP*.

4.6 WT41 HAL Module

The WT41 HAL module provides the hardware abstraction layer for the *WT_41* Bluetooth module. It is responsible for the initial reset, it sends characters to the module via UART and receives characters from the module via UART. The UART communication uses UART3 with a baudrate of 1 Mbit/s, 8 data bits, no parity, 1 stop bit (i.e., 8N1), and hardware flow control (RTS/CTS) in both directions. For the communication from the WT41 to the microcontroller, a ringbuffer must be implemented to increase the throughput. As stated on Section 1, you must not use busy waiting in this module.

The following functions have to be implemented:

```
error_t halWT41FcUartInit(
     void (*sndCallback)(),
     void (*rcvCallback)(uint8_t)
    );
```

This functions initializes UART3 as listed above, and prepares the ringbuffer of the receiving part. It also resets the Bluetooth module by pulling the reset pin *PJ5* low for 5 ms.

Whenever the module receives a character from the Bluetooth module it has to be put it into a ringbuffer. The buffer should be able to hold at least 32 bytes. For *every* character put in the ringbuffer, the *rcvCallback* callback function must be called. Re-enable the interrupts before calling the callback function. Make sure you do not call *rcvCallback* again before the previous call has returned.

If there are less than 5 bytes free in the buffer, the HAL module should trigger the flow control, by setting CTS to *high*, to indicate that it currently cannot handle anymore data. If the buffer gets at least half empty (less than 16 bytes stored in the case of a 32 byte buffer) the module should release flow control by setting CTS to *low*.

```
error_t halWT41FcUartSend(uint8_t byte);
```

This function should send the byte given as a parameter to the Bluetooth module. The corresponding *sndCallback* callback function will be called when the byte has been copied into the shift register of the UART, i.e., the byte is currently being sent to the Bluetooth module. You have to ensure that the callback is not called if there was no preceding send, e.g., after a reset.

When the *WT_41* Bluetooth module sets RTS to *high*, no more data must be sent to the module.[6] When the Bluetooth module clears RTS, transmission of data to the module can resume. While the check for the *low* RTS pin can be done right before a character is sent (copied into the UART data buffer), the recognition of the change on the RTS pin from *high* to *low* has to be done interrupt-driven using a *pin change interrupt*.

Attention

In case *halWT41FcUartSend* is called while the reset of the *WT_41* Bluetooth module is still performed, the data bytes has to be buffered. After the reset has concluded, the buffered bytes has to be sent and *sndCallback* called.

This approach must also be taken when the hardware flow control of the *WT_41* (RTS) is active.

The Bluetooth stack will send data byte-wise and continue with the next byte only when sndCallback is called, so no extra buffering needs to be done.

4.7 Graphical LCD

The graphical LCD stack contains two modules:

1. The HAL module abstracting access to the LCD controllers and thus providing consistent and transparent access to the complete LCD.
2. The GLCD module providing high-level functions to set, clear, and toggle individual pixels and to draw primitives such as lines, rectangles, circles, ...

The split in two modules is highly recommended by not necessary. Note that if you chose a different path which leads to duplicated code, there will be point deductions in accordance with the coding guidelines!

We are providing an example implementation as a precompiled library. This library is intended to allow you a rapid start with the application development.

The GLCD datasheet is available in TUWEL.

Note

You can choose to implement the graphical display or the standard 2×16 character LCD. If you choose the standard character LCD then only text writing (Section 4.7.2) has to be implemented. Note that in this case you can only achieve reduced points!

If you want to get the points assigned for the GLCD module you have to implement both sub-modules (drawing and text writing) by yourself.

4.7.1 GLCD

The GLCD module provides the high-level functions used by the application.

Attention

While implementing the GLCD driver, you will find out that the hardware does not always behave according to the datasheet. Unfortunately, this is not a rare trait of hardware (especially if it is very cheap).

[6] A transmission in progress can be finished, but no new transmission must be started.

It has to provide at least the following functions:

```
void glcdInit(void);
```

The function initializes the GLCD HAL layer and the GLCD itself. After the function is executed, the display should be cleared and the address should be set to the upper left corner.

```
void glcdSetPixel(const uint8_t x, const uint8_t y);
void glcdClearPixel(const uint8_t x, const uint8_t y);
void glcdInvertPixel(const uint8_t x, const uint8_t y);
```

These three pixel manipulation functions should set, clear and invert pixels. The coordinates should be such that $x = 0$, $y = 0$ is in the upper left corner and $x = 127$, $y = 63$ is in the lower right corner of the display.

Note

Care as to be taken with the drawing functions below: A pixel manipulation function may not be idempotent, as is the case with *glcdInvertPixel*. Thus the result may be incorrect if a pixel is drawn twice.

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

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

The *DrawLine* function takes two *xy_points* and draws a line in between them. How the line is drawn can be specified by the *drawPx* callback function which is called for every pixel that is on the line with the corresponding coordinates. That is, the callback should be one of the three pixel manipulation functions listed above.

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

The *DrawRect* function takes two *xy-points* and draws a rectangle using the points as opposite corners of the rectangle. The function has to work no matter which corners of the rectangle are provided. Again, the callback specifies how the pixels are drawn.

```
void glcdFillScreen(const uint8_t pattern);
```

The *FillScreen* function fills the whole GLCD screen with the provided pattern. The pattern should be filled in every page of the display, i.e., vertically over the whole display.

```
void glcdSetYShift(uint8_t yshift);
uint8_t glcdGetYShift();
```

The function *glcdSetYShift* sets the y-shift for the GLCD, allowing you to implement hardware supported vertical scrolling. The parameter `yshift` corresponds to the y-position in RAM that becomes the top line of the display. The function *glcdGetYShift* returns the current y-shift value, which you can buffer in memory instead of reading it back from the GLCD.

The following function can *optionally* be implemented:

```
void glcdDrawCircle(const xy_point c, const uint8_t radius,
                    void (*drawPx)(const uint8_t, const uint8_t))
void glcdDrawEllipse(const xy_point c, const uint8_t radiusX,
                     const uint8_t radiusY,
                     void (*drawPx)(const uint8_t, const uint8_t));
```

The *glcdDrawCircle* function draws a circle centered at point `c` with radius `radius`, using the specified pixel manipulation function. The *glcdDrawEllipse* functions draws an ellipse with the center at point `c` and a radius along the x- resp. y-axis of `radiusX` resp. `radiusY`. Please note that you might need to get a bit more creative how to draw circles using an 8-bit microcontroller rather than a normal PC.

```
void glcdDrawVertical(const uint8_t x,
                      void (*drawPx)(const uint8_t, const uint8_t));
void glcdDrawHorizontal(const uint8_t y,
                        void (*drawPx)(const uint8_t, const uint8_t));
```

These functions draw a straight line across the GLCD with a constant value for x resp. y coordinate.

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

This function extends *glcdDrawRect* by also drawing every point inside the rectangle with the specified draw function.

4.7.2 Writing Text

If you decided to implement the GLCD, additionally to the functions listed in the previous section, the following functions have to be implemented in the GLCD module. In case you only want to implement the 2×16 character LCD these are the only function you need to implement.

```
void glcdDrawChar(const char c, const xy_point p, const font* f,
                  void (*drawPx)(const uint8_t, const uint8_t));
```

The *DrawChar* function displays the character c at position p using font f by using the given pixel manipulation function. We are providing you a font description file on the course homepage.

```
void glcdDrawText(const char *text, const xy_point p, const font* f,
                  void (*drawPx)(const uint8_t, const uint8_t));
```

DrawText repeatedly calls DrawChar for all chars in the zero-terminated string *text*.

```
void glcdDrawTextPgm(PGM_P text, const xy_point p, const font* f
                     void (*drawPx)(const uint8_t, const uint8_t))
```

DrawTextPgm differs from *DrawText* in the face that the string *text* is expected to be in the program memory.

4.7.3 HAL GLCD

The *HAL GLCD* module implements the hardware abstraction layer for the GLCD module. It provides transparent access to the two display controllers of the display. Each of the two controllers is responsible for 64×64 pixels. The controllers only allow access to pages of 8 pixels each.

The following functions have to be implemented:

```
uint8_t halGlcdInit( void );
```

Initializes the microcontroller's interface to the GLCD, reset and initializes of the display controllers. After calling this function the GLCD should be empty and ready for use.

```
uint8_t halGlcdSetAddress(const uint8_t xCol,
                          const uint8_t yPage);
```

This function sets the internal RAM address to match the x and y addresses. While $0 \leqslant$ `xCol` $\leqslant 127$ is the horizontal coordinate from left to right, $0 \leqslant$ `yPage` $\leqslant 7$ denotes the 8-bit pages oriented vertically from top to bottom. Note that this convention differs from the chip datasheet where the orientation of `x` and `y` is different.

`uint8_t halGlcdWriteData(const uint8_t data);`

This function writes data to the display RAM of the GLCD controller at the currently set address. The post increment of the *writedisplaydata* operation, which is provided by the GLCD controller, should be transparently extended over both display controllers. That is, executing

```
halGlcdSetAddress(0x3F, 0x01);
halGlcdWriteData(0xAA);
halGlcdWriteData(0x55);
```

yields the following:

- As the `xCol` value is less than `0x40`, *Controller* 0 is selected and its RAM address is set to (`0x3F`, `0x01`).
- `0xAA` is written to the RAM of *Controller* 0, followed by an increment of the address.
- The module is aware of the current state (after the post-increment the address is `0x40` and thus beyond the "border between the controllers"), selects *Controller* 1, and sets its RAM address to (`0x00`, `0x01`).
- `0x55` is written to the RAM of *Controller* 1, followed by an increment of the address.
- After the execution of the second write, *Controller* 1 is active and the RAM address points to (`0x01`, `0x01`).

`uint8_t halGlcdReadData();`

This function reads data from the display RAM of the GLCD controller at the currently set address. The post increment of the *readdisplaydata* operation should be transparently extended over both display controllers. That is, executing

```
halGlcdSetAddress(0x3F, 0x03);
temp=halGlcdReadData();
temp2=halGlcdReadData();
```

yields the following:

- As the `xCol` value is less than `0x40`, *Controller* 0 is selected and its RAM address is set to (`0x3F`, `0x03`).
- The RAM of *Controller* 0 is read at position (`0x3F`, `0x03`), followed by an increment of the address.
- The module is aware of the current state (after the post increment the address is `0x40` and thus beyond the "border between the controllers"), selects *Controller* 1, and sets its RAM address to (`0x00`, `0x03`).
- The RAM of *Controller* 1 is read at position (`0x00`, `0x03`). The address is automatically incremented.
- After the execution of the second read, *Controller* 1 is active and the RAM address points to (`0x01`, `0x03`).

Some Hints for the Implementation of the Module

As interfacing with a complex module, like the GLCD, is error-prone and thus can become very cumbersome, we give you some hints about the inner structure and how to get started.

Respect the Timing! There are timing diagrams for read access and write access in the datasheet of the GLCD [1, pages 11,12]. In particular, take care of the 140 ns setup time before the rising edge of the *enable* signal and take care of the $\geqslant 320$ ns data delay time when reading data. These delays are only a few *nop* operations, so you can use inline assembler to get accurate timings, e.g., `asm("nop");`.

Start Simple! Start with a very minimalistic version of the module, only able to handle one controller and without any special features. It should only be able to write bytes on the display. Once this works, you can extend the functionality.

Reduce Complexity! Building complex functions out of simpler ones helps generating clean debuggable code. Among others, we recommend the following "private" functions to help implement the module's functionality:

```
void halGlcdCtrlWriteData(const uint8_t controller,
                          const uint8_t data);
uint8_t halGlcdCtrlReadData(const uint8_t controller)
void halGlcdCtrlWriteCmd(const uint8_t controller,
                         const uint8_t data);
void halGlcdCtrlSetAddress(const uint8_t controller,
                           const uint8_t x,
                           const uint8_t y);
void halGlcdCtrlBusyWait(const uint8_t controller);
void halGlcdCtrlSelect(const uint8_t controller);
```

4.8 ADC

The ADC module of the microcontroller is used to perform two measurements, on different channels, for the application.

1. The ADC is used to convert the analog voltage of the potentiometer P5 to a digital value which is used in the volume control.
2. Another channel of the ADC is set into a high-gain mode to generate noise, which is used to seed the PRNG.

The ADC driver has to be completely interrupt driven.

Note

Keep in mind that care must be taken when switching channels, as stated in [2, Sec. 26.5.1].

4.8.1 Volume Adjust

As is generally the case, the volume of the music playback should be changeable. Increasing the volume should be achieved by turning the potentiometer clockwise. Use ADC Channel 0 as input and try to approximate a linear loudness scale.[7] The function $1-(1-x)^4, x \in [0,1]$ is a good approximation and can be implemented using 16-bit integer arithmetic (scale wisely). You may also use an alternative function to approximate a linear loudness scale. If you do so, you have to provide a justification in the protocol why your function fulfills the above requirement.[8]

4.8.2 PRNG

The random number generator and its implementation are described in the Rand howto [3]. The interface you should implement is described in [3, Sec. 4.1]. Please note that the *rand_shift* function should be implemented using inline assembler.

The functionality of the PRNG is basically as follows: Whenever the application needs a random bit, it calls the high-level function `rand1()` (or `rand16()` if a random 16 bit word is needed). When the ADC finishes a conversion that is used as an entropy source, the high-level function `rand_feed()` is called. All high-level functions should make use of the `rand_shift()` function.

[7] The term linear might be misleading without context: The volume setting of the MP3 module has an linear effect on the amplitude of the sound being played. However, the human hearing is non-linear, and can in fact be considered logarithmic. Thus, the linear loudness scale is actually a logarithmic scale.

[8] A plot will suffice in most cases.

5 Demonstration and Protocol

To be awarded points for your application, you have to submit a `*.tar.gz` archive to TUWEL until 25.11.2018, 23:59. The name of the top-level folder of the archive must be your matriculation number. Inside the top-level folder, place the Application folder from the template and the Protocol folder with the Listing.pdf. You can use the Makefile of the template to generate an appropriate archive. See below for a link to the template which can also generat archives following this convention.

You are also obliged to participate in a delivery talk. The delivery talks will be held between the 26.11.2018 and the 29.11.2018. You must register in TUWEL for a 30 min slot. Please do so early, there are enough slots for everybody (and we will open additional ones if necessary) but we cannot accommodate everybody on the same day!

During the delivery talk, the tutor will download your application from TUWEL, check its functionality on the lab hardware, and will walk through your solution with you. Be prepared that the tutor asks you questions about your implementation. Only submissions that were approved by the tutor are considered by us. We also require you to print out the first page of your Protocol (the adapted version from the template is sufficient), sign it at the two indicated spots, and give it to the tutor at the talk.

The lab Protocol itself should be written in LaTeX, explaining your implementation and the decisions you made during implementation. It should also contain a break-down of your working hours needed for the application and list which tasks required how much time (e.g., reading manuals, implementation, debugging, writing the protocol, ...). A template for the layout of the code/protocol separation, including a LaTeX template, is available under Application 1 additional material in TUWEL. The protocol, including the solutions for the theory tasks if applicable, has to be submitted as a `*.tar.gz` archive to TUWEL until 2.12.2018, 23:59. The name of the top-level folder of the archive must be your matriculation number. Inside the top-level folder place the Protocol folder from the template. in addition to the LaTeX files in the Protocol folder, include the Protocol.pdf in the Protocol folder! The Makefile of the template provided by us has a target to generate an appropriate archive.

5.1 Checklist

- Finish writing Application
- Check that the `listings.tex` in the Protocol folder includes all source files
- Generate a code archive via the provided Makefile target `code`
- Upload the archive to TUWEL
- Download the submission from TUWEL and
  1. verify that the submission compiles in the lab and
  2. works on the target as intended.
- Finish writing the protocol and preparing for the second exam. Furthermore participate in a delivery talk.
- Generate a lab protocol archive via the provided Makefile target `protocol`
- Upload the archive to TUWEL
- Download submission from TUWEL and check that all required files are included.

6 Grading

Please note that the points below are upper bounds that are possible to reach. However, there is always the possibility of point reduction due to exhaustive use of resources, not using sleep mode, not fulfilling specification, ...

Also, remember the "Collaboration Policy" at the course web page which states 15 points deduction for cheating for all involved. Discussion among students is welcome, but this is no group task. All programming and theory tasks have to be done on your own!

Points	Sub	Part
25		Solution of App1
	7	Game application
	5	Bluetooth/UART
	4	Music Playback (MP3 and Volume)
	5	GLCD
	4	Rand, ADC

References

[1] Winstar Display Co., LTD, *WDG0151-TMI-V#N00 – Specification*. [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/glcd/glcd_128x64_spec.pdf

[2] Atmel, *ATmega640/1280/1281/2560/2561*, rev. 2549N-05/11. [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/atmega1280/atmega1280-manual/view

[3] P. Fimml, "HOWTO: A Simple Random Number Generator for the ATmega1280 Microcontroller." [Online]. Available: http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/misc/task1-specific-stuff/rand_howto.pdf/view