MCVU Application 1
Pong$_{v1.1}$

Version	Date	Remark
1.1	Apr 17	Added clearifications (marked) and theory tasks
1.0	Mar 26	Initial release

General Remarks

In the whole application you must not use busy waiting except for the LC–Display busy flag polling![1]

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 must 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)? What happens if the connection to a Wii-mote is lost? ...

1 High-Level Specification

You have to build a 2-player game called "Pong" similar to the original game published by Atari in 1972. The game is controlled by two Wii-Motes connected to the microcontroller via a bluetooth extension board. You have to ensure that the mapping between Wii-mote and player is clear from the leds on the Wii-motes.

The playing field of the game is the 128x64 pixel GLCD with the short edge as scoring area.

When a player scores a point, a sound from a SD Card should be played via the MP3 decoder extension board. The volume of the playback should be controllable via a potentiometer.

A game lasts until one player has scored 5 points.

The current score of the game should be displayed either on the GLCD, or on the 2x16 character Display.

[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 50μs time to execute. You are also allowed to use busy flag polling during the initialization of the LCD.

Overview

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

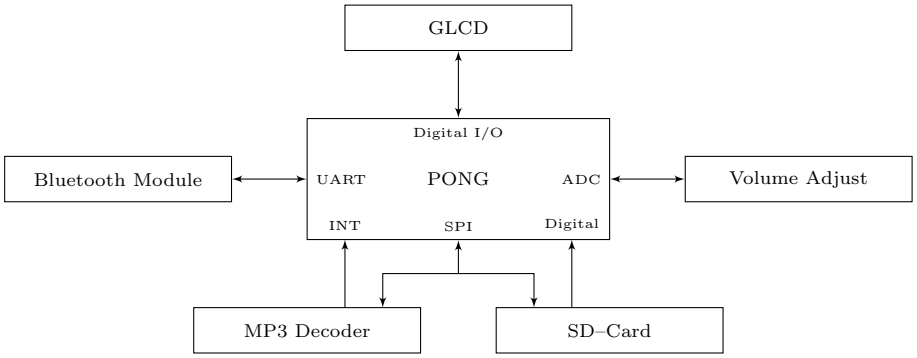

Figure 1: Pong – External Interfaces

Volume Adjust: The volume should be changeable by turning a potentiometer on the bigAVR6 board connected to the ADC. Ensure that by turning the potentiometer clockwise the volume should increase. Use ADC Channel 0 as input.

GLCD: The *GLCD interface* gives feedback to the user via an LC–Display. We do not want to overspecify the visuals nor the gameplay, as (1) they are not really part of the course and (2) we think that in this case the freedom gives you the opportunity to be creative and will come up with things we do not have thought of. Feel free to play around and change the specification of the game, e.g., let the platforms shoot and a hit lets a platform disappear for some time, allow to increase or decrease the speed by pressing keys on the Wii-mote, increase the speed of the game over time, ...

You can also think of using the internal EEPROM for saving a "highscore" list.

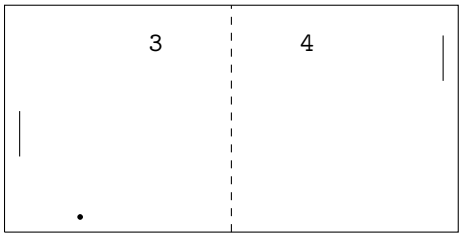

Figure 2: Sample Output on the LC–Display

Bluetooth Module: The bluetooth module uses an 1Mbit UART with hardware flow control for communication. Please ensure that the flow control

works, otherwise the module could crash the application. The module is connected to the bigAVR6 board by a ribbon cable attached to the expansion header on *PORTJ*.

For further information see Section 3.2.

SD Card: The SD Card stores the sounds you should use for playback after a point is scored. A detailed document with a list of sounds and their alignment on the card will follow. The sounds are written as raw data on the SD Card. 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 problem with the onboard slot is, that it drives the MISO line even if the chip-select is disabled and thus monopolizas the interface and destroys all other communication on the bus.

SD Card Block Access: The (read) access to the SD–Card is in data blocks with a length of 32 bytes. This part is provided as a precompiled library to you. The library handles the calculation of the real byte addresses out of the block addresses and the proper initialization of the SD Card.

SPI: The low level access to the card is done via SPI. Details can be found in Section 3.7.

Pinout: The SD Card slot on the SmartMP3 extension should be attached to the bigAVR6 board by 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
$\overline{\text{MMC\_CS}}$	PG1
$\overline{\text{MP3\_CS}}$	PB0
$\overline{\text{MP3\_RS}}$	PB4
SCK	PB1
MISO	PB3
MOSI	PB2
$\overline{\text{MMC\_CD}}$	PG2
DREQ	PD0
BSYNC	PB5

Table 1: Connection plan of the SmartMP3 extension

MP3 Decoder: We use a *VS1011e MP3 Audio Decoder* on the MikroElektronika SmartMP3 add-on board [1] which has a quite extensive data sheet [2]. We provide a precompiled library to you which handles the set-up and communication with the decoder. You just have to provide the data for it.

It shares the SPI module with the SD Card. The connection plan is shown in Table 1.

2 Implementation Remarks

2.1 Support Guide

To help you getting the application done, we have provided a variety of test programs ans libraries. All libraries can be downloaded from the course homepage at: `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/misc/task1-specific-stuff`

1. **GLCD Library (libglcd):** We provide you a GLCD library to enable fast start into application programming and easy debugging.
2. **Font Definition (font):** To make programming a text-mode on the GLCD easier for you we provide you a font specification.
3. **Wii-mote Bluetooth Stack (libwiimote):** The stack for accessing the Wii-mote. Packed and ready to go.
4. **Source of Wii-mote Bluetooth Stack (srcwiimote):** The stack for accessing the Wii-mote. As source files. Feel free to change.
5. **Wii-mote Bluetooth Stack Demo (wiimoteDemo):** A demo program for testing the hardware. Just edit the mac address and compile.
6. **Bluetooth Stack UART Test (wiimoteUartTest):** A test program for testing your UART implementation. If everything works it should behave like the Stack Demo.
7. **MP3 Decoder Library (libmp3):** The library for handling the MP3 decoder chip. Packed and ready to go.
8. **Source of MP3 Decoder Library (srcmp3):** The library for handling the MP3 decoder chip. As source files. Feel free to change.
9. **SD Card Library (libsdcard):** The library for accessing SD Cards. Packed and ready to go.
10. **Source of SD Card Library (srcsdcard):** The library for accessing SD Cards. Packed and ready to go. As source files. Feel free to change.
11. **SD Card Track Index (sdcardIndex):** A track list which file starts where and is how long on the SD Card that is provided in the lab. The addresses given are byte addresses and have to be transformed into block addresses for access through the SD Card library.

2.2 Advises

- Place constant strings in the Flash instead of the RAM.
- Keep the ISRs as short as possible.
- Use background tasks.

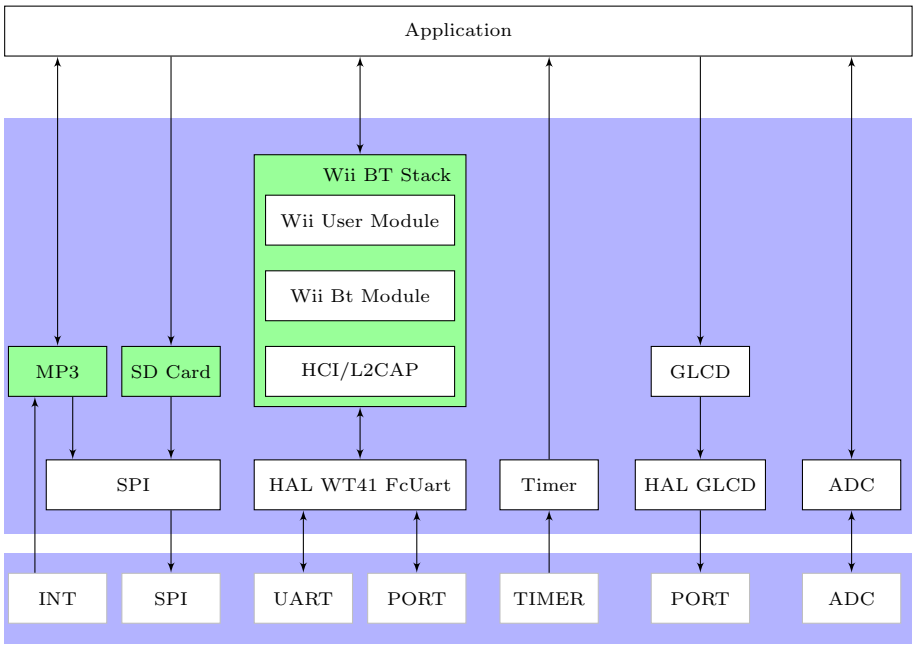

Figure 3: Software stack proposal (including control flow)

3 Detailed Specification

A proposal of the software stack of the pong game can be found in Figure 3. The modules filled green are provided by us. The modules are explained in the following:

3.1 Wii Bluetooth Stack

The Wii bluetooth stack is already implemented for you. It handles everything from the implementation of the bluetooth L2CAP layer on top of the HCI communicating with the bluetooth chip via your WT41 HAL module. It supports up to 7 Wii-motes. The number of supported motes can be changed by changing the *define WII* either directly in the stack, or in the makefile. To save resources the default number of supported Wii-motes is 4.

3.1.1 Wii User Module

The Wii user module provides the high-level abstraction for the Wii-motes to allow for a 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)
    )
```

The *UserInit* function takes two function pointers as parameter. *rcvButton* is the callback called from the module whenever a Wii-mote sends a datagram containing information about the states of the buttons. The callback has the following parameters: *wii* indicated the index of the Wii-mote that sent the datagramm. *buttonStates* holds the actual state for every button on the Wii-mote.[2] *rcvAccel* is the callback called from the module whenever a Wii-mote sends a datagram containing accelerometer data. The callback has the following parameters: *wii* indicated the index of the Wii-mote that sent the datagramm. x, y, and z are the values of the accelerometer.[3]

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

```
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 built up or after a disconnect. Parameters of the callback are the corresponding Wii-mote index and the new status of the connection.

The function returns *ERROR* if the bluetooth stack is not ready for a new connection, otherwise it returns *SUCCESS*.

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.*

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

The *UserSetAccel* function enables or disables the accelerometer built in in the Wii-mote with the index *wii* depending on the value of the parameter *enable*.

[2] Look at `http://www.wiibrew.org/wiki/Wiimote#Buttons` for the semantics of the value.

[3] Look at `http://www.wiibrew.org/wiki/Wiimote#Accelerometer` for the descriptions of the value.

After the corresponding command is sent to the Wii-mote, the callback *setAccelCallback* is called. The callback provides the corresponding Wii-mote index as parameter, as well as a status. In the current implementation the status parameter of the callback will always be *SUCCESS*, as the callback is only called if the command was sent successfully.

The function returns *ERROR* if the bluetooth stack or the corresponding Wii-mote is not ready for the command, otherwise it returns *SUCCESS*.

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

The *UserSetLeds* 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. The callback provides the corresponding Wii-mote index as parameter, as well as a status. In the current implementation the status parameter of the callback will always be *SUCCESS*, as the callback is only called if the command was sent successfully.

The function returns *ERROR* if the bluetooth stack or the corresponding Wii-mote is not ready for the command, otherwise it returns *SUCCESS*.

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

The *UserSetRumbler* 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. The callback provides the corresponding Wii-mote index as parameter, as well as a status. In the current implementation the status parameter of the callback will always be *SUCCESS*, as the callback is only called if the command was sent successfully.

The function returns *ERROR* if the bluetooth stack or the corresponding Wii-mote is not ready for the command, otherwise it returns *SUCCESS*.

3.1.2 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.

3.1.3 HCI Module

The *HCI* module handles the low-level communication between the firmware of the *WT_41* bluetooth module and the microcontroller and handles the logical links to the Wii-motes via *L2CAP*. The module uses the functions provided by your implementation of the *HAL_WT41_FC_UART* module and provides the needed callbacks.

3.2 WT41 HAL Module

The WT41 hal module provides the hardware abstraction layer for the *WT_41* bluetooth module. It is resposible for the initial reset, it sends characters to the module via UART and receives characters from the module via UART. The UART communication is done by UART3 using a baudrate of 1Mbit/sec and hardware flow control (RTS/CTS) in both directions. For the communication from the WT41 to the microcontroller, it should use a ringbuffer to increase the throughput.

Questions about the use of busy waiting in this module are answered in the first of the general remarks.

Therefore, it has to provide the following functions to the bluetooth stack:

```
error_t halWT41FcUartInit(
     void (*sndCallback)(),
     void (*rcvCallback)(uint8_t)
    )
error_t halWT41FcUartSend(uint8_t byte)
```

The *halWT41FcUartInit* initializes the UART3 for 1Mbit asynchronous communication, 8 data bits, no parity, 1 stop bit, and prepares the ringbuffer of the receiving part. It also resets the bluetooth module by pulling the reset pin *PJ5* low for *5ms* (use a timer for this, you have plenty of them).

Whenever the module receives a character from the bluetooth module it should put it in a ringbuffer. The buffer should be able to hold at least 32 bytes. For every character put in the ringbuffer, the *rcvCallback* callback function should be called (one after another). Re-enable the interrupts before calling the callback function.

If there are less than 5 bytes free in the buffer, the hal module should trigger the flow control by setting CTS *HIGH* indicating the bluetooth module it currently cannot handle any 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 the flow control by setting CTS *LOW*.

In the other direction, if the WT41 bluetooth module sets RTS to *HIGH*, no more data must be sent to the module.[4] If the bluetooth module clears RTS, transmission of data to the module can continue. 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 via a *pin change interrupt*.

The *halWT41FcUartSend* function should send the byte given as parameter to the bluetooth module. The corresponding *sndCallback* callback function

[4] A transmission in progress can be finished but no new transmission should be started.

should be called if the byte is copied into the shift register of the UART, i.e., the transmit buffer is empty and 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 (reset).

~~If the UART transmit buffer is empty the *sndCallback* callback function should be called to indicate that the module is ready for the next character. The first callback must only be called after the initial reset of the bluetooth module.~~

3.3 Graphical LCD

The graphical LCD stack contains two modules: (1) The HAL module abstracting over the two LCD drivers and providing a consistent and transparent access to the complete LCD. (2) The GLCD module providing the high-level functions to set, clear, and toggle individual pixels and to draw primitives such as lines, rectangles, circles, ...

We are providing an example implementation as a precompiled library. This library is intended to allow for a rapid start.

The GLCD data sheet is available on the course homepage [4].

Note *You can either choose to implement a text mode on the graphical display to display text such as the score, or else you can implement the standard character LCD.*

If you want to get the points assigned to the GLCD module you have to implement both sub-modules by yourself. If you do not implement a text mode, you have to additionally implement the character LCD to get the points.

3.4 GLCD

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

The 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.

```
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 this is done can be specified by the *drawPx* callback function which is executed 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.

```
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 glcdDrawCircle(const xy_point c, const uint8_t radius,
                    void (*drawPx)(const uint8_t, const uint8_t))
```

The *DrawCircle* function draws a circle centered in *c* and with radius *radius* with the specified pixel manipulation function. Please note that you might need to get a bit more creative how to draw circles using an 8-bit microcontroler rather than a normal PC.

```
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.

Writing Text

If you decide to write text on the GLCD, additionally the following functions have to be implemented in the GLCD module. Otherwise, you can use the $2x16$ character LCD to display text as it can display characters directly.

```
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 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*.

3.5 HAL GLCD

The *HAL GLCD* module implements the hardware abstraction layer for the GLCD module. It provides transparent access to the two display controllers by the following functions:

`uint8_t halGlcdInit( void )`

Initializes the microcontroller to interface the GLCD and initializes the display controllers. After this function the GLCD should be empty and ready for use.

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

This functions 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 data sheet as in the data sheet the orientation of `x` and `y` is a different one.

`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, the address is automatically incremented.
- The module is aware of the current state (after the post increment the address is $0x40$ and thus beyond the "border between the controllers") and selects *Controller* 1 and sets its RAM address to $(0x00, 0x01)$.
- $0x55$ is written to the RAM of *Controller* 1, the address is automatically incremented.
- 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 $(0x3F, 0x03)$. The address is automatically incremented.
- 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 quite complex module like the GLCD is very error-prone and thus can become very cumbersome, we give you some hints about the inner structure and how to get starting.

- **Respect the Timing!** There are timing diagrams for read access and write access in the datasheet of the GLCD [4, pages 11,12]. Especially you should take care of the $140ns$ setup time before the rising edge of the *enable* signal and when reading data you should take care of the $\geqslant 320ns$ data delay time. This 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. If this works, you can extend the functionality.
- **Reduce Complexity!** Building complex functions out of easier ones helps generating clean debuggable code. Among others, we recommend the following "private" functions helping to implement the modules 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)
```

3.6 SD Card

The SD Card module is provided as a pre-compiled library [5] to you. It handles the block access to a SD Card. The library handles the chip-select signals of the SPI access and before it accesses the card, it checks if it is really inserted. Please note, that the CRC of the block read is not checked by the library. In fact, the library only takes a block address and returns the data stored on the SD Card. Therefore, it uses the SPI module which implements SPI byte read and SPI byte send routines. Note that the functions *spiSend(uint8_t c)* and *spiReceive()* used by the module have to be implemented in the SPI module and that the SPI module has to be initialized somewhere else before using any function of the SD Card module.

Note *We have tested the library well against the SD Cards we use in the lab. If you use other SD Cards we cannot say anything about the function. Additionally, please ensure not to use SDHC cards as they are known to be not supported. If you make solid modifications to the library, making it more compatible to other SD Cards, we would appreciate if you give us early feedback. We would than review the changes and make them available for the other students too.*

The library provides the following functions:

```
error_t sdcardInit( )
```

The *init* function initializes the SD Card module and the SD Card.

```
error_t sdcardReadBlock(
     uint32_t blockAddress,
     sdcard_block_t buffer
    )
```

The *readBlock* function reads one 32 byte block from the SD Card. The parameter *blockAddress* specifies the block address and *buffer* is a pointer to a buffer of at least 32 bytes. Both functions return a value of type *error_t* which gives information about the result of the operation. The provided structures are shown below.

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

typedef uint8_t sdcard_block_t[32];
```

3.7 SPI

The SPI module implements a driver for the hardware SPI module of the ATmega1280. Clock Polarity and Clock Phase of the SPI settings can remain the initial values. The SPI clock rate should be set to $f_{OSC}/4$. Since at this high

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

SPI data rate, compared to the transmission time, the interrupt overhead is very high and often larger than the actual work to do, you are allowed to use busy waiting to detect a completed transfer in this module. Please note, that for both – the SD–Card and the MP3 Decoder – half duplex communication is sufficient and that sending starts with the MSB.

3.8 MP3

Like the SD Card module, the MP3 module is provided to you as a pre-compiled library (also as source code).

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

The library provides the following functions:

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

The *Init* function initializes the hardware and sets the access mode of the decoder chip. It also sets up the external interrupt for data request of the decoder chip. The parameter is the callback function that should be called when a data request of the decoder chip is issued. Please note, that the callback function is called in the interrupt context. Therefore, it should be as short as possible.

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

The *SetVolume* function does exactly what its name states. It sets the output volume of the decoder chip. Please, on behalf of the other lab users we ask you to make use of the function thoughtfully. The higher the value, the louder the sound.

```
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 32bytes of music data to the decoder chip. After sending data, you can use the function

```
bool mp3Busy(void)
```

to check if the decoder chip is still ready for new data. If so, you can directly send the next 32bytes for decoding. Otherwise you should wait for the dataRequest Callback.

There are a lot of ways to implement the playback of the MP3 files. We did not want to restrict you to a special way of doing it, but since there were questions about it 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.[6] If the *dataRequestCallback* is called, set a flag to tell the background task that new data can be copied. As the callback is called in an interrupt, either the background tasks are still executing, or the background tasks will be executed right after the return of the ISR. In both cases the copy task should start over.

3.9 ADC

The ADC is used to convert the analog voltage output of the potentiometer to a digital value. The ADC part has to be completely interrupt driven.

4 Theory Tasks

Please work out the theory tasks very carefully, as there are very limited points to gain!

1. **[2 Points] Buffersize and flow control:** Assume the UART Setup as in the Application setup, that is, $BAUD$ bit/sec, 8N1. Furthermore, assume the ring buffer we have implemented is empty at the beginning, $Buffersize$ bytes large, and for simplicity forget about all the hardware and software implementation details and assume the UART Module could directly write to the buffer. The flow control is triggered in the moment the last empty place in the buffer is written. An element is removed from the buffer in the moment the corresponding callback is called.

   (a) As in the application, the receive callback function is issued subsequently for every received character. Assume the bluetooth module sends datagramms of length $DG_{length} > Buffersize$. Derive a formula for the upper bound on the execution duration of the callback function t_{CB} such that if one datagram is sent, the flow control is **not** triggered. The formula should depend on $BAUD$, $Buffersize$, and DG_{length}. Note that a UART-Mode of 8N1 is specified! Please be very careful when building the formula not to make an off-by-one mistake. Additionally calculate the upper bound for the following values: $BAUD = 10^6 bit/sec$, $Buffersize = 49 Byte$, and $DG_{length} = 64 Byte$.

   (b) Assume the previously mentioned datagramms come periodically every P_{DG} seconds. Derive a formula for the upper bound on the execution duration of the callback function t'_{CB} such that no matter how long the system runs the flow control will **never** be triggered. Additionally, calculate the upper bound using the values from before and assume $P_{DG} = 2.5 * 10^{-3} s$

2. **[1 Point] Pong vs. Billiard warmup:** In the following assume for simplicity that the playground of Pong is a rectangle of height 1 and width 2 and that both players are perfect, that is, they always hit the

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

ball with the tennis racket. Also assume, that there are no pixels, that is, we are playing in a continuous environment. Thus, this scenario is equivalent to a billiard board of height 1 and width 2 with one ball and no holes. The lower left corned has Cartesian coordinates $(0, 0)$, the upper left corner $(0, 1)$ and the lower right corner $(2, 0)$. Further assume (as in billiard) that the incidence angle is equal to the emergent angle whenever the ball hits a boundary or the players' tennis rackets. In case the ball hits a corner, it thus simply reverts its direction at the corner. The speed of the ball is constant during the whole game, i.e., we assume zero friction.

A game is called *periodic* if the ball repeatedly runs along the same finite track over and over again.

For each natural number n give an initial position and initial direction of the ball such that it hits the border at least n times before it reaches the same initial position with the same initial direction again (the game gets periodic). Prove that your solution is a majorizing series.

For example, initial position $(0, 1/2)$ and initial direction $(1, 0)$ yields that the ball will be at the initial position $(0, 1/2)$ with initial direction $(1, 0)$ after it hit the (opposite) boundaries twice.

3. **[2 Points] Pong vs. Billiard aperiodic gaming:** *Give an initial position and initial direction of the ball such that the ball will never reach the initial position again. Formally prove that your solution is correct.*

   Hint: Draw a (potentially endless) row of billiard tables at a piece of paper and assume that the ball can pass through the side boundaries of the tables such that it runs though all the tables. For example consider the initial position to be $(0, 0)$ and the initial direction to be e.g. $(1.5, 1)$. Observe what happens.

5 Demonstration and Protocol

If you want points for your application, you have to submit a **.tar.gz* archive to myTI until 10.05.2012, 23:59. Before you have to demonstrate *your* program to *your* tutor! Be prepared that the tutor asks you questions about your implementation you have to answer. Only submissions that were approved by the tutor are counted. You also have to write a LaTeX protocol explaining your implementation and the decisions you have made during implementation. It should also contain a break down of your working hours needed for the application and for which tasks you spent how much time (e.g., reading manuals, implementation, debugging, writing the protocol, ...). A LaTeX template ~~will~~ is be provided by us available at `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/misc/task1-specific-stuff/Application_Protocol_Template.tar.gz/view`.

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, ...

Therefore, if you aim for 12 points you should not gamble on dropping all theory tasks. Also remember the "Collaboration Policy" at the course web page which states 15 points detention for cheating for all involved. Discussion among students is welcome, but this is no group task. All programming and theory tasks has to be done by your own!

Points	Sub	Part
12		Application
	4	Pong
	2	UART
	3	MP3
	3	GLCD
5		Theory Tasks (every task is evaluated separately).

References

[1] MikroElektronika. *SmartMP3 Additional Board Manual.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/smartmp3_manual_v101.pdf/view`

[2] VLSI Solutions. *VS1011e - MP3 Audio Decoder.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/vs1011e-mp3-audio-decoder/view`

[3] Winstar Display Co., LTD. *WH1602B-TMI-ET LCD Module – Specification.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/2x16-lcd/WH1602B-TMI-ET.pdf`

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