MCVU 2017 – Application 1
FANtastic v1.2

Version	Date	Remark
1.2	April 10	Rewording. Changes in Section 4.3 (ZigBee) without change markings!
1.1	April 03	Updates regarding ZigBee
1.0	March 28	Initial release

Contents

1 General Remarks

We highly recommend you to first read the whole specification before starting reading datasheets or even programming!

Throughout the whole application you must not use busy waiting, except for the 2x16-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 shall not use it. Please, note that the coding guidelines, as published on the course homepage, apply to this application!

No specification is complete! Design decisions have to be documented in the protocol, e.g., what happens if the ZigBee module does not boot-up? 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 mc-leitung@tilab.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.

If there is a problem with a target in the lab, e.g., broken touchscreen, please write an email to mc-leitung@tilab.tuwien.ac.at so that it can be fixed!

Be aware that some PDF-viewer can copy unicode characters from a PDF, which look in many fonts identical to their ASCII related character. Yet, the compiler through cryptic/nonsensical warnings, at best. The following characters are prone to be affected: -, _, and *.

2 High-Level Specification

This specification document describes a controller for a fan, whose target speed can be specified. The application must be developed for the bigAVR6 development kit using an ATmega1280. The fan's speed can be controlled locally, or via wireless connection from a remote location. Additionally, a microphone is used to detect a clog in the intake of the fan. If such a clog s detected, the control target chances from constant RPM to a reduction of the generated noise and vibration. The GLCD is used to display (i) the current fan speed, (ii) the locally set fan speed, (iii) the remotely set fan speed, and (iv) if a clog has been detected (noise reduction). Furthermore, is shall be possible to render the FFT-spectrum of the microphone's input on the GLCD.

2.1 Overview

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

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.

[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.

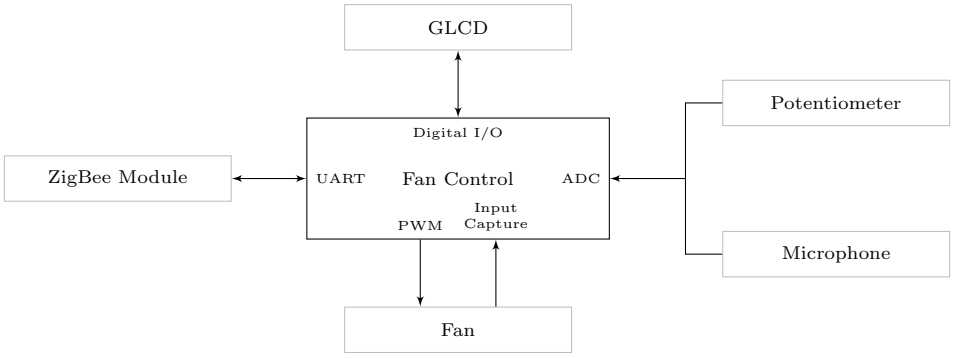
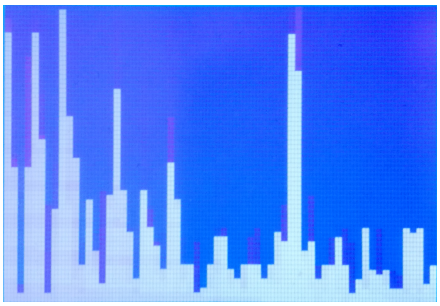


Figure 1: Fan Control – External Interfaces

(a) An example of the general overview.



(b) An example of the FFT screen.

Figure 2: Example output on the LC–Display. You are free to implement your own version.

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 locally control the speed of the fan.

2.1.3 Microphone

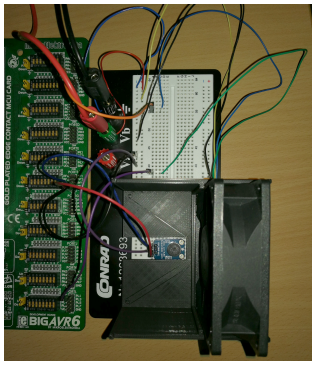
The microphone [1] should be connected to the ADC of the bigAVR6 board on Channel 7.

2.1.4 Fan

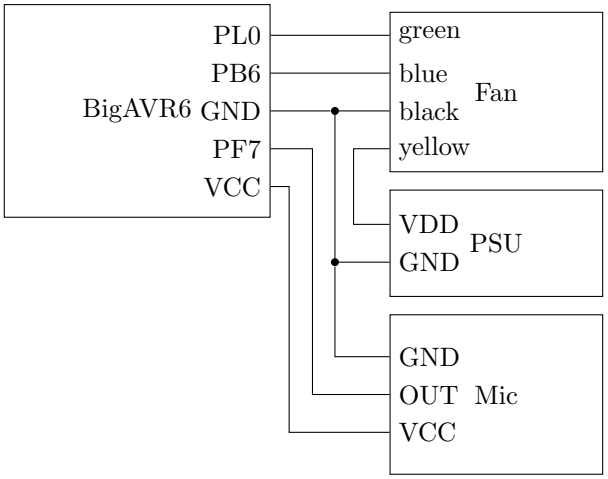
The fan [2] is the main components of this application. It is connected to the bigAVR6 board via a breadboard, as shown in Figure 3.

By connecting the blue wire directly to pin PB6, and generated a PWM signal on this pin, it is possible to control the speed of the fan. The tacho signal (green wire) should be connected to Pin PL0, so that the current RPM of the fan can be determined. The fan requires a properly sized resistor connected to VCC to generate a tacho signal. Luckily, the external pull-ups on the board have an appropriate size, thus enable the external pull-up for PL0 on SW11 and set J11 to VCC. At no time, the green wire must be in contact with the yellow or black wire!

The fan's operational voltage is above the 5 V the bigAVR6 board can provide. We thus will equip the lab with a power supply (PSU) which can generated the required 12 V. Connect the PSU with the test leads (Kabelstrippen) to the breadboard's lead connectors. These connectors have a hole (visible when turned open) where jumper wire (Schaltdraht) can be attached. Setup up the breadboard to have a common ground (black wire of the fan) between PSU and microcontroller board. The yellow wire of the fan must be connected to the plus-side of the PSU. The maximum voltage allowed for the fan is 13 V, be sure to setup the voltage before connecting the positive test lead to the PSU!



(a) Photograph of setup of the external hardware components.



(b) Schematic overview of the required connections

Figure 3: Overview of the connection setup between the microcontroller board, the fan, and the PSU.

Attention

Only make changes to wiring when their respective power supply (PSU or bigAVR6) are turned of! Incorrect handling of the wire connections can harm you, your colleague, and the equipment!

2.1.5 ZigBee

The ZigBee module (ATZB-24) is directly attached to the bigAVR6 board via the expansion header on *PORTJ*, with switches 3 and 7 enabled. It provides a serial interface over which data transmission/reception can be performed. The serial interface uses 38.4 kbps UART with hardware flow control.

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 from the course homepage at: `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/misc/task1-specific-stuff`

GLCD Library (libglcd): We provide you with a GLCD library. This library shall enable you to start with other parts of the application first, and will aide in debugging.

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

SerialNet library (libserialnet): We provide you with a library which implements the required ZigBee calls and the UART. This library is here to enable you to integrate the remote features before implementing the ZigBee module.

3.2 Advice

- Use the provided libraries (GLCD, ZigBee) 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 architecture of the fan control can be found in Figure 4. The white modules have to be implemented by you and are explained in depth in the following sections.

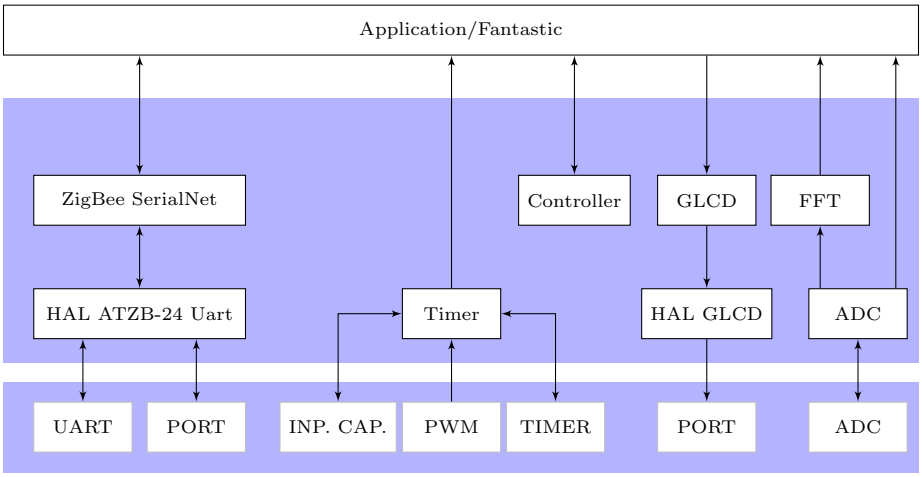


Figure 4: Software architecture proposal. The arrows show the control flow between the respective modules.

4.1 FANtastic

You have to build a controller, which uses a PWM signal to control the fan's speed. The current speed of the fan can be determined with the tacho signal provided by it. The target speed for the fan can be given via the potentiometer, or it can be received wirelessly via ZigBee. On reception of a RPM target over ZigBee, this target speed shall be used. Another ZigBee command switches back to local target speed control. You can also add a button to do this.

As clogged air intake to the fan causes unpleasing noises (it just gets louder), this should generally be avoided. Especially, as for arbitrary reasons[2], this also causes major issues with customer satisfaction. Therefore, a microphone has been located near the fan. Such a clog changes the recordable spectrum of the fan notably. By using a FFT on the output-signal of the microphone, the controller shall detect a clog in the intake, switch into noise-reduction mode, wirelessly signal for maintenance, and use a different control target.

It is not required to detect the removal of the clog online. Due to the low RPM in this mode, this would be rather challenging. Resetting the mode should be possible via a button, and via ZigBee.

Note that a switch between control targets should behave similar to a reset: the time related components should be set to 0.

The GLCD has a resolution of 128×64, and shall act as a local display. It shall display the current speed (in RPM), the set speed target, and which control target is selected. Furthermore, it shall be displayed if the noise-protection mode is active, and its respective parameters. Further, it should be possible to switch to a view of the FFT-spectrum with a button, as this will surely help you debug things.

[2]Some blind lawyer in Hell's Kitchen complained about it in a previous version... Has he an enhanced sense of hearing or what?

4.2 Controller

Attention

Testing the control loop will require you to artificially influence the fan. This can be achieved electronically, by lowering the voltage, setting a aggressive current limitation, or by mechanically slowing the fan.
Take care when doing so. Actions which endanger the health of your colleague and yourself, as well as intentional damaging the equipment (sticking a pen into the fans blades while running), will have consequences if the teaching staff is made aware of them!

4.2.1 RPM Controller

The specific type of controller to be used is up to you. We only require that it has the following properties:

- the controller is of a continuous type, i.e., no bang-bang control.
- the control loop is stable.
- the error between reference and fan RPM shall be zero[3] within a few seconds.

The controllers computation shall be executed with 16-bit arithmetic. As some computation can overflow, the controller has to be implemented in *inline assembler* with over/underflow checks for every computation.[4] We recommend to initially implement the controller with 32-bit arithmetic (`int32_t`), and once it works as intended, rewrite it to 16-bit arithmetic in assembler.

We recommend an execution frequency of 10 Hz for the controller. As the controller has to respond as periodic as possible you can either compute the control value in blocking mode, or use a second task to set the PWM duty-cycle. Note that the later version introduces additional dead-time!

Parameter Determination The method to determine the parameters for the control loop is left open. Nonetheless, the determined parameters have to work on multiple fans/boards, expect obviously broken/defective equipment.

Note

The time constant for the fan to increase speed is shorter than for deceleration! Keep this in mind when determining the parameters.

4.2.2 Noise Controller

The controller should be switched to this mode once a clog has been detected. To do this, find in the DC-free spectrum[5] the 8 largest components and sum them up. We will refer to this sum as x. As the size of x will vary with the current fan speed, the RPM have to be taken into account when detecting a restriction. One way could be the following formula:

$$\frac{x}{16} > 400 + 50 \cdot \frac{\text{RPM}}{1024}.$$

We found it to work quite good for RPM of 1500 upward.

Of course, you are free to use a different detection method based on the FFT, if it achieves similar results.

The control-target in this mode shall be that the sum of the 8 highest components of the DC-free spectrum is below 300. The specific controller-type is left open, although no overshooting is allowed! Furthermore, the control error may be larger than 0 in the infinite, i.e., you may leave an error of some size. Yet, the controller output must not produce a fixed (low) RPM value, i.e., no trivial solutions.

[3] As an error of zero will not be achievable due to environmental influences, we consider a slow "oscillating" behavior around zero as sufficient.

[4] If you can justify why a certain computation cannot overflow, the checks can be omitted. The justification has be given as a code comment or in the protocol! If done in the protocol, it has to be clear which line of code is referred to.

[5] Dropping the lowest 2 components should achieve this.

4.3 ZigBee

For the ZigBee communication we use the EasyBee board [3]. This board has an ATZB-24 chip[4] on it, which performs the physical access to the ZigBee network. The communication with the chips happens over UART with the so called SerialNet protocol. SerialNet is a development by Atmel [5] which is text based and uses AT-commands, which are commonly used in telecommunication.

This module requires you to implement the high-level FANtastic protocl, a handler for SerialNet protocol, and a HAL for the UART.

4.3.1 ATZB-24 HAL Module

The ATZB-24 HAL module provides the hardware abstraction layer for the UART communication with the ATZB-24 module. The UART communication uses UART3 with a baudrate of 38.4 kbps, 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 ATZB-24 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 halATZBFcUartInit(
     void (*sndCallback)(),
     void (*rcvCallback)(uint8_t)
    );
```

This function initializes UART3 as listed above, and prepares the ringbuffer of the receiving part.

Whenever the module receives a character from the ATZB-24 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 halATZBFcUartSend(uint8_t byte);
```

This function should send the byte given as a parameter to the ATZB-24 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 ATZB-24 module. You have to ensure that the callback is not called if there was no preceding send, e.g., after a reset.

When the ATZB-24 module sets RTS to *high*, no more data must be sent to the module.[6] When the ATZB-24 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 *halATZBFcUartSend* is called while the hardware flow control of the ATZB-24 (RTS) is active, the data byte has to be buffered. After the flow control has been reseted, the buffered byte has to be sent and *sndCallback* called.
The SerialNet 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.

Hint

Due to the properties of the SerialNet protocol, you might want to additionally implement a send function for strings. This function must adhere to the flow control with the ATZB-24, on a byte level!

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

4.3.2 SerialNet

The SerialNet module implements communication with the ATZB-24 on a protocol level, and requires an implemented UART HAL for its purpose. As mentioned earlier, the protocol is based on AT-commands thus simple strings have to be send, and parsed on reception.

On power-on the ATZB-24 will reset, which takes a notable amount of time. The completion of the boot process can be determined by sending `AT`. If the ATZB-24 is ready for reception, you will receive `OK`. After the module has been booted, it will respond to commands either with `OK` or `ERROR`.

After the module is ready, we recommend the sequence listed in Table 1 to setup the network connection.

On completion of the procedure, you will receive events when another node leaves or joins the network. These events, enable by `ATX`, will be transmitted in the format `EVENT:<text>`. Where `<text>` can be one of the following: join, lost, child_joined, child_lost, and calibr. All by join and lost can be ignored. Those two events signal that the connection/disconnection to/from the ZigBee network. If the network connection has been lost, transmission has to be halted until the connection has been restored.

Received data will be sent by the ATZB-24 in the format `DATA:<addr>,<bcast>,<length>:<data>`, where `<addr>` is the address of the sender, `<bcast>` is 1 if the message was send as a broadcast, 0 otherwise. The pair `<length>` and `<data>` are the payload. You can cut of any received data longer than 20 bytes, although null-terminated of all strings is required.

To send data over the ATZB-24, use the command `ATD <addr>,<arq>,<length>\r<data>\r`. Where `<arq>` set to '1' will require the recipient (on ZigBee protocol level) to acknowledge reception of the packet. Only after the acknowledgment has been received, or a timeout occurred, you will receive `OK`.

The central control station, placed by us in the lab, will be reachable under the address `0x00`.

Attention

The GSN corresponds to a MAC address and has to be unique in the lab. For this purpose, you should use your student ID.

Attention

To reset the ATZB-24 you have to power-cycle the microcontroller board. Pressing the reset-button on the bigAVR6 is not sufficient!

Hint

The SerialNet protocol offers a ping-like command. With `AT+WPING <addr>`, where `<addr>` is the short network address, you will receive `OK` if the node with the specified address replied, otherwise `ERROR`. This allows to check if node is connected, besides observing the event stream.

Note

While the SerialNet specification requires only a \r at the end of a command, we observed more stable behavior if also a \n is additionally given.

Note

In version 1.2 of this specification, we removed trailing `E` from `AT+WAUTONET=0` in Table 1. This will result in the ATZB-24 always echoing back received commands. The removal has happened as the results we saw were inconsistent.
We therefore recommend to either add the `E` after `ATZ`, or use the commands `ATE0` and `ATE1` to manually turn the echoing off resp. on. The echoing can be useful during debugging.

Remote Control Station The remote control station is provided by us as a compilation unit. Please ensure that only one such station is active in the lab at all times! A short help regrading the usage can be found when pressing PB7.

This allows you to set up a separate network (PAN ID and channel mask) while testing. To do this, add a call to `serialnet_set_base_network` in `all_io.c` line 213. This call needs to be made before calling `serialnet_init`, otherwise the changes have to be made manually with the appropriate AT commands.

Command	Purpose
`AT+WAUTONET=0\r\n`	Disable automatic networking during configuration
`ATZ\r\n`	Soft reset the module
`AT+IFC=2,2\r\n`	Enable flow control
`AT+GSN=XYZ\r\n`	Set MAC address (64 bit, hex) for the node
`AT+WPANID=1620\r\n`	Set node's PAN ID (16 bit, hex) identification of network)
`AT+WCHMASK=100000\r\n`	Set node's channel mask (2.4GHz channel)
`AT+WROLE=1 +WSRC=`*MTNR*`\r\n`	Switch to coordinator function, and set address
`AT+WWAIT=1000\r\n`	Enable data transmission waiting timeout (1 s)
`ATX\r\n`	Tell module to transmit any EVENT and DATA to us
`AT+WJOIN\r\n`	Join a network
`AT+WAUTONET=1\r\n`	Enable automatic network join on power up, reset and connection loss

Table 1: Recommended initialization commands for the ATZB-24. \r represents the ASCII for carriage ret(urn), and \n represents newline.
The value *MTNR* for WSRC shall be the last 16-bit, in hexadecimal, of your student ID. This is required as using `0xFFFF` for a stochastic assignment leads to error more often than we find reasonable.
Note that the SerialNet seems to have issues with leading zeros for the PAN ID and the channel mask.

4.3.3 FANtastic Protocol

We run a custom protocol which transmits and receives over SerialNet. The protocol allows to set the target RPM, alert the master about a noise-reduction, and allow the master to reset the noise-reduction mode resp. remote RPM mode.

On entering the noise-reduction mode, the text `NOISEALERT` shall be sent to the master. The master will send `RESETALERT` to reset an active noise-reduction mode.

Also the master may send `SETRPM:<param>`, where `<param>` is the new control target for the RPM. On receiving `RESETRPM`, the control target for the RPM is set back to the value determined from the potentiometer.

4.4 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 on the course homepage [6].

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 4.4.2) 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.

4.4.1 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).

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.

Note

In case you do not utilize this function in your application, include the possibility to test it!

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.

4.4.2 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.

Note

You may add functions to the driver if they provide an increase in performance your application's specific needs. This does not include functions which should be in a separate module, i.e., drawing of the complete UI.

4.4.3 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 [6, 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);
```

4.5 ADC

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

1. The ADC is used to convert the analog voltage of the potentiometer P5 to a digital value which is used as the target RPM for the fan.
2. Another channel of the ADC is used for sampling the output-signal of the microphone, which is used as input for the FFT.

The ADC driver has to be completely interrupt driven.

Note

Keep in mind that care must be taken when switching channels, as stated in [7, Sec. 26.5.1].

4.5.1 RPM Adjust

Increasing the RPM should be achieved by turning the potentiometer clockwise. Use ADC Channel 0 as input and apply a 5-sample median filter on the upper 8-bit of the conversion result. Scale the output of the filter into the range of 650 to 3400 RPM.[7] If you leave out the filter initially, you might have issues with the controller, due to the every changing value for the control target.

[7] The scaling does not need to be perfect. Depending on rounding errors in the calculations you might not even be able to achieve these values with the controller.

4.5.2 FFT

To compute the FFT, sampling of the microphone's output-signal is required. This signal is connected to Channel 7 of the ADC. To achieve good results for the FFT, we recommend sampling with a frequency in the range of 6–10 kHz. While we recommend a range for the frequency, to achieve correct and consistent results the frequency has to be constant in your implementation! Note that the FFT does not need run continuously, a few executions per second are enough. Consult the relevant section of the manual on how to configure the ADC to do this efficiently.

To compute the FFT over the sampled signal, we highly recommend you to use a library. For the development of the example solution, we used the "Fast fixed-point FFT modules" available at `http://elm-chan.org/cc_e.html`.[8] Of course, you can use another module, or even write your own FFT implementation. The specific FFT settings might vary between implementation, although we found that a 128-point FFT strikes a good balance between performance, used memory, and granularity of the spectrum.

5 Theory Tasks

In the theory task we want you to develop argumentation skills that allow you to reason about the problem you have to solve, and the solution you are designing. Clear presentation of ideas is crucial for communication with team members, bosses, customers, etc. We want you to precisely explain why a specific solution solves the problem or why not. Hence, we ask you to provide rigorous argumentation. Points are solely awarded for proper mathematical argumentation.

Consider a distributed system consisting of n sites, e.g., a microcontroller, each of which is operating their own fan. Each site has an own preference of the speed at which the fan should be operated. However, the technical process requires that the fans are operated at approximately the same speed. Hence, the sites are exchanging messages to agree on the speed.

Tasks

1. **[1 Point] The Problem:** There are n sites, each with a unique identifier in $\{1, \ldots, n\}$. Each site i has an initial preference value v_i, and a speed value s_i that is initialized to a default value $\perp$. Each site i sets s_i according to the following specification

   Progress. Each site i eventually sets s_i to a value different from $\perp$.

   Validity. For each site i, if i sets s_i to a value different from $\perp$, then s_i lies between the minimum and maximum preference values, i.e., $\exists j \in \{1, \ldots, n\} : v_j \leqslant s_i$ and $\exists j \in \{1, \ldots, n\} : v_j \geqslant s_i$.

   Agreement. For any two sides i and j, if both s_i and s_j is different from $\perp$, then $s_i = s_j$.

   - Consider the protocol (without communication) executed at each site consisting of one line: "`s := v`". Explain what parts of the specification does it solve, what parts are broken?
   - Consider the protocol (without communication) executed at each site consisting of one line: "`s := 42`". Explain what parts of the specification does it solve, what parts are broken?
   - Consider the trivial protocol (without communication) executed at each site that does nothing. Explain what parts of the specification does it solve, what parts are broken?

2. **[2 Point] A Protocol:** Design a distributed protocol in pseudocode in which processes exchange messages and use the preference values to compute a speed value. To send a message to site j use the primitive `send(j,msg)`. You can use the primitive `receive()` that removes the first message `(i, msg)` from the receive-buffer and returns it. If `(i, msg)` is returned, then `msg` was sent by site i. If there is no message in the buffer, then $(\perp, \perp)$ is returned.

   Assume that every message sent eventually will reach the receive-buffer, but message delays may vary arbitrarily. Argue why your protocol solves the specification from Point 1.

3. **[2 Point] Faults:** Assume the following failure scenarios

[8] The include for `avr/pgmspace.h` is missing in the `ffft.h` header of this library.

(a) messages can be lost

(b) messages cannot be lost but corrupted, that is, sites may receive messages with arbitrary content.

(c) messages cannot be lost but one site may crash.

- For each of the scenarios, explain how these scenarios influence your protocol. Which parts of the specification does it solve under the scenarios, which parts not?
- Is it possible to fix the protocol to solve the problem in the faulty environments?

6 Demonstration and Protocol

If you want points for your application, you have to submit a `*.tar.gz` archive to myTI until 14.05.2017, 23:59. The name of the top-level folder of the archive must contain your matriculation number. Inside the top-level folder place the Application folder from the template, and also include the Protocol folder with the Listing.pdf. You can use the provided Makefile of the template to generate an appropriate archive. See below for a link to the template which can also generated archives following this convention.

You are also obliged to participate in a delivery talk. The delivery talks will be held between the 15.05.2017 and the 18.05.2017. You must register in myTI for a 25 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 myTI, check its functionality on the labor 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 counted. We also require you to print out the first page of your Protocol (the version from the template is sufficient), sign it at the two marked spots, and give it to the tutor at the talk.

You also have to write a protocol 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 for which tasks you spent how much time (e.g., reading manuals, implementation, debugging, writing the protocol, ...). A layout-template for the code/protocol separation, including a LaTeX template, is available at http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/misc/task1-specific-stuff/application-protocol-template-tar.gz/view. The protocol, including the solutions for the theory tasks if applicable, has to be submitted as a `*.tar.gz` archive to myTI until 17.05.2017, 23:59. The name of the top-level folder of the archive must contain your matriculation number. Inside the top-level folder place the Protocol folder from the template. Additionally 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.

6.1 Checklist

- Write 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 myTI
- Download the submission from myTI and
  1. verify that the submission compiles in the lab and
  2. works on the target as intended.
- Enroll for a delivery talk.
- Write the protocol. Participate in the delivery talk. Prepare for the second exam.
- Generate a protocol archive via the provided Makefile target `protocol`

- Upload the archive to myTI
- Download the submission from myTI and check that all required files are included.

7 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 20 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 have to be done on your own!

Points	Sub	Part
20	2	Application
	6	RPM-Controller
	3	Noise-Controller/FFT
	3	ZigBee/UART
	4	GLCD
	2	ADC
5		Theory Tasks (every task is evaluated separately).

References

[1] Maxim, *Low-Cost, Micropower, SC70/SOT23-8, Microphone Preamplifiers with Complete Shutdown.* [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/chip-datasheets/max4465-max4469.pdf/view

[2] I. Delta Electronics, *Superflo FAN.* [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/aub0812vh-sp00-rev05.pdf/view

[3] MikroElektronika, *EasyBee Additional Board Manual.* [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/extension-boards/easybee_manual_v100.pdf/view

[4] Atmel, *ZIGBIT 2.4GHZ WIRELESS MODULES.* [Online]. Available: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/chip-datasheets/atmel-8226-zigbit-2-4ghz-wireless-modules-atzb-24-a2b0_datasheet-2.pdf/view

[5] Atmel Corporation, "AVR2051: SerialNet." [Online]. Available: http://www.atmel.com/Images/Atmel-8389-WIRELESS-SerialNet_UserGuide_AVR2051.pdf

[6] 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

[7] 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