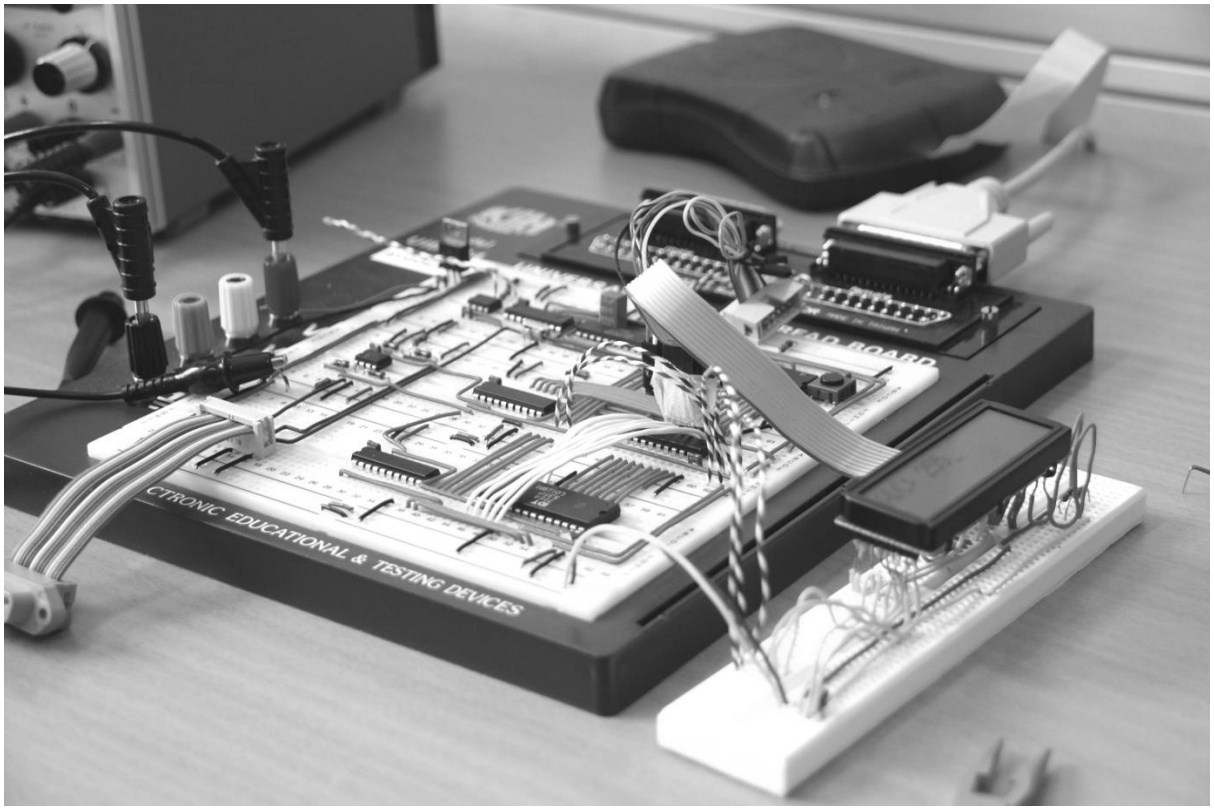
NTNU | Norwegian University of Science and Technology

Department of Engineering Cybernetics

TTK4155

Industrial and Embedded Computer Systems Design

Ping pong game with a distributed embedded control system



Version 6.0.0 - Aug 2025

Table of Contents

1 Introduction

This term project aims at creating a distributed computer-controlled electromechanical ping-pong game. The main challenge will be to assemble the hardware components and develop software for the microcontrollers, creating a fully functional embedded computer system that will enable you to play a refreshing game of ping-pong. The term project is, in essence, a rather comprehensive laboratory exercise that requires allocation of a substantial amount of time for lab work throughout the semester. To make the project easier to complete, it is divided into a number of smaller weekly exercises that will ensure continuity in the work and the necessary progress in the project. The electronic components, game boards, and development tools needed to realise the game will be handed out at the beginning of the semester.

Please read this **entire** document thoroughly. It has been developed and improved over many years and is your main guide to the project.

1.1 System overview

As shown in Figure 1, the system is divided into two main parts: Node 1 and Node 2. Node 1 consists of the breadboard and the User-I/O board, and Node 2 consists of an Arduino Due development board with a dedicated shield for motor control.
In the first exercises, only Node 1 will be used. Node 2 will be added later, when the game board and the CAN bus are included. Figure 2 gives an overview of the components that each of the two nodes consists of. As shown, the CAN bus enables communication between the two nodes and can also be used to connect two complete systems together in two-player mode.

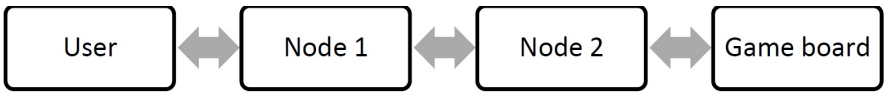


Figure 1: System overview

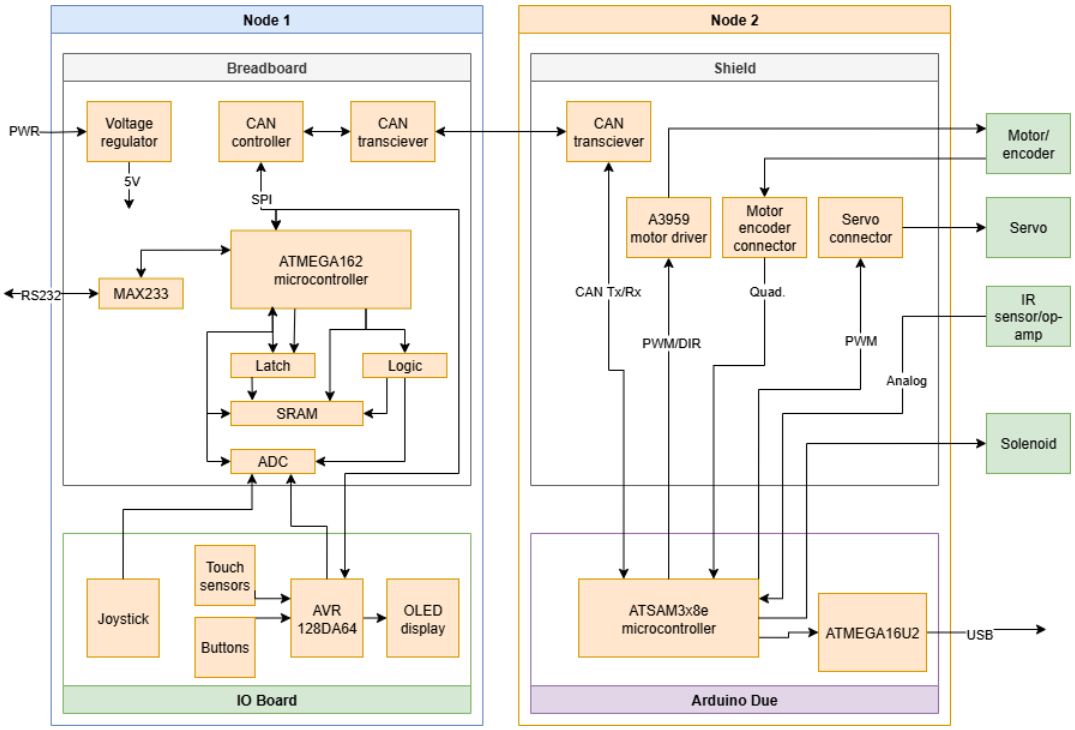


Figure 2: Node 1 and 2 with their main components

1.2 Practical information

1.2.1 Groups

The project will be carried out and evaluated in groups, each consisting of three students. Registration will be taken care in a shared Excel document published on Blackboard close to the first lab lecture. Each group will be given a unique number, which corresponds to a specific workbench number and its associated computer and equipment in the laboratory. As this project relies on the availability of equipment such as oscilloscopes, power supplies and so on, most groups will prefer to work with the exercise on the lab. **Other groups are using the same desks as you; make sure to tidy up before you leave!**

1.2.2 Lectures

Lab lectures will be held by the teaching assistant prior to the exercises. The topics will be directly relevant to the upcoming project exercise and will contain useful information, hints and tips.

1.2.3 Schedule

Table 1 gives an overview of all project exercises and a schedule showing which week they are expected to be completed. Most exercises are planned to take one week. Based on previous experience there will also be "catch up weeks", where groups can catch up with the plan if they fall behind schedule.

Week	**Exercise to be completed**
36	1: Initial assembly of microcontroller and RS-232
37	2: Address decoding and external RAM
38	3: A/D converting and joystick input
39	4: OLED display and SPI part 1, IO buttons
40	
41	5: SPI part 2 and CAN controller
42	6: Getting started with ARM cortex-M3 and communication between nodes
43	*Catch up week*
44	7: Controlling servo and IR
45	8: Controlling motor and solenoid
46	*Catch up week*
47	*Evaluation*

Table 1: Exercise plan

1.2.4 Approval of exercises

After completing an exercise, the result **must be shown to and approved by one of the teaching assistants**. This will help you verify that it is performed with sufficient quality and make it easier for you to proceed. If the teaching assistants are not available, you may start working on the next exercise to avoid delays, but make sure to get it approved as soon as possible. However, it is important to note that the approval of exercises by lab assistants are mandatory, but only provisional. **The formal and definitive approval of the lab project takes place during the evaluation week where the groups are required to demonstrate and explain their solution to the course leader/evaluators**.

1.2.5 Lab

The lab is located in the EL Building 2, second floor, room G203 and G204 (up the stairs near "Infohjørnet" helpdesk). If you are a student of Engineering Cybernetics or is signed up for TTK4155 this semester, you should have access to this room automatically. Otherwise, send an e-mail to the teaching assistant as soon as possible with your full name and NTNU-username.
The lab is reserved for this course a fixed period every week. The time will be announced in the beginning of the semester. Teaching assistants will be available in most of these periods. Beyond that, you are free to use the lab when it's not reserved for other courses. **Make sure to tidy up before you leave.**
There are lockers outside the lab where your group-specific components and equipment can be locked up using your own padlock. Use the locker marked with your group number.
Some of the equipment in the lab is shared with other groups and courses, so please **do not remove or lock in equipment that will be used by others such as game boards, motor interface boxes or oscilloscopes!**

Computers and equipment

The real-time lab is equipped with computers with all relevant software installed. The computers will mainly be used for developing software in C for the Microcontroller units (MCUs), and reading datasheets of various components. You cannot install additional software on the computers. If you feel that vital software that can be beneficial to all groups are missing, contact the TAs and perhaps the software can be acquired and all computers updated. Specialized software e.g. for your "extras" must be run on your own laptop. **Do not store your project locally**, but use GIT/SVN or your home/student folder. Whenever a new image is rolled out to the computers in the lab, all local data will be deleted. This might happen several times during the semester.

The lab is also equipped with oscilloscopes, multimeters, signal generators and other necessary tools.

1.2.6 Standard components

The development kit required to carry out the project will be handed out at the beginning of your first lab day.
The kit consists of relatively fragile components and should be handled with utmost care and **be returned to the component store in good and tidy order immediately after the final evaluation (see information in "Kit list" document published on Blackboard). This is a requirement in order to get access to the final exam**. Damaged components should be handed to the teaching assistants.

Component store
Components and tools such as wires, screwdrivers, pliers, jumper cables, headers etc. can be handed out at the lab. You should have ample access to e.g. wires to make your connections neat. If the required components for some reason are not available in the laboratory, you may contact the personnel at the component store located in the Electronics Building D, room D-040, where you can get some common components. **Always remember to sign out the components you get** in the corresponding binder in the section marked TTK4155. The people in the store will assist you in finding the right components.

Component list
An up to date list of the content of the development kit is published on Blackboard. Please make sure that you have received all tools and components in the list. If not, contact the teaching assistants or personnel at the component store immediately.

1.2.7 Documentation and datasheets

All necessary datasheets are published on Blackboard. **Do not print these**, as they are several hundred pages, and you only need small excerpts of them. In addition, a searchable and indexed PDF is much more useful than a pile of paper. Printing small parts of the datasheets, e.g. pinouts, might still be helpful.
Microchip Studio's help system also contains a lot of valuable information. For information about the various development tools and debugging devices, see `Help > View Help`. Also, the Microchip Studio User Guide describes the software itself.

Useful links
AVR Freaks (support site with forum and articles):
`http://www.avrfreaks.net/`

Microchip Studio YouTube Channel

AVR Libc (description of many of the libraries available, as well as how the AVR architecture works):
`http://www.nongnu.org/avr-libc/`

SAM3X - Arduino DUE pin mapping:
`https://www.arduino.cc/en/Hacking/PinMappingSAM3X`

Google (search for a part number – you might find datasheets, application notes, hints and tips):
`https://www.google.com`

1.3 How to work with the project

The project is comprehensive and rather time consuming, which is why it is divided into several smaller exercises. **Be careful and thorough when implementing each part** as later exercises will be based on the work done in earlier exercises and the final result will depend on the quality of all parts. The output of most exercises should be presented as a separate and fully functioning electronic circuit with its associated software driver, which can be used as a building block for later exercises. **To ensure modularization and reusability, organize your code as compact drivers with logical interfaces.** More information about how to do this can be found in chapter 2.2.1.
When working with an exercise, **test** what you have done **frequently** and verify that it works the way it is supposed to. Assembling everything and writing software without stepwise testing is prone to failure.
Use the web frequently. Many of the problems you might encounter have been solved by others before you. Still, **you may not copy other people's work or code except where it is explicitly stated in this assignment text or when distributed by the course staff (e.g. on Blackboard).** Remember to always cite your source for code not written by you. Your code will be run through an rigorous plagiarism test, and **any matches not declared through a citation may be dealt with according to NTNUs guidelines** for Cheating on exams available at innsida.
Read through this document thoroughly before asking for help. It contains necessary background information, specific steps needed to complete the exercises, useful links and pointers to documentation, and hints and tips concerning frequently occurring problems and mistakes.
This text also describes how to debug your hardware (2.1.10) and software (2.2.8-2.2.9) in an efficient manner. These methods are the same as the teaching assistants would use, so make sure you have done everything that is described in the debugging guides before asking for help.
Datasheets for all relevant electronic components will be published on the course's Blackboard site.

Make sure to read these carefully. Most of the information you need is contained in these documents.
If you finish an exercise early, or have some extra time one week, **you are strongly recommend to move on with the next exercise.** Many groups have been delayed because of faulty components, hard-to-find bugs or difficult tasks, and being one or more weeks ahead can prove valuable at a later stage. Also, if you finish the whole project early, you will have more time to use your creativity and implement extra features, which will improve the final result and probably impress the evaluators. Make sure you visualize the project in a larger context than just the scope of your group and the assignment at hand. E.g. would another developer who has never seen your code before be able to understand it and continue development easily? Although something works, it does not mean it is implemented in a good way. Do you really need to update the display at 1000 Hz, or could you save some CPU capacity for other tasks that may come?

2 Background Information

2.1 Hardware and electronics

2.1.1 Breadboard

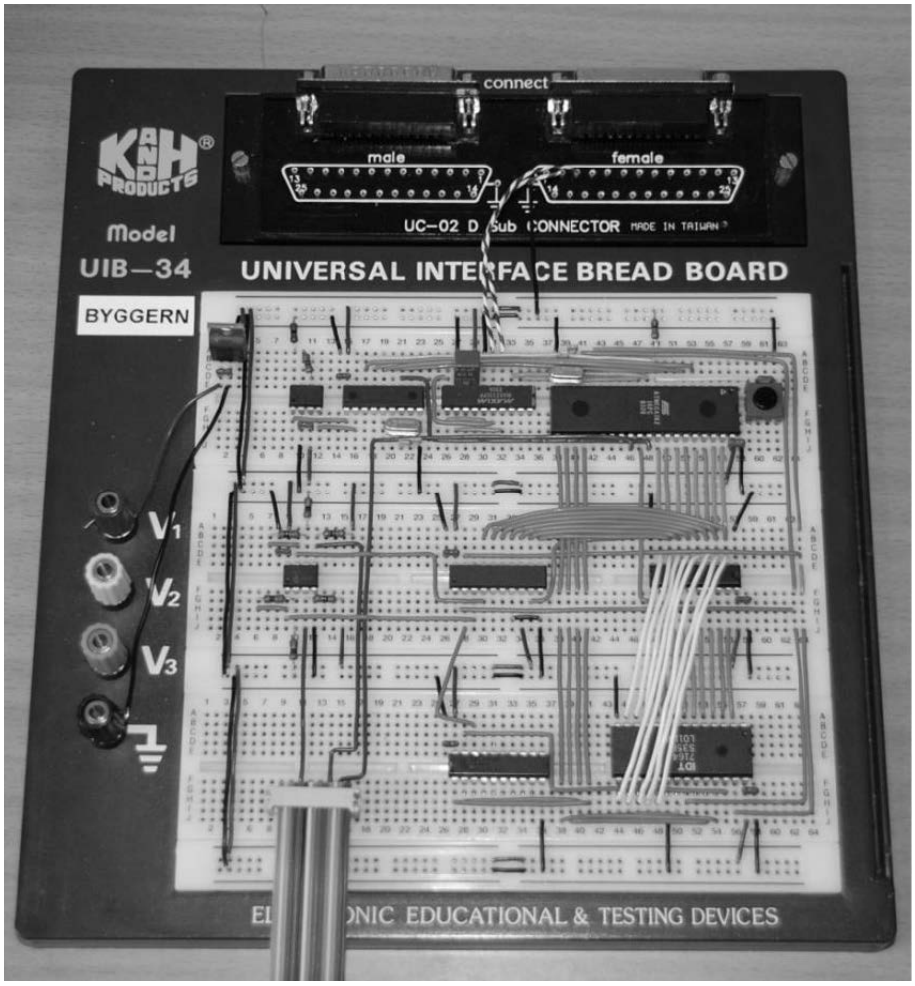


Figure 3: Breadboard

A breadboard is a construction base for prototyping of electric circuits. Integrated circuits (ICs) and wires can be placed directly into the board without any soldering. This allows one to build and modify a circuit quickly and easily. Holes are interconnected in groups, so that one can place an IC without short-circuiting its pins and access its pins from several holes. Figure 4 shows a typical hole pattern for a breadboard. Note the power buses marked with red and blue on the breadboards.

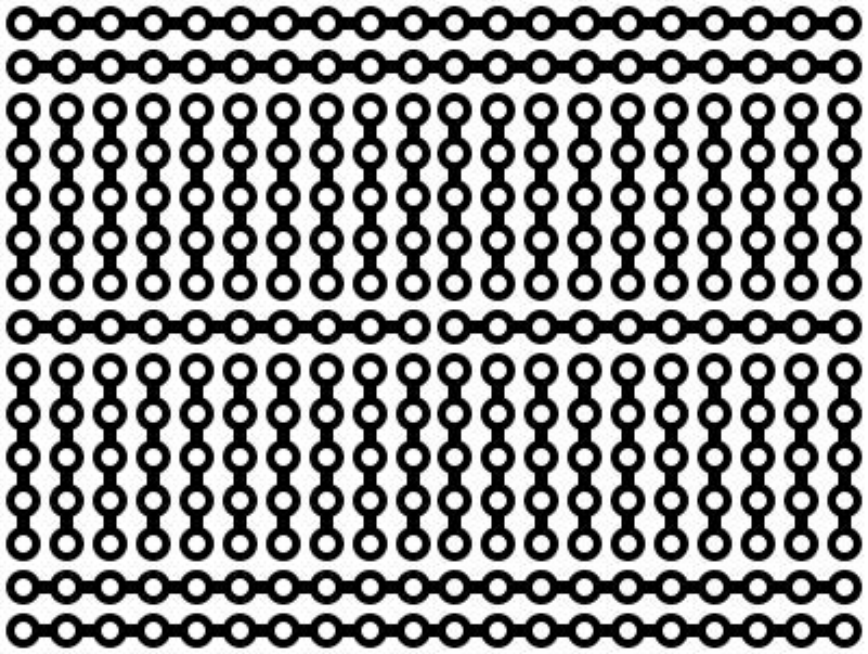


Figure 4: Typical hole pattern of a breadboard (Wikipedia)

Node 1 consists of a breadboard where you are going to place ICs and connect them by wires. **You are strongly recommended to follow the directions for IC placement given in Figure 5**. This will make it easier for the TAs to debug your wiring. Also, make sure to place the components in the correct direction as this will save you some wiring – note the arrows on the ICs in the figure. When adding and connecting new components it is relatively easy to end up with a chaotic "rat's nest" of tangled wires which is almost impossible to work with. Care must therefore be taken to use cables of proper length and color, and to place them in a systematic manner. Keep the wiring close to the breadboard and avoid diagonal wires. Ask the TA's if you need more/longer cables if the supplied set or the reserve at the lab doesn't suffice. This extra effort will pay off when new components have to be added to an already densely populated breadboard.

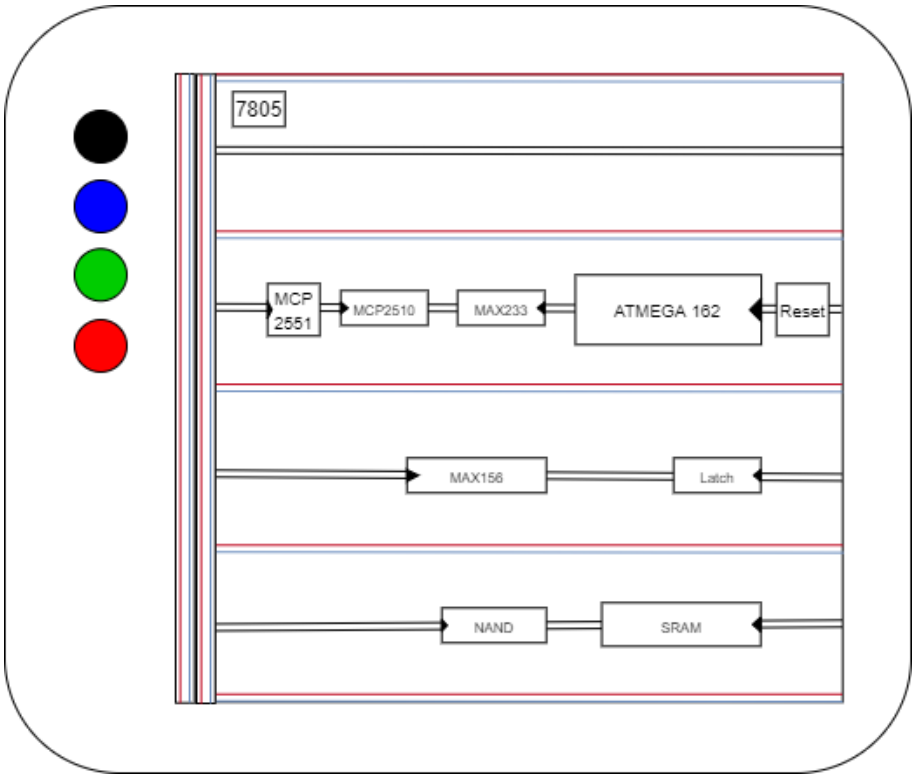


Figure 5: Component placement on node 1.

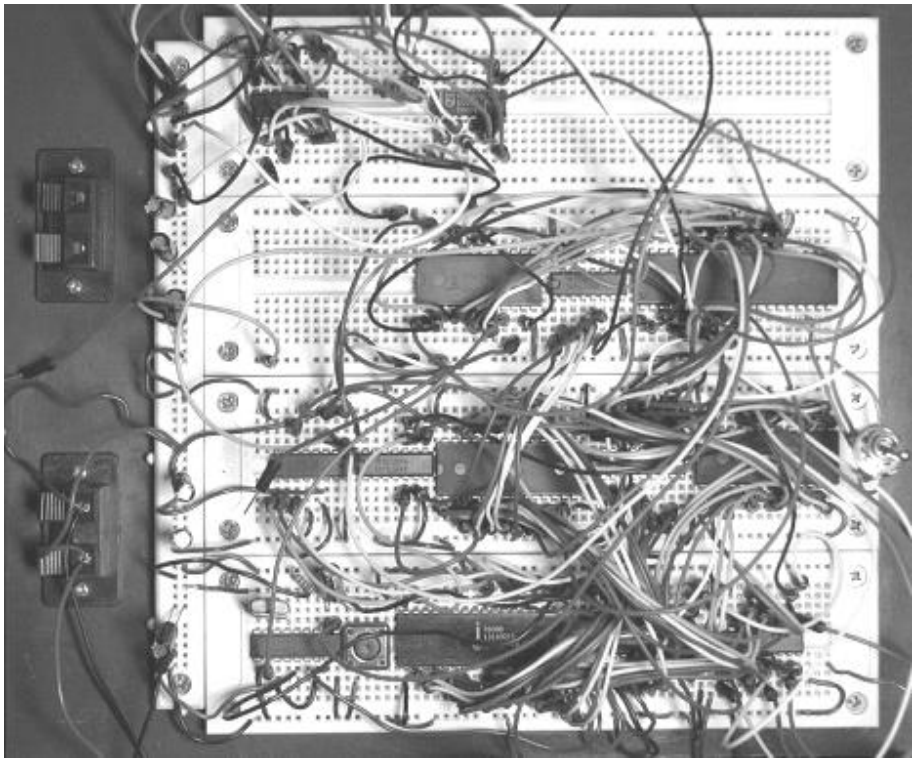


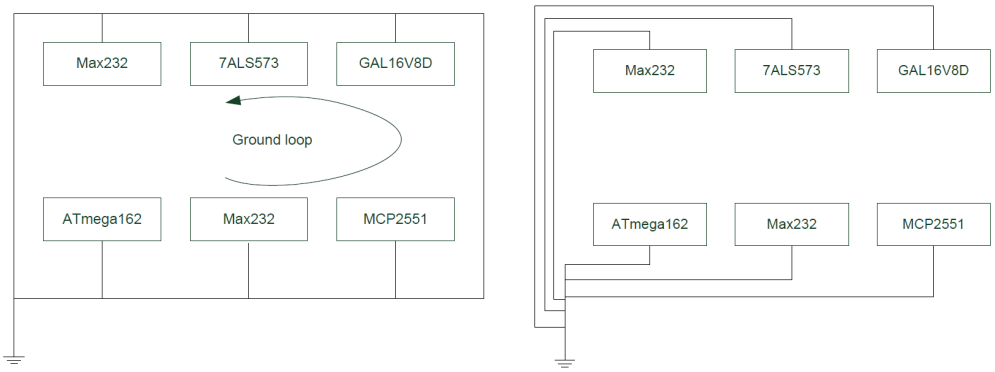
Figure 6: Messy assembly (Wikipedia)

2.1.2 Noise, grounding and decoupling

Although being quick and easy for circuit prototyping, electrical noise is an inherent problem with all breadboards. Thus, proper decoupling of all ICs and ensuring good ground connections, especially to the crystals and microcontroller, is extremely important. Connect all ICs with star-point grounding (all ground connections should originate from the same physical point). The same principle applies for the voltage supply. Try to experiment with the grounding if you experience noise problems. Typically, SRAM read or write errors are clear indications of noise related problems.
You should avoid (large) closed loops of wire while trying to make all connections logical and well arranged at the same time.

Decoupling
Unfortunately, the supply lines to the circuits are not ideal as they all introduce certain amount of impedance (resistive, inductive and capacitive) between the power source and the IC, which may lead to undesirable AC effects and noise that affects IC operation. Moreover, the power source itself has a finite response time and can only accommodate changing demands in current supply up to a limited frequency (depending on supply type). As shown in the datasheets, the power consumption of an IC is not constant and they usually draw a highly variable amount of current while operating, e.g. when CMOS circuits change their logical state. This combination will result in noise-like voltage drops occurring at the voltage supply pins of the IC, which can cause considerable trouble and malfunction of the circuit. Furthermore, the problem will typically get worse and more unpredictable when several ICs are connected to the same supply lines.
The most important remedy against these undesirable effects is the decoupling capacitor. The decoupling capacitor is a capacitor that is connected between ground and the voltage supply pins of the IC. Importantly, it must be connected as (physically) close to the IC as possible in order to limit the effect of lead impedances. The decoupling capacitor will then operate as a fast local power source that will counteract voltage drops during transient changes in IC power demand ("decoupling" the IC from the central power supply). Moreover, due to its impedance characteristic ($\frac{1}{j\omega C}$) the decoupling capacitor will work as an effective shunt against noise signals of certain (high) frequencies that may occur on the supply lines. Capacitor value depends on the noise frequencies but 10-100 nF seems to be a good compromise in many situations.

Star-point grounding
Star-point grounding is an important concept. In the connection demonstrated in Figure 7, the voltage difference from "real" ground to an IC's ground terminal will depend on the current consumption of the other ICs along the ground rail because of non-zero impedance of electrical conductors. Thus, the ground potential of each IC is bound to be different between the different ICs in the circuit and will depend on the ground return current, which is clearly an undesirable situation. In addition, if the ground wires are connected as a closed loop as shown to the left in Figure 7, known as a ground loop, a particularly critical situation may appear. The circuit will be very vulnerable to induced electrical noise from surrounding magnetic fields (motors, solenoids, transformers etc.) and this type of connection has to be avoided at any cost. The right hand side of Figure 7 shows how this problem can be remedied by connecting all ground return wires to a common point close to the supply ground in a star-like connection. It is also desirable to reduce the length of the common ground wire as well as attaining a more balanced wire length. Star-point connection should also be used for the voltage supply.



Ground loop (wrong) Star grounding (correct)

Figure 7: Grounding

Power drain
Some components, especially the solenoid, drain a lot of transient current when activated. This might lead to voltage drops on the supply lines and ground bounce with subsequent reset of microcontrollers and malfunction of communication lines as a result. Decoupling capacitors may not be able to absorb noise transients of those magnitudes and isolating such devices to a separate voltage supply may prove to be the best solution.

2.1.3 LEDs

LEDs can be a valuable debugging tool for instance when used to display the logical value of a signal. Moreover, they often serve as indicators showing that e.g. the power supply is enabled, the system state is initialized etc. It is also common to connect LEDs to digital output pins and control these from software to indicate different states, values, control flow and so on. For example a LED that toggles for each n-th iteration of the main program loop, provides a visual indication that the code is in fact executing.
Figure 8 shows two different LED configurations – one gives light when the output signal is high, the other one when the signal is low. Intuitively, a lit diode could indicate a high signal, but most logical circuits can sink more current than they can source, and thus one should consider using the active low configuration. Another solution is to use a LED driver circuit (buffer/amplifier), but this requires a slightly more complicated circuit. Be sure to check in the datasheet that the IC output is rated for the current source/drain that is needed to operate the LED correctly.

2.1.4 Pull-up and pull-down resistors

Some ICs and circuits are deliberately designed to only be capable of driving their output signals low (open collector/drain). Outputs cannot be actively driven to a high level, that is, the output is left floating/high impedance as long as the output is not low. In such cases a high output level has to be ensured by connecting an external resistor, also known as a pull-up resistor, between the output and the logical high voltage level (whatever that might be).
Pull-up resistors have many uses and are essential e.g. in open collector logic, level conversion and communication buses. The resistor value affects the transient response of the output and the current consumption, and has to be selected accordingly. Typically, values in the range $10k$ - $100k$ are used with CMOS circuits.
Pull-down resistors works in the same fashion, only the other way around with the resistor connected between the output and the ground (low) level.

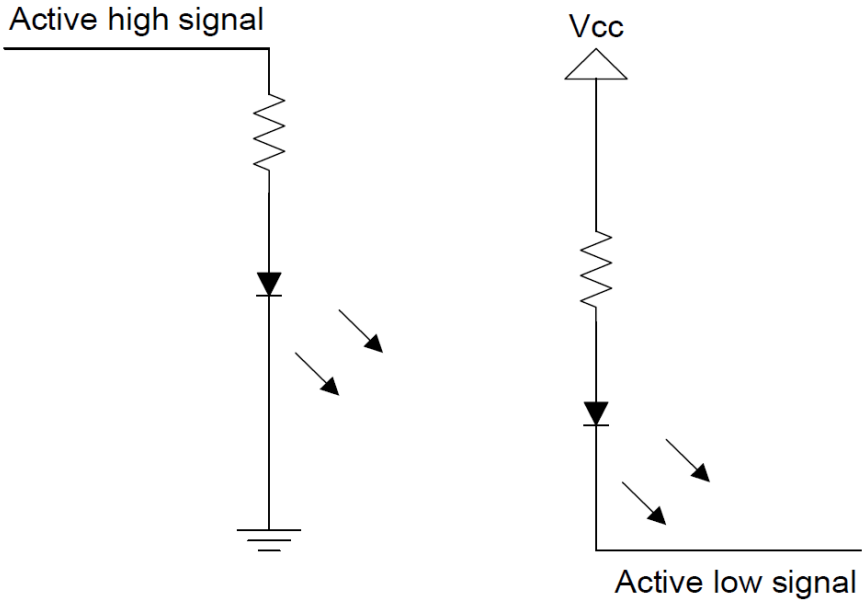


Figure 8: Led connections

2.1.5 The Microchip AVR microcontrollers

The AVR microcontroller[1] used in this project is based on an 8-bit RISC (Reduced Instruction Set Computer) CPU with a modified Harvard architecture (that is, separate memories for program and data). The RISC architecture implies that almost every instruction can be carried out in just one clock cycle. However, like most RISC processors, data in memory has to be explicitly loaded into the processor's register file first before they can be used as operands to an instruction, and results must be explicitly stored back from the register file to data memory (using load and store instructions). I/O registers can be operated using special instructions (read-modify-write).
The various MCUs in the AVR family have different built-in support for communicating with external devices. In this project we will make use of modules for UART and SPI communication. We are also going to use the external memory bus interface to connect an SRAM and other devices. The different MCUs of this family also have a rich set of other peripherals and features, which are described on Microchip's product pages and datasheets.
We will use an ATmega162 in Node 1. Most of the information required to start developing with it is contained in its datasheet.

2.1.6 The ARM Cortex-M family and the Microchip SAM microcontrollers

The ARM Cortex-M is a family of 32-bit RISC processor cores designed for use in both microcontrollers and SoCs such as power controllers, I/O controllers and touch screen controllers. The ARM Cortex-M family contains many different core designs with both Harvard (M3, M4, M7 and more) and Von Neuman (M0, M0+, M1, M23) architectures. The core design is made by ARM who sell their designs as intellectual property (IP) instead of manufacturing their own silicon chips.
The Microchip SAM microcontroller used in this project (ATSAM3X8E) is based on the ARM Cortex-M3 architecture (`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.cortexm.m3/index.html`) which implements the entire Thumb-1 and Thumb-2 instruction sets together with some other instructions. Together with the ARM core, the microcontroller includes many peripheral functions such as timers, analog interfaces (ADCs and DACs) and serial communication such as USART, SPI, I2C and CAN. You might also find other peripherals and

[1]The AVR architecture was originally conceived by two NTNU students and the first AVR MCUs was brought to the market by Atmel Corporation in 1997. Atmel was acquired by Microchip Technology in 2016, however, it is still frequently referred to as the Atmel AVR, or just AVR.

features relevant to the project. Use the datasheets and have a look at Microchip's product pages for more information.

2.1.7 User-I/O board

The User-I/O board contains a Microchip AVR128DA64 microcontroller. It has already been flashed with necessary drivers for the hardware units on the board, and you are not expected or encouraged to reprogram it. You will communicate with it via an SPI bus which you will set up about halfway through the project. The 2x7 header contains multiple signals, among others the raw output from the joystick. The board has the following hardware units:

- Microchip AVR128DA64 microcontroller
- 128 x 64 pixels OLED display
- Thumb joystick with button
- 8-way HAT switch (small 'joystick')
- 11 buttons
- Touch slider
- Touch pad
- 6 LEDs

Refer to the User-I/O board manual published on Blackboard for further details on its operation and interface.

2.1.8 Arduino Due

Some of you might be familiar with the Arduino boards. The physical layout and simple programming environment are equal for all Arduino boards, spawning a huge amount of modular hardware, blogs and guides.
The Arduino is usually programmed through the USB interface using for instance the Arduino IDE. This gives access to a vast number of libraries and offers rapid development. However, we will use the JTAG interface and Atmel Studio to program the Arduino. Although you will not have access to the Arduino libraries, the code will more closely mirror that on node 1, debugging is vastly simplified and one avoids the struggle with many Arduino libraries which are poorly implemented.

2.1.9 Atmel-ICE

The Atmel-ICE is a powerful development tool that lets you debug programs that are actually running on the AVR microcontroller silicon (on-chip debugging). Together with Microchip Studio you can step through code line-by-line, read and manipulate most I/O and control registers directly, insert breakpoints and so on. This tool is also used to upload programs (compiled code) to the microcontroller's onboard flash memory.
The interface and language used to communicate with an AVR during debugging is called JTAG. This is one of many programming and debugging standards, most AVRs support both JTAG and programming via SPI, called In System Programming (ISP). To prevent outside sources from recovering a program, reprogramming a chip after production or to free up the pins used for the programming interface, it is possible to disable the JTAG and ISP interfaces through internal settings, or fuses in the microcontroller.

2.1.10 Hardware debugging

There ***will*** be errors and bugs when assembling the breadboard - and that's completely normal. Debugging might be difficult, especially for the teaching assistants who don't know the details of your setup. Before asking for help, make sure to go through the following debugging steps, make schematics and ensure that you follow the tips given here and in the relevant exercise. Read the instructions given in the exercise again and make sure you have carried out the steps exactly as they are described there.

1. If you have completed an exercise without testing and then run into problems, you should backtrack and verify that each step is done correctly.
2. The lab is provided with multimeters and oscilloscopes. Use these frequently when assembling and debugging the hardware and software to verify that everything works correctly.
3. Read the datasheets carefully, they contain (almost) everything you need to know.
4. Verify all connections (pin to pin, not wire to wire) using the multimeters continuity test (**remember to turn off power supplies first!**).
5. Remember that tantalum capacitors are polarized. Verify that all capacitors are connected the right way.
6. Check that you have connected positive voltage and ground correctly to all ICs (otherwise, you might have damaged the ICs).
7. Ensure that the power supply is on.
8. Ensure that all ICs are pressed all the way down into the breadboard.
9. Verify that you have remembered decoupling capacitors for all ICs and that they are connected nearby their voltage supply and ground pins.
10. Measure the voltage between the ground pin and voltage supply pin on the ICs.
11. If the IC should be communicating with other circuits, use the oscilloscope to see if there is any activity on the bus and if it seems to be correct.
12. Borrow ICs from another group to eliminate code errors, but do not fry more chips than necessary (verify supply voltages!).

2.1.11 The Oscilloscope

The lab is equipped with high-end, four-channel oscilloscopes taking samples at 300MHz and supports decoding several digital protocols through the logic analyzers. An oscilloscope records and visualizes voltage levels across time, with time on the x-axis, and voltage on the y-axis. Important parameters to set on the oscilloscopes are:

- Vertical scale: Determines voltage range on the y-axis.
- Timebase (time horizon): Balances the sampling-rate. The oscilloscope only has a fixed amount signal storage, so sampling rate is reduced for large time horizons, and increased for smaller time horizons.
- Trigger condition: Under what condition should the oscilloscope capture new data. Often, a rising or falling edge at a certain level.
- Trigger level: The voltage level checked by the trigger condition.
- Trigger mode: Single-shot, normal mode or auto. Single-shot stores data from a single trigger event, normal mode updates data at each trigger event and auto is the same as normal, but forces a trigger event after a certain time has passed.

Short application note:
https://www.liquidinstruments.com/blog/2019/09/27/introduction-to-oscilloscopes-an-oscilloscope-familiarization-guide-with-mokulab/

The oscilloscope probes are costly to replace, so we ask you to please threat them gently. The sharp end shall not be inserted direcly to the breadboard, and is only intended for directly measuring at IC pins etc. For measuring on the breadboard, you should instead use the attached hook with an attached lead to measure on the breadboard. The same goes for the ground crocodile clip, connect it to the breadboard via a small lead.

2.1.12 Tips

- The ICs must be handled with care. Use the ESD-equipment provided to prevent damage from electrostatic discharge as CMOS-chips are particularly sensitive to this problem.
- Turn off the power supply when working with the components, e.g. mounting new ICs or connecting wires.
- Use the colored wires cleverly. Color-coding will help you identify different kind of signals and wires. For instance, ground and voltage supply is often black and red, respectively. Data buses, interrupt signals, analog signals etc. should have their own colors.
- Remember that not connecting a signal is not the same as setting it to 0! Unconnected pins will probably float and take on random states. Ensure that all unused input pins of an IC are connected to either supply voltage or ground.

2.2 Programming in C

C-programming skills are essential to this project. If you don't master C sufficiently, you are recommended to buy a book or go through tutorials on the web. Books such as The C programming Language, Kernighan & Ritchie or C Programming: A modern Approach, K.N. King are recommended. The following links might also provide good help:
`http://www.cs.cf.ac.uk/Dave/C/`
`http://publications.gbdirect.co.uk/c_book/`
`https://github.com/TTK4235/Intro-to-C-Linux`

2.2.1 Modularization and drivers

As the project goes on, the number of code lines will increase steadily and the software will at some point become too complex to handle if care is not taken to write maintainable code. Also, different parts of the system interfere with each other and errors might be very hard to resolve if the code is not broken into logical and manageable modules with clear interfaces (modularization). Well-designed code will also make it easier for the teaching assistants to provide help when needed. A good way to make the code in this project easy to follow and maintain is dividing it into driver modules where possible. The more complex exercises with several ICs and peripherals used, as the CAN exercise, could be divided into several layers; for example an SPI-driver, a driver for the MCP2515 CAN controller, and a high-level interface for sending and receiving CAN messages.
For each driver, make one header file (.h) and one code file (.c). The header file should provide an interface in terms of prototypes of each function that other modules should be able to call, and if really necessary, shared variables. Then, the header file may be included in the code files of other modules that need to use this drivers' functionality.
Things you might want to avoid is unnecessary dependencies and global variables. For instance if your CAN module is dependent upon the MCP2515 module it's generally harder to reuse the CAN module for other CAN controllers. And likewise if the CAN controller driver is dependent upon your SPI driver it's harder to reuse the module for the same CAN controller over parallel bus interface (not applicable for MCP2515 but more as a general example). Global variables allow

non-obvious side effects which might be fine if you never do mistakes, but for the rest of us there's "always" a better alternative.
`http://www.fast-product-development.com/modular_programming.html` gives some quick hints.
`http://www.icosaedro.it/c-modules.html` provides guidelines for modularizing C code. Note that the EXTERN and _IMPORT macros are not strictly necessary as this can be done in different ways.

2.2.2 Documentation

As far as possible, your variables, functions and defines should have self-explanatory names. This eliminates the need for extensive comments and explanations.
However, you should comment/document the code you write in such way that functions without self-explanatory names or hard-to-read algorithms can be understood by other people (and yourself later). This will also help evaluating the project, and if you have done a good documentation job it will be easier to achieve a good evaluation.
Automated tools that will generate on-line documentation in HTML from the comments in your source files are freely available, for example Doxygen. See `https://www.doxygen.nl/index.html` for more information.

2.2.3 Ports

Most of the pins on AVRs can be configured by the application software as digital inputs or outputs by way of dedicated control registers. Groups of pins are further organized into ports, each having 8 pins. For instance, ATmega162 has 4 ports: PORTA, PORTB, PORTC and PORTD. Each port on the AVR MCU used in this project is controlled by three 8-bit registers: PORTx, DDRx and PINx (where x is the port name, for instance PORTA). Each bit in the registers corresponds to one pin on the port. The DDR registers are used to configure a port as an input or output, and different pins might have different directions within a single port. Setting a bit to '1' designates it as an output, and '0' as an input. For instance, setting DDRA = 0x51 (= 0101 0001) makes the pins PA0, PA4 and PA7 outputs, and the rest inputs.
The PORT registers have two functions, depending on the pin being an input or output. If the pin is an output, the PORT register is used to control the states of the pins – '1' sets a pin to a logical high value, and '0' sets it to a logical low value. If it is an input, the PORT register enables an internal pullup on the pins set to '1'.

PIN registers can only be read, and each bit will correspond to the state of the digital inputs.

Note that most pins have shared functionality (multiplexed). That is, other peripheral units on the microcontroller might override or be overridden by these. For instance, the external memory interface of ATmega162 occupies PORTA, PORTC and bit 6 and 7 from PORTD. It is possible to mask out the most significant bits of PORTC if some of these pins are needed for other purposes.

Compared to the 8-pin ports of the ATmega162, the ATSAM3X8E have ports with up to 32 pins each. The pins are controlled by the Parallell IO (PIO) controllers. All IO-pins have several input and output modes such as pullup, multi-drive (open drain) and input change interrupt. They are also multiplexed, giving the opportunity to switch between different peripherals on the same IO-pin (many peripherals are also available on two or more IO-pins). For more information about PIO, see Section 31 in the ATSAM3X8E datasheet.
The most important PIO registers for this project are listed in Table 2. As an example, PIO can be enabled on Port A, pin 1 by writing `PIOA->PIO_PER = 0x1u << 1;` When using the pin for a on-chip peripheral, it has to be disabled in the PIO_PSR register. See Figure 31-3 in the SAM3X8E datasheet
Pitfall: Remember to enable PIOA in PMC (Described in Section 28 of the datasheet)!

Offset	Register	Name	Access
0x0000	PIO Enable Register	PIO_PER	w
0x0004	PIO Disable Register	PIO_PDR	w
0x0008	PIO Status Register	PIO_PSR	r
0x0010	PIO Output Enable Register	PIO_OER	w
0x0014	PIO Output Disable Register	PIO_ODR	w
0x0018	PIO Output Status Register	PIO_OSR	r
0x0030	PIO Set Output Data Register	PIO_SODR	w
0x0034	PIO Clear Output Data Register	PIO_CODR	w
0x0038	PIO Output Data Status Register	PIO_ODSR	r - r/w
0x003C	PIO Pin Data Status Register	PIO_PDSR	r

Table 2: Useful PIO registers for ATSAM3X8E

2.2.4 Bit manipulation

Configuration and control of a microcontroller and its peripherals are usually carried out by manipulation of bits in dedicated control registers. Hence, register oriented programming and bit manipulation comprises an important part of any microcontroller program. Fortunately, the C language is furnished with a rich set of bit manipulation operators that let you modify the bits you want. Depending on the compiler, these will often compile into efficient architecture-specific bit manipulation instructions. The header file `avr/io.h` in AVR Libc associates the names and addresses of all AVR registers as well as the bit names and positions for each bit in the various registers. This allows you to manipulate registers and bits using their logical names instead of numeric addresses and bit position s. This file will be added to your project for Atmel Studio users, for Linux `avr-gcc` will have the path to this file. If you want to take a look at the file specific for the Atmega162 it can be found here: `https://github.com/vancegroup-mirrors/avr-libc/blob/master/avr-libc/include/avr/iom162.h`.

Some useful expressions for manipulating bits:

```
1   /* Set pin 0 on port A */
2   PORTA |= (1<< PA0);
3
4   /* Only the bit in position CS02 set: */
5   (1 << CS02);
6
7   /* Only the bits in position COM01, COM00, and CS02 set:*/
8   (1 << COM01) | (1 << COM00) | (0 << WGM01) | (1 << CS02) ;
9
10  /*Set the bits CS02 and COM01 in TCCR0 register, clear the other bits: */
11  TCCR0 = (1 << CS02) | (1 << COM01);
12
13  /*Set the bits CS02 and COM01 in TCCR0, leave the other bits unchanged: */
14  TCCR0 |= (1 << CS02) | (1 << COM01);
15
16  /* Clear the bit CS02 in TCCR0, leave other bits unchanged: */
17  TCCR0 &= ~(1 << CS02);
```

Figure 9: Bit manipulation

For more information about bit manipulation, have a look in a C-language reference book or simply do a Google search. You will need to do a lot of operations like these, and defining macros can be useful. An example of some bit manipulation macros follows:

```
#define set_bit( reg, bit ) (reg |= (1 << bit))
#define clear_bit( reg, bit ) (reg &= ~(1 << bit))
#define test_bit( reg, bit ) (reg & (1 << bit))
#define loop_until_bit_is_set( reg, bit ) while( !test_bit( reg, bit ) )
#define loop_until_bit_is_clear( reg, bit ) while( test_bit( reg, bit ) )
```

Figure 10: Bit manipulation macros

To avoid data hazards, the ATSAM microcontroller has dedicated set, clear and status registers for many of its peripheral functions. You can set, clear or read a output pin in the Parallel IO (PIO) by writing to the registers as shown in Figure 11:

```
#include "sam.h"

//PIO - Parallell IO

/* Set port A pin 1 high */
PIOA->PIO_SODR = PIO_SODR_P1;       //SODR - Set Output Data Register
                                     //PIO_SODR_P1 = (0x1u << 1)

/* Set port A pin 1 low */
PIOA->PIO_CODR = PIO_CODR_P1;       //CODR - Clear Output Data Register
                                     //PIO_CODR_P1 = (0x1u << 1)

/* Read status of port A */
uint32_t status;
status = PIOA->PIO_ODSR;             //ODSR - Output Data Status Register
```

Figure 11: Bit manipulation for an output pin in ATSAM3X8E

2.2.5 Polling and interrupts

Embedded systems are typically required to react within reasonable time to relevant events occurring in their environment and thus need an efficient mechanism to detect and handle the situation when something happens. Two common and essentially different techniques exist to accomplish this task: polling and interrupts.
Polling, also known as busy waiting, means that the processor continuously monitors if the event has happened, typically by actively checking for a bit change in a particular status register. This technique is easy to implement but will consume a lot of resources in terms of processor time. If many events have to be monitored concurrently the polling technique may turn out to be too slow and/or other important tasks could be starved for processor time.
In contrast, interrupts automatically divert the processor from the execution of the current program to a dedicated piece of code that deals with some event that has occurred, for instance that an ADC conversion has completed or that the logical state of an input pin has changed. Events that may trigger an interrupt will vary between different processors and architectures. The occurrence of an interrupt will automatically cause the processor to save its current state and call a dedicated subroutine, known as the interrupt service routine (ISR), corresponding to the identity of the interrupt. When an interrupt has occurred, the processor determines its identity in essentially two ways, either by polling all interrupt generating devices to find out which one was responsible for the interrupt (polling interrupts), or by receiving the identity code directly from the interrupting device (vectored interrupts). The identity code is used to look up the address of the corresponding interrupt service routine in the processor's interrupt vector table and then a call to this routine will be executed.
Interrupts represent a considerably more advanced and complex mechanism than polling. Even though they come with a cost in terms of saving state information before executing the ISR, they more than often gives a huge advantage in terms of less waste of processor time and better response

times. Whether polling or interrupts should be used have to be decided for each individual case based on knowledge of the properties of these mechanisms and how they match the requirements of the particular application.
Many embedded applications require the system to wake up periodically to execute various tasks. While not using an operating system, this functionality is most elegantly achieved by using the MCU's timer/counter unit and its associated timer interrupt. With some careful programming and extra effort around the timer interrupt this can be developed into a lightweight version of the traditional OS task scheduler mechanism. By default, when an interrupt handler is called, other interrupts remain disabled. This restriction is possible to evade if needed (be careful!), see the AVR Libc manual. You can also enable or disable all interrupts on the AVR at any time using the functions `sei()` or `cli()`, respectively. All interrupts on the SAM can be enabled or disabled with the functions `__enable_irq()` and `__disable_irq()`. This can be useful in critical sections of your code (where interruptions are unacceptable).
How to create interrupt routines is described in the AVR Libc manual and how to activate the hardware interrupts of an AVR/SAM is described in the AVR/SAM datasheets.

```
1  // POLLING EXAMPLE (ATmega162)
2  // 8 buttons connected to PORTA, 8 leds to PORTB
3
4  #include <avr/io.h>
5
6  int main() {
7      // Buttons input (the whole port)
8      DDRA = 0;
9      // Leds output (the whole port)
10     DDRB = 0xFF;
11     while(1){
12         // Read A, set it out on B
13         PORTB = PINA;
14         //Do something else in X ms
15         }
16
17 }
```

Figure 12: Polling example

```
// INTERRUPT EXAMPLE (ATmega162)
// 1 button connected to PORTE PIN0 (INT2)

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

volatile uint8_t BUTTON_PRESSED = 0;

int main() {  // Button input  DDRE &= ~(1<<PE0);

     // Disable global interrupts
     cli();

     // Interrupt on rising edge PE0
     EMCUCR |= (1<<ISC2);
     // Enable interrupt on PE0
     GICR |= (1<<INT2);

    // Enable global interrupts
    sei();

    // Set sleep mode to power save
    set_sleep_mode(SLEEP_MODE_PWR_SAVE);

     while(1){  if(BUTTON_PRESSED){   // Respond to the button press
     BUTTON_PRESSED = 0;     }  // Good job! Now you can sleep =)
     sleep_enable(); }

}

    // Asynch. Interrupt from rising edge on PE0
ISR(INT2_vect) {
    BUTTON_PRESSED = 1;  // Wake up the CPU!
}
```

Figure 13: AVR Interrupt example

2.2.6 Struct and union

Although C is not an object oriented language, it is still possible to write well-structured code. Structs enable grouping of variables together as a single entity in an object oriented fashion, but it cannot include associated methods like in a true object oriented language. For example, a struct can group all variables needed for sending a CAN message:

```
    typedef struct {
        unsigned short id;
        unsigned char length;
        char data[8];
    } can_message;
```

Figure 14: Struct

The `typedef` statement lets you refer to the struct as a data type subsequently in the program. You can also use unions to access variables as different types, with possibly different sizes:

Here, the 8 bytes of data can also be accessed as two long integers called `\positions"`, (long integers are 4 bytes long in 8-bit architectures).

```
typedef struct {
    unsigned short id;
    unsigned char length;
    union {
        char data[8];
        long positions[2];
    };
} can_message
```

Figure 15: Struct with unions

2.2.7 Useful libraries

File name	Description
`avr/io.h`	AVR device-specific IO definitions
`avr/interrupt.h`	Interrupts
`util/delay.h`	Convenience functions for busy-wait delay loops (AVR)
`stdint.h`	Standard integer types
`sam.h // sam3x8e.h`	SAM setup and register declarations

Table 3: Useful libraries

It is recommended to check the AVR Libc documentation and see if there are functions in these libraries you may find useful for your application. For the SAM microcontroller you might find it useful to dive into the code of ASF (Atmel Start Framework) functions. However, we will not use these functions directly. Note that the delay functions for AVR require the variable `F_CPU` to be set to the current clock frequency. This can be configured in Atmel Studio, `Project` > `Configuration Options` > `Frequency`, or by writing the lines of code in Figure 16.

```
//NOTE: F_CPU must be defined before including delay in every file, not just in e.g. main!
#define F_CPU 16000000 // clock frequency in Hz
#include "util/delay."h
```

Figure 16: Include delay

2.2.8 Software debugging with Microchip Studio

Microchip Studio gives the opportunity to step through your code, read and modify most I/O registers on the fly, read variables, insert breakpoints and so on. Debugging directly on the AVR/SAM microcontroller chips is possible by using the Microchip Studio in conjunction with the Atmel-ICE. The AVR Tools User Guide gives an in-depth description of the debugger and several useful tutorials can be found on Youtube.
It is usually recommended to disable compiler optimization when stepping through code (`Project` > `Configuration Options` > `Optimization: -O0`). If not, the code execution order might not be as expected. If `_delay_ms()` is used, though, optimization is needed. Comment the delays out, or use optimization level 1. Keep optimization on during normal operation.

2.2.9 Software debugging with GDB

GDB (GNU Debugger) is a popular debugger made for the GNU project with many useful features for debugging. The most relevant for this project is adding `breakpoints` that stop the program

at specific lines of code. You can also make the debugger stop at given conditions such as `i > 10`. Information and manuals for the GNU Debugger can be found at `https://www.gnu.org/software/gdb/documentation/`.
You can flash the program and start the debugger with the command `make debug`. Note that the debugger does not connect to the microcontroller directly. Instead it uses avarice(AVR) or openOCD(ARM) to connect to the Atmel ICE. This will open in another terminal on `make debug` and must not be closed or tampered with while debugging is running. The debugger communicates with avarice/openOCD over TCP, making it possible to have the Atmel Ice connected to one computer and running the debugger on another.

Command	**Description**
`break 6`	Stop on line 6
`break 6 if i > 10`	Stop on line 6 if i > 10
`break _delay_ms`	Stop when _delay_ms is called
`watch i`	Break when i is changed
`continue`	Continue running the program
`print i`	Print the value of i
`info break`	Show information on breakpoints
`condition 2 i > 20`	Change the conditions for breakpoint 2, only stop when i >20
`disable 2`	Disable breakpoint 2
`delete 2`	Delete breakpoint 2
`set (i=3)`	Set variable i to 3
`step`	Step forward one line, step into function calls if any
`next`	Step forward one line, run function calls immediately
`run`	Run the program

Table 4: Some useful GDB commands

2.3 Tools and software

To develop software for the MCUs, you are free to use the development environment and tools of your own choice. Microchip Studio (Windows only) together with AVR-GCC is an good option because of its excellent debugging facilities and integration with compiler and programming tools. Alternatively, AVR-GCC together with any text editor in Windows, Linux or OS X can be used. Visual Studio Code is a popular free editor with good programmer tools and embedded Git. It can also be set up with the extension Live Share to support real-time collaborative development on the same project.
In addition, you will need terminal software to communicate with the MCUs over RS-232.

2.3.1 Microchip studio 7

Microchip Studio is built specifically for AVR/SAM microcontrollers and together with WinAVR and the Atmel-ICE, Microchip Studio constitutes a complete integrated development environment for the AVR/SAM microcontrollers. Spending some time exploring the various features that Microchip Studio provides and viewing the video tutorials provided on Youtube are definitely worthwhile. The debugging feature is of special interest. Using Atmel-ICE, one can show the values of, and modify, every I/O register in the microcontroller while running software. Microchip Studio can also simulate all microcontroller devices in the AVR family.
Microchip Studio 7 and updates can be found at `https://www.microchip.com/mplab/avr-support/atmel-studio-7`.

2.3.2 Programming without Microchip Studio

Microchip Studio is not available for Linux and OS X users. In Linux, AVRDUDE can be used for programming the AVR MCUs (transferring the binary program to the microcontroller). For the SAM MCUs OpenOCD can be used. The necessary configuration files (including Makefiles) are available on Blackboard. Make sure to update the Makefile with your own filenames. Transfer the program to the MCU with the command `make flash` in the terminal.

If `avr-gcc`, `avr-libc` or `avrdude` is not installed at Sanntidssalen, you can do this with the command `sudo apt-get install gcc-avr avr-libc avrdude`. The toolchain for node to can be set up with the following: **Toolchain for linux**

1. Call `sudo apt install gcc-arm-none-eabi` to install the compiler
2. Install prerequisites for a newer version of openocd with `sudo apt install libusb-dev libusb-1.0-0-dev libusb-1.0-0`
3. Open-jtag also requires `sudo apt install libhidapi-dev libftdi-dev libftdi1-dev`
4. Download openocd `git clone https://git.code.sf.net/p/openocd/code openocd`
5. Go into the folder `cd openocd`
6. Create the configure script with `./bootstrap`
7. Create and go into a directory for the build `mkdir build; cd build`
8. Configure the build `../configure --enable-cmsis-dap --enable-openjtag --prefix=/opt/openocd`
9. Make and install with `make`, then `make install`

2.3.3 Version control

Files and catalogs stored on the computers in the lab may be deleted at any time due to maintenance work or by other users. **You cannot rely on data being stored securely in this lab, so you should store files on portable memory or an external server.** Your home directory can be used for storage.
Using a version control system is highly recommended. However this is closer to an addition than a substitution for proper file storage. You should have a safe working storage where you do the work. After a feature or logic change is done you should upload the change into your version control system of choice. This will also give you the opportunity to restore older versions of your software when needed. The Orakel service (helpdesk) can provide your group with storage area. A powerful and a highly used version control system for software is GIT. For a practical introduction go to `https://try.github.io`, and for more info on how to collaborate using GIT check out the "Collaborating" part of `https://www.atlassian.com/git/tutorials/syncing`. You can also use GitLab with your NTNU username and password here (`https://gitlab.stud.iie.ntnu.no/users/sign_in`). Another popular version control system, widely used in multiple industries, is SVN `http://subversion.apache.org/`. Many other alternatives exist as well.

2.3.4 Terminal

A PC terminal application is needed when communicating with your embedded system over RS-232. Putty or Termite is recommended, but other terminals as Br@y++'s terminal can also be used. More information can be found at `https://sites.google.com/site/terminalbpp/`.

In Linux, there are also many alternatives available, such as 'screen', 'minicom', 'picocom' or putty.

3 Exercises

3.1 Initial assembly of microcontroller and RS-232

Three things are needed to set up the microcontroller unit (MCU) with basic functionality: a stable voltage supply, a clock signal and a reset circuit. We will also make arrangements for connecting the MCU to the PC, using JTAG for debugging and programming (uploading of binary program code to the MCU's internal flash memory).

RS-232 is a serial protocol that makes it possible for the microcontroller to send and receive data to and from the PC terminal. This will be a very useful feature during the development and debugging process. Moreover, by linking the `printf` function to your serial driver, you can conveniently let the MCU display text and other information on the PC terminal using the standard C print functions.

3.1.1 Creating a new project in Microchip Studio

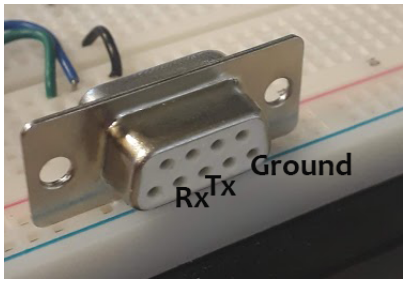
After opening Microchip Studio, a welcome dialog will appear. Click "New Project", choose "GCC C Executable project", fill out the project name and choose a location. In the next window, choose the microcontroller device you are going to write software for (ATmega162). After pressing "Finish", the project will be created.

3.1.2 Creating a new project in Linux

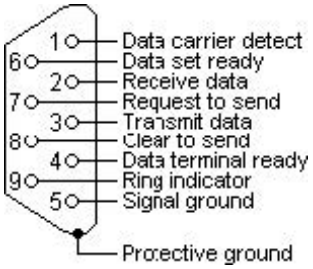
Chose your favourite text editor and create a folder. Add the Makefile found on Blackboard to this folder. To flash the code you write into the ATmega162, open a terminal in this folder and type `make flash`. It may be necessary to add `sudo` to the front of the command.

3.1.3 RS-232

RS-232 is a serial communication interface standard that is widely used in embedded systems. It has served the industry for a very long time, and because of its simplicity and abundance, it still has widespread use today. In this project, we will just need three pins of the RS-232 interface, namely RX, TX and signal ground, as shown in Figure 17. Be careful to connect the wires correctly to the female RS-232 connector on the breadboard.



(a) Female RS-232 connector



(b) Male RS-232 connector

Figure 17

DB9 pin	Name	Description
1	DCD	Data carrier detect
2	RX	Recieve data
3	TX	Transmit data
4	DTR	Data terminal ready
5	SGND	Signal ground
6	DSR	Data set ready
7	RTS	Request to send
8	CTS	Clear to send
9	RI	Ring indicator

Table 5: Pinout for RS-232

3.1.4 Exercise

1. Take up the breadboard, insert the voltage regulator LM7805, and connect it to the power supply and capacitors as described in its datasheet. You may use a 1 µF capacitor on the input instead of the recommended 0.33 µF. Use a voltage supply of $8 - 12V$ and verify using a multimeter that the output voltage is a stable $5V$. Turn off the power supply.

2. Insert the AVR ATmega162 MCU and connect its power pins to the output of the regulator. Remember to use decoupling capacitors as discussed earlier.

3. Connect a reset circuit to the MCU's reset pin as described in Atmel's Application Note AVR042. The diodes can be omitted. Connect the push button between "External Reset" and ground, and use a 0.1µF capacitor for the RC circuit.

4. Connect the crystal oscillator (4.9152 MHz) close to the XTAL pins of the MCU as described in the ATmega162 datasheet. Use 22 pF load capacitors between each leg of the crystal and ground.

5. Connect the JTAG interface for Atmel-ICE to the breadboard. See the Atmel-ICE and ATmega162 manuals for which pins to connect.

6. Create a simple test program and upload it to the MCU using the Atmel-ICE. For instance, the test program could make a square wave signal to one of the output pins by toggling a digital output. This could then be verified by connecting an oscilloscope probe to the relevant pin.

You have now connected the basic components needed to get the MCU up and running. The following steps will enable serial communication over RS-232.

7. Insert the MAX233 IC and connect it as described in its datasheet. TXD and RXD pins on the ATmega162 should be connected to $\mathrm{T1_{IN}}$ and $\mathrm{R1_{OUT}}$ on the MAX233, respectively. Correspondingly, $\mathrm{T1_{OUT}}$ and $\mathrm{R1_{IN}}$ are to be connected to the serial line to the computer by using a connector and a DB9 M/F cable.

8. Program a driver for ATmega162's UART interface. This driver should contain functions for sending and receiving data to and from the serial interface. Think about how to handle the following situations:

   (a) The application tries to send a new character while the UART is busy transmitting the previous one

   (b) The application wants to be notified when a new character is received

9. Create a test program to verify the driver and connections works correctly. The program should both transmit and receive data over the serial line. Use the terminal program on the PC when testing.

10. To link the `printf` function to the UART driver you only need to make a function that transmits one character to the UART, and call the function `fdevopen(transmit function, receive function)`. For more information about this function, see the section about the Standard I/O module (`stdio.h`) in AVR Libc's user manual at `http://www.nongnu.org/avr-libc/user-manual/group__avr__stdio.html`.
11. Verify that you now can send text and values of variables by using the standard printf function.
12. Set up version control for your code base, see 2.3.3

3.1.5 Tips

- Make sure you **always turn off the power supply before connecting components!**
- Use the multimeter to verify that all inputs (VCC, GND and RESET) have correct voltage.
- Use the oscilloscope to check the clock signal at XTAL2.
- Have a look at section 2.1.10 Hardware debugging.
- Before programming the MCU, verify its fuse bits. This check can be executed from Microchip Studio. With the Atmel-ICE connected, click Tools > Device programming > Choose Atmel-ICE, the right processor, JTAG interface, press "Apply" and verify that the fuse bits are as follows:
  - Brown-out detection disabled.
  - On-chip debug enabled.
  - JTAG interface enabled.
  - Serial program downloading.
  - Ext. Crystal osc: Frequency 3-8 MHz start-up time: 16K CK + 65 ms.
  - Do not divide clock signal by 8 (CLKDIV8 disabled)
- **Fuse bits in Linux:** run the command `avrdude -U lfuse:w:0xFD:m -U hfuse:w:0x19:m -U efuse:w:0xFF:m -U lock:w:0xFF:m` in a terminal to configure the avrdude extension to set the correct fuse bits. Source: eleccelerator. Use this permalink with the presumed correct settings or this generic link and configure according to the previous tip in order to calculate the correct `avrdude` command. Fuses only need setting once, if any. If you successfully solve this first lab exercise, you can rest assured that fuses are **not** the problem in future exercises.
- See section 2.2.3 for how to set pins on the ATmega162.
- Have a close look on the example circuit in the MAX233 datasheet. Pay particular notice to the pins that should be interconnected, and the decoupling capacitor between voltage supply and ground.
- Most of the serial cables in the real-time lab are "crossed", that is, 'transmit' in one end is connected to 'receive' in the other. Use a multimeter to verify this.
- For now, keep the driver interface simple, and postpone further design decisions until you know the requirements of the application.
- `printf` is a large and computationally expensive function, and may alter the behaviour of your programs because of the delays introduced.
- C is a minimalistic language in regards to safety mechanisms, and no static or dynamic checks are done to prevent or detect stack overflow. This means that if you use a high amount (but less than 100%) of memory due to static allocation you might corrupt the memory due to the stack growing into the heap. There are generally three ways to solve this: stay well within reasonable limits in regards to memory usage, do static analysis or do runtime checks (or use a programming language which contains runtime checks). Keeping well within limits might be the most practical approach for our application, but feel free to use some dynamic/static checking if you find this rewarding.

3.2 Address decoding and external RAM

In this exercise you will be designing a memory map and implementing an address decoder that will allow connection of memory and I/O devices using the MCU's external parallel bus interface (ATmega162 supports a parallel interface by way of its multiplexed address/data bus). Access of external memory and I/O devices can then be carried out simply by using standard read and write instructions to certain addresses in the MCU's memory space. You will also connect the first external unit to this system – a 64K SRAM IC (we will only use some of its memory space).

3.2.1 External memory

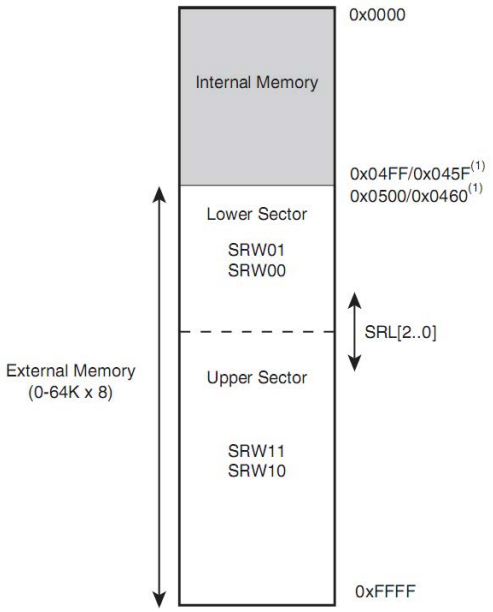


Figure 18: External memory [ATmega162 datasheet]

The organization of the ATmega162 data address space is shown in Figure 18. Note the area labeled "External Memory", which corresponds to the address space between 0x0500 and 0xFFFF. When the microcontroller is configured to use external memory, the pins of PORTA and PORTC and some of the pins of PORTD forms the data/address bus and will be reserved for transferring data between the microcontroller and the external memory IC (or other I/O devices connected to the memory bus):

- PA0 – PA7 will form a multiplexed address/data bus that alternates between transferring bit 0 – 7 of the 16-bit memory address, and the eight data bits.
- PC0 – PC7 will transfer bit 8-15 of the 16-bit memory address.
- PD6 is used as the WR signal (write control strobe – falling edge indicates to external the device that the processor has output valid data on the data bus. Rising edge indicates that data should be latched into the external device).
- PD7 is used as the RD signal (read control strobe – falling edge indicates that the external device should output valid data on the data bus. Rising edge indicates that the processor will latch in data from the data bus).
- PE1 is used as ALE (address latch enable – rising edge indicates start of a bus cycle and opens the latch for the low-byte address to be written to its output. Falling edge locks the latch and indicates that a valid 16-bit address is output to the address bus).

This interface is further described in the section "External Memory Interface" in the ATmega162 datasheet.

This project uses the JTAG interface which is also located on PORTC, that is, the four most significant bits on the address bus. Fortunately, ATmega162 can mask out bits from the address bus making them available for the JTAG interface instead of the address bus. The addresses are still valid, but the pins will not react when writing to the corresponding addresses. This behaviour can be exploited when organising the address space. See the section "XMEM Register Description" for information about how to accomplish this.

The microcontroller uses 16 bit addresses which makes an address space of $2^{16} = 65536 = 64$k different addresses. As the 4 MSBs are occupied by the JTAG interface we get a resulting address space of $2^{12} = 4096 = 4$k.

AVR-GCC can also be configured to make use of the extra data memory capacity, see AVR Libc's user manual to find out how it is done. Anyhow, this should not be necessary as the ATmega162 has sufficient amounts of data memory.

3.2.2 Memory mapping and address decoding

From the microcontroller's point of view, the address space represents a continuous logical block of data memory. Physically, this may not be entirely true as many different devices (memory and I/O) can be connected to the MCU via this interface as long as they have a parallel interface and can read and/or write to the data bus.

We are going to utilize this feature by connecting two different devices to the external bus: an SRAM and an A/D converter. When there is more than one unit on the bus, an address decoder will be necessary in order to choose which unit to be activated based on the address output on the address bus. Usually, the address decoder is realized by means of simple digital logic gates, but sometimes more complicated logical functions are required. The chapter "The AVR Microcontrollers" in the book "Designing Embedded Hardware" explains the AVR memory bus and the concept of address decoding in detail.

3.2.3 NAND gates

A NAND-gate performs the logical operation NOT AND. These gates can be used to produce any of the other logical operators, NOT, OR and AND. Because of this, any boolean expression can be implemented with only NAND-gates. This can be done by applying De Morgan's laws to the boolean expression or using an online boolean calculator, for instance Wolfram|Alpha (`https://www.wolframalpha.com/examples/mathematics/logic-and-set-theory/boolean-algebra/`).

3.2.4 Address latch

Because the pins AD0 – AD7 are used for both addresses and data (multiplexed address and data bus) we need a way to "hold" the addresses while the data is output to the same pins. To accomplish this, an 8-bit D-latch, 74ALS573, is used. The microcontroller provides a signal for opening the address latch (ALE) while the low-byte address is output on the bus, and closing it when the data is output on the bus. The latch together with the ALE signal will then effectively demultiplex the address and data signals, providing an 8-bit data bus and a 16 bit address bus as required.

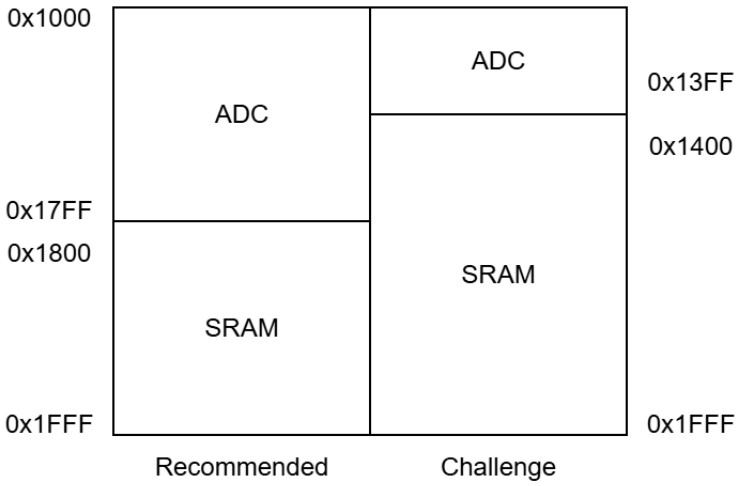


Figure 19: recommended memory mapping

3.2.5 Suggested memory mapping for this project and code example

As shown in Figure 18, addresses up to 0x04FF are internal and cannot be addressed externally. The external addresses should thus start on 0x0500, but it is recommended to start on 0x1000. Using the 12 address bits remaining after masking out pins for the JTAG interface, this will give an external address range of 0x000 to 0xFFF, because the 4 MSBs are masked out and will be ignored. This means that we can use all 12 bits of external address bus. Starting on 0x0500 would be more complicated, convert the numbers to binary to see why.

To make address decoding simple, we can locate the ADC at addresses 0x1000-0x17FF, and the SRAM at 0x1800-0x1FFF, as shown in Figure 19 . This will allow us to use only a single bit of the address to perform the address decoding. We will only use 0x800 $\times$ 8 bits = 16kbit = 2KiB of the SRAM, and at the same time we will "waste" much of the address space on the ADC. This is the disadvantage of partial address decoding, but it gives very simple address decoding logic. We could of course have included additional address lines in the decoder to make more efficient use of the address space.

For instance, when writing 0x51 to address 0x183E, the AVR detects that this address belongs to the external memory. As the 4 MSBs are masked out, the address 0x83E will be put on the external address bus, and the ALE signal goes high at the same time. The address decoder detects that this address belongs to the SRAM, and thus enables the Chip Select signal for this IC, and disables all other connected peripherals. The ALE signal then goes low, and the address latch "remembers" the 8 LSBs of the address. Then, the value 0x51 is placed on the data bus, and WR goes low. In the end, WR goes high, and the write operation is completed.

3.2.6 Accessing memory

Accessing the external memory bus is simple. The following test program checks all bits (if SRAM = 2kB) in the entire SRAM for write/read errors:

```c
#include <stdlib.h>
void SRAM_test(void)
{
    volatile char *ext_ram = (char *) 0x1800; // Start address for the SRAM
    uint16_t ext_ram_size = 0x800;
    uint16_t write_errors = 0;
    uint16_t retrieval_errors = 0;
    printf("Starting SRAM test...\n");
    // rand() stores some internal state, so calling this function in a loop will
    // yield different seeds each time (unless srand() is called before this function)
    uint16_t seed = rand();
    // Write phase: Immediately check that the correct value was stored
    srand(seed);
    for (uint16_t i = 0; i < ext_ram_size; i++) {
        uint8_t some_value = rand();
        ext_ram[i] = some_value;
        uint8_t retreived_value = ext_ram[i];
        if (retreived_value != some_value) {
            printf("Write phase error: ext_ram[%4d] = %02X (should be %02X)\n", i, retreived_value, some_value);
            write_errors++;
        }
    }
    // Retrieval phase: Check that no values were changed during or after the write phase
    srand(seed);
    // reset the PRNG to the state it had before the write phase
    for (uint16_t i = 0; i < ext_ram_size; i++) {
        uint8_t some_value = rand();
        uint8_t retreived_value = ext_ram[i];
        if (retreived_value != some_value) {
            printf("Retrieval phase error: ext_ram[%4d] = %02X (should be %02X)\n", i, retreived_value, some_value);
            retrieval_errors++;
        }
    }
    printf("SRAM test completed with \n%4d errors in write phase and \n%4d errors in retrieval phase\n\n", write_errors, retrieval_errors);
}
```

Figure 20: SRAM test, the code is available under Misc. Resources in Blackboard

.

3.2.7 Exercise

1. Connect the latch following the descriptions in the datasheets for the latch and the microcontroller. This setup can be tested by connecting LEDs to the output of the latch. Writing to different addresses in the external address space should make the LEDs blink accordingly.

2. Connect the SRAM IC and verify that it is working properly by running the given test program. The code is available under `Lab support data` → `Misc. Resources` in Blackboard. Do not worry if there are a small number of errors. The breadboard and cabling are prone to noise during both read and write cycles.

3. Design an address decoder which makes it possible to communicate with the following units:

   - ADC
   - 16 kb SRAM (needs at least 2048 addresses (8 bit × 2048 addresses = 16384 bits = 16kbits = 2 KiB))

4. Implement the address decoder by connecting the correct pins of the address bus to the chip select pins of the SRAM and ADC chips.

Implement the address decoder using the NAND gates. You may not need all of them. Connect it, and verify that the correct chip select signals are generated by running a test program and measuring the resulting output signals.

Voluntary challenge: 3KiB SRAM
For a more efficient use of the address space, design your address decoder using the NAND gate IC supplied with your kit. Using two bits of the address bus instead of one, you can locate the ADC at 0x1000-0x13FF and the SRAM at 0x1400-0x1FFF. This will give you 3KiB of SRAM instead of 2KiB, which might come in handy when you integrate the OLED display in a later exercise. As mentioned in the introduction of this exercise, this will require some careful planning and boolean algebra.

3.2.8 Tips

- Remember to enable the external memory interface by setting the SRE bit in MCUCR.
- A low-pass filter on the ALE signal might be necessary for stable operation of the address latch. The schematic for STK501 gives an example.
- If the RAM test often fails for certain combinations of addresses and data, for example when writing 0x00 to addresses ending in 0xFF, this can be an indication of power supply problems. Ensure decoupling capacitors are in place, and remember star-point connection for supply voltage and ground. You might also need to experiment with different centers for the stars.
- The SRAM IC has two chip enable signals. Choose a suitable signal for the address decoder, but do not let the other signal float!
- The address bits from the microcontroller do not necessarily have to be connected to the corresponding pins on the SRAM IC – addresses will only be placed in different physical positions in the SRAM. Thus, choose an ordering that will lead to neat and tidy wiring.
- AVR-GCC can be configured to use the extra memory capacity provided by the SRAM. See the AVR Libc user manual to find out how. It is not recommended to store vital data in the SRAM in this setup due to the noise susceptibility.

3.3 A/D converting and joystick input

In this exercise you will connect an Analog to Digital Converter (ADC) with external memory interface to your system, which will make it possible to read input signals from the analog joystick on the User-I/O board (see board manual for further details).

3.3.1 Joystick

The joystick has two internal variable resistors (potmeters), which will vary according to the joystick's position along its two axes. It also has a button that connects its output to ground when pressed, as shown in Figure 21.

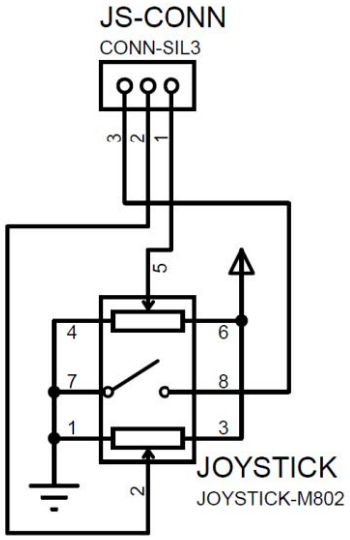


Figure 21: Joystick and internal schematics

3.3.2 Touch

The board has a touch slider and a touch pad. The AVR128DA64 uses Atmel's free QTouch library to read and process the touch signals. It then does some internal processing before outputting the signals as analog signals through a DAC.

3.3.3 MAX156

The MAX156 is an ADC with a parallel interface, which can easily be connected to ATmega162's memory bus in the same way as the SRAM. It does not need any address lines as it will only receive data on the data bus, but it will need a chip select signal derived from the address bus using the address decoder implemented in the previous exercise. The MAX156 does not have an internal clock, and requires an external clock signal to work properly.

Read the datasheet to find out how data is to be read. Note that **timing is important**.

3.3.4 Exercise

1. Connect the ADC, so that it can be accessed by the external memory interface of ATmega162. Verify that the chip select signal is activated when its address space is accessed.

2. Power on the User-I/O board and check that the positions of sliders and buttons are shown on the OLED.
3. Connect the joystick to the ADC, and verify that the ADC can read the joystick position.
4. Use an oscilloscope to find the pins that output the touch signals.
5. Connect the analog signals from the User-I/O board to the ADC.
6. Write an expression explaining the relationship between joystick angle and voltage.
7. Make a software driver that can return the joystick and slider positions. They should be roughly equal to (or at least consistent with) the values displayed on the OLED.

3.3.5 Tips

- If the positions and buttons are not shown on the OLED, hand your User-I/O board in to one of the course administrators so that they can reflash it.
- The potentiometers inside the joystick work as voltage dividers.
- Use the oscilloscope to find the relationship between the joystick position and its internal resistance. Find the output voltage as a function of the internal joystick resistance.
- To accommodate for later exercises, you are advised to make one function that will return the joystick's analog position (e. g. X: 83%, Y: -21%), and one function that will return the joystick's direction. The first one can be accomplished by returning a struct containing the position value in both directions, and the second by using thresholds and returning an enumerated value (LEFT, RIGHT, UP, DOWN, NEUTRAL).
- It is recommended to implement auto-calibration in software.
- A PWM signal from the AtMega162 can be used as the ADCs external clock. *Use PD4 or PD5 to avoid conflicts for later exercises.*
- By connecting the `MODE` pin of the ADC to 0/1 (table 4 of the datasheet), many of the settings in the configuration register (Table 1) will be ignored. This also applies to the D7/$\overline{ALL}$ bit (= 0), making conversion results to be read with consecutive $\overline{RD}$ pulses.
- The internal voltage reference for the MAX156CNG+ is set for 2.5 V. How does this fit with the voltage output of the joystick and touch surfaces? Have a look at **External reference** in the datasheet.
- When writing configuration to the ADC, the compiler might optimize away a subsequent read from the same location, because it expects that the value will not change. To tell the compiler that the content of the memory location might be changed without the compiler's knowledge, use the qualifier `volatile` when defining the pointer to the memory-mapped ADC.
- Use your oscilloscope to investigate the timing properties of the signals reaching the ADC, and ensure that they satisfy the requirements given in the datasheet.

3.4 OLED display and SPI part 1, IO buttons

In this exercise you will configure the MCU's SPI bus interface to communicate with the IO board in order to display information on the OLED and receive input from the buttons. This includes making a user interface for the ping-pong game, with elements such as the game score and options to start a new game. In the next exercise, you will use the same SPI bus to enable CAN bus communication between Node 1 and Node 2 (i.e. ATmega162 and Arduino), which many students find to be a tough exercise. It is therefore quite important that you solve this exercise carefully and test your solution rigorously in order to make debugging easier in the coming weeks. If you haven't already, this is a great time to make friends with your oscilloscope. One important aspect is to design your SPI driver such that it may support communication with at least two slave units. This should include considerations about bandwith and prevention of bus contention.

3.4.1 MCOT128064H1V-WM

The display to be used is a monochrome 128×64 dot matrix display module. It is driven by the IO board microcontroller, which in turn communicates with the display's SSD1309 controller by SPI. The SSD1309 has its own set of commands which may be found in the datasheets on Blackboard. The OLED display datasheet has an example of basic configuration/initialisation, while the SSD1309 datasheet contains the full command set.

The display is hardware configured to use the SPI interface (BSn = 00), and connected to appropriate pins on the IO board's MCU for a 4-wire SPI connection. Your SPI driver will need to be able to send commands to the IO board MCU such that it uses the D/!C signal correctly.

The controller has 128×64 bits = 1kB RAM for the data, divided into eight pages. When one data byte is written into the RAM, all the rows of the current column are filled. Each bit represents one pixel so with this 8-bit architecture you can minimum write 8 pixels at a time.

The display is an excellent opportunity for creative features like, e.g. making animation, own characters, dual buffering, drawing functions (circles, cubes, ...) etc. Particularly industrious students have even implemented vector graphics.

3.4.2 SPI communication

SPI is a very popular serial, synchronous, full duplex master/slave bus for inter-IC communication. The bus itself consists of three signals shared between all units connected to the bus: MISO (master in, slave out), MOSI (master out, slave in) and SCK (serial clock). In addition, each unit has an SS (slave select) signal and only the slave with this signal active will participate in the transmission.

An SPI transmission using the AVR microcontroller goes on like this:

1. MCU selects one of the slaves by setting its corresponding SS signal to low.
2. The MCU starts the clock signal SCK when the program writes to the SPI data register (SPDR; given that SPI module is enabled). For each clock period one bit of the SPDR will be shifted from the master to the slave (MOSI), and one bit from the slave to the master (MISO).
3. When transmission completes after eight clock periods, SS will be pulled high to indicate that the operation has completed and release the slave. The datasheets of the MCU and ICs that uses SPI provide a good description of the bus.

Figure 22 illustrates how the SPI bus should be set up in this system. Notice that the bus from the IO board MCU to the OLED display is effectively simplex, as the MISO line has been replaced with a data/command line which is used to instruct the OLED about whether the incoming byte

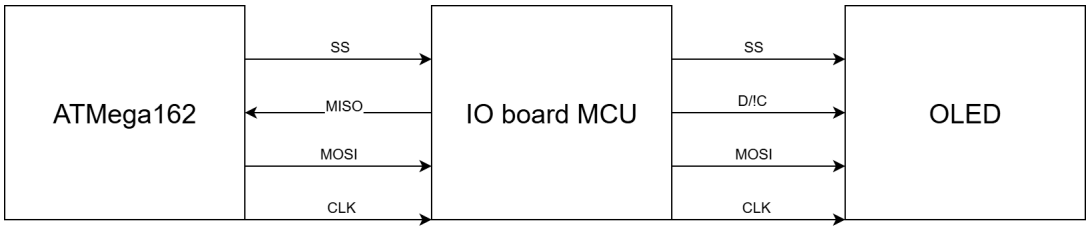


Figure 22: SPI buses in this exercise

represents a command or data. Your task will be to make the IO board MCU "forward" messages from your ATmega to the OLED display.

3.4.3 Modularisation

This exercise is complex and relies on the correct functioning of several modules. You are strongly advised to partition your drivers into several layers. This will make your code easier to maintain and understand, and, importantly, it opens the possibility of reuse. In the next exercise, you will appreciate being able to reuse the low level SPI driver for communication with the CAN controller. When integrating Node 2 in a later exercise, you may also find it useful to reuse some of the higher level CAN code. Modularisation as a design pattern is crucial to efficient development in any software project, and you should use this exercise to familiarise yourselves with the concept if you haven't already. Modularization is further explored in section 2.2.1.

3.4.4 Exercise

1. Place the IO board connection adapter in a convenient location on your bread board, oriented such that it may be neatly connected to the MCU's SPI pins.
2. Connect the SPI pins, and choose suitable pins for the slave select signals.
3. Write an SPI driver for your ATmega. It should have the following capabilities:
   - Select slave n
   - Write byte
   - Read byte
   - Also convenient: Read/write n bytes
4. Write an OLED display interface on top of the SPI driver:
   - Initialisation
   - Go to line
   - Go to column
   - Printf (or at least your own simplified version)
   - Funny graphics?
   - Remember that the SPI driver itself will also be used for CAN bus
5. You have access to several fonts for use with the display, but they are too large to be stored in data memory. Store them in PROGMEM and use them from there.
6. Test your driver/interface (it's scopin' time!)
7. Connect the IO board, and try to send data to the OLED display.
8. Make a framework for a user interface that can be navigated in using the joystick. As a minimum, it should be able to let the user navigate up and down in a menu consisting of a list of strings, and return the menu position when the joystick button is clicked. Also, think about how you would implement sub-menus using this framework.

9. Expand your interface to be able to receive data from the buttons on the IO board. You could for instance toggle an LED on your breadboard.
10. Optional: Create your own characters.
11. Optional: Store a local copy of the display data in the SRAM, write to that instead, and create a function that periodically (e.g. 60 Hz) writes the whole display with the data from the RAM.
12. Optional advanced: Create drawing functions like `oled_line(x0,y0,x1,y1)`, `oled_circle(x,y,r)` etc.

3.4.5 Tips

- See table 7-1 in the OLED advance datasheet (SSD1309) to get an overview over the control signals.
- Read the two OLED datasheets. There you will find an example init function, how the memory is mapped to the different pixels etc. Everything you need to create stunning graphic actually!
- You can find fonts on Blackboard.
- Recommended functions: oled_init(), oled_reset(), oled_home(), oled_goto_line(line), oled_goto_column(column), oled_clear_line(line), oled_pos(row,column), oled_print(char*);
- AVR PROGMEM tutorial: `https://www.nongnu.org/avr-libc/user-manual/pgmspace.html`
- If you print the display from the SRAM, dual frame buffering will help you avoid flickering, tearing and other artifacts. Wikipedia: `http://en.wikipedia.org/wiki/Multiple_buffering`
- If you wish to use printf both for RS232 and OLED you can set them up as a stdio stream. Check out FDEV_SETUP_STREAM in the stdio avr-libc documentation. After you've created the stream simply point stdout to your stream for printf to work.
- Compared to the previous exercise, the OLED has a bigger command set. Be sure to understand how the display works, both with respect to hardware and software before you start.
- Remember that SPI is a full duplex bus, that is, communication goes in both directions at the same time. Thus, there are no pure read or write operations. The ATmega162 will start transmission when the SPDR register is written to. When the transmission completes you will find the received byte in the same register.

3.5 SPI part 2 and CAN controller

This exercise will make the basis for enabling CAN network communication in our system. It will be realized using a standalone CAN controller with an SPI interface to the MCU and a CAN transceiver for driving the bus lines. The CAN exercise is quite demanding, and students often need extra time to complete a working solution. It is therefore very important that you read the documentation thoroughly and set aside enough time to do each part properly. First, you will adapt the SPI driver you made in the previous exercise to enable communication between the MCU and the CAN controller. This should be fairly straightforward if you separated the SPI and OLED specific code properly. Then, you will make a driver which lets you access the CAN controller's control registers. In this exercise, the CAN controller will be tested using it in "loopback mode" where it sends data to itself. In the next exercise, you will enable communication between Node 1 and Node 2 as described in chapter 1.1.

3.5.1 CAN bus

CAN (Controller Area Network) is a fault tolerant field bus with excellent real-time characteristics. Being a multi-master bus, every node connected to the bus can initiate sending of CAN messages and all nodes will receive all messages at the same time. It is not possible to address a message to a specific node, but messages have IDs that the CAN controller can use to decide whether a message should be accepted or ignored.

The message ID also serves as a priority mechanism. If several nodes try to send a message simultaneously the message with the lowest ID will win the arbitration and proceed without interruption. Arbitration in CAN is more thoroughly described in the CAN 2.0B specification. Reading the application notes "AN713: CAN Basics" and "AN228: A CAN Physical Layer Discussion" is also recommended.

3.5.2 MCP2515

Node 1 uses the CAN controller MCP2515 which implements the data link layer and some of the physical layer of the CAN protocol specification, version 2.0B. Each node will be physically connected to the bus via a CAN transceiver (MCP2551/MCP2562).

The MCP2515 is a stand-alone CAN controller with an SPI interface. Actually, it is a customized microcontroller and thus it requires the same minimal configuration as an AVR microcontroller: a clock signal, a reset circuit and a stable power supply. The reset signal can be taken from the existing reset circuit.

There are several sources of interrupts in the CAN controller, and MCP2515 has a common interrupt output pin that should be connected to the ATmega162. After an interrupt is generated, ATmega162 should find out which interrupt that actually triggered by reading the MCP2515 register CANINTF.

3.5.3 SPI pt 2

By now you are already familiar with SPI. However, in the last exercise you only had two nodes to contend with, one slave and one master. In this exercise we're introducing a second slave, the CAN controller. As we are not daisy chaining the SPI nodes, the number of slaves is limited by the number of slave select signals we can output from the MCU. You were advised to set aside a suitable pin for this purpose in the previous exercise. In addition to pins, an important aspect of managing multiple nodes on a bus network is bandwidth: how much data can be sent at a given time, and how do we prevent one node from clogging the access of another (i.e. bus contention)? If you didn't do this during the last exercise, you should do the math to find out how often you can

send OLED, button and CAN data between the nodes on the SPI bus, and brainstorm methods to manage bus contention as preparation for this exercise.

3.5.4 Modularization pt 2

As in the previous exercise, this one consists of several parts which must work separately and together. This time, you should ensure that your CAN drivers are layered: a hardware layer riding on top of the SPI driver, and a messaging layer which should be independent of hardware. Reusability of the upper layer can save you some time when implementing CAN on node 2. An example of how the CAN driver could be organized is shown below:

Layer	Content
CAN communication	High level sending and receiving CAN messages
MCP2515 manipulation	Low level driver for setting up the CAN controller and accessing its control and status registers
SPI communication	SPI communication driver

Table 6: CAN modularization

The high level message sending interface will be used by your application, and in principle, it should be designed so generally that it is possible to use it on other CAN controllers without any modifications. The low level functions will implement the MCP2515 instruction set, described in chapter 12 of the CAN controller's datasheet. These will then use SPI to transfer data between the MCU and CAN controller. Modularization is further explored in section 2.2.1.

3.5.5 Exercise

1. Make the basic connections for the MCP2515, that is, clock signal, reset circuit and power supply.
2. Connect the MCP2515 to the ATmega162's SPI bus and one of its interrupt pins.
3. Create a driver for the controlling the MCP2515. Implement the instructions below, as described in MCP2515's datasheet chapter 12. The read instruction might be a good start as it is required for testing of other instructions.
    - Read
    - Write
    - Request to send
    - Read status
    - Bit modify
    - Reset
4. Create a driver for CAN communication using the MCP2515 driver. It should have the following features:
    - Initialization of the CAN controller. Use the loopback mode for now.
    - Send a message with a given ID and data (for instance, by passing a struct containing the ID, length and data to a send function)
    - Receive a message.
5. Test this driver by sending/receiving some messages (in loopback mode, you will receive what you sent immediately).

3.5.6 Tips

- Ensure that only one SPI slave is active at any time, otherwise they will try to drive the MISO line concurrently making a short circuit.
- Check your initialization code for the SPI driver. The ATmega162 datasheet contains examples worth reading.
- Use the oscilloscope to verify that the SPI signal lines work as expected.
- Bus contention: the ATmega has timer/counter modules which could be used to set the rate of information flow on both the SPI and CAN buses.

3.6 Getting started with ARM cortex-M3 and communication between nodes

This is the exercise where Node 2 will be introduced for the first time. As shown in section 1.1, it consists of the Arduino Due and a shield mounted on top of the Arduino.
NOTE: The Arduino Due uses 3.3V Transistor-Transistor Logic (TTL) levels. Do not connect 5V devices to it directly as this might damage the pins.

3.6.1 Arduino USB Communication

The Arduino Due allows us to use the USB connection as a USB to serial bridge via a separate ATmega chip onboard, ATMEGA16U2. The ATSAM3X8E is connected to the USB bridge through UART via PA8 and PA9. A simple UART and printf library to get you started is available on Blackboard.

3.6.2 Arduino Shield

The custom Arduino Shield sits on top of the Arduino Due providing the following:

- Two 4 mm banana plug receptacles for motor power input
- Two 2 mm banana plug receptacles for motor power output
- One 3x2 pin header for motor encoder input
- One 3x1 pin header for servo power/control output
- One 5V voltage regulator for servo power
- One 2x1 pin header for CAN wires
- One jumper for CAN terminal resistor
- One MCP2562 CAN transciever
- One Allegro A3959 motor driver

Please do not remove the shield.

A thorough description of the shield and its usage in this project may be found in the *TTK4155 Motor Shield* document.

3.6.3 ATSAM3X8E on Linux

For programming the ATSAM3X8E on Linux some additional files will be needed in addition to the Makefile. These files set up the microcontroller and specifies where in memory the program is to be flashed. The files can be found on Blackboard. Include the folder named SAM and the Makefile in the same folder as your C-code.

3.6.4 CAN bus bit timing

All nodes on a CAN bus must have the same bit timing in order to communicate. The CAN *bit time* (the time it takes to eject one bit on the bus) is made up of non-overlapping segments where each segment consist of a given number of *Time Quantas* (TQs). Normally the *CAN bit time* consist of four segments:

1. Synchronisation segment (t_{sync}): This segment is used to synchronise the nodes on the bus.
2. Propagation segment (t_{prop}): This segment compensates for physical delays between nodes.
3. Phase segment 1 and 2 ((t_{seg1}, t_{seg2}): These segments are used to compensate for edge phase errors on the bus. The sampling point is the point between t_{seg1} and t_{seg2}.

An illustration is shown in figure 23.

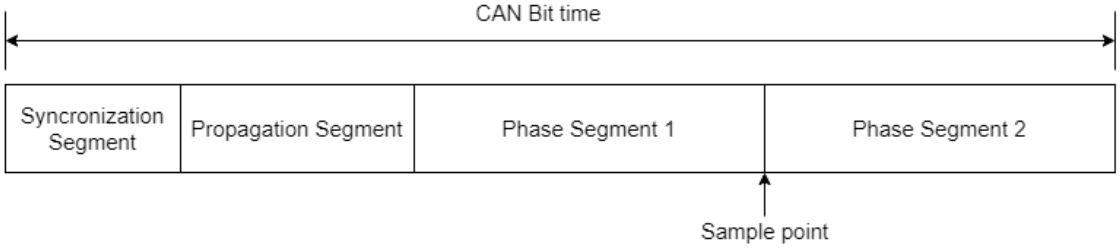


Figure 23: CAN Bit timing

The bit time may be configured by setting registers in the CAN controllers. Make sure that each segment consists of the same number of Time Quantas on every node and that the Time Quantas are equal. The length of a time Quanta can be set by adjusting the Baudrate according to the master oscillator clock frequency.

3.6.5 Exercise

1. Recommended but optional: Write a program which enables (i.e. sets high) the servo header signal pin, and verify your result with an oscilloscope. Think of it as a "hello world" for the Node 2 shield.
2. It is useful to implement serial communication also on Node 2. A simple library is available in Blackboard. Make sure to find/set the baudrate in `uart.c`.
3. Connect the CAN transceiver MCP2551 on Node 1 using the information provided in the datasheet. You can use the 22k resistor for slew-rate limiting.
4. Import the CAN library for node 2 to your project. The library is available in Blackboard. Feel free to adapt it to your needs as the project progresses, e.g. by implementing message ID masks, circular buffers etc.
5. Connect Node 1 and Node 2 together using the CAN bus, conforming to the "AN228: A CAN Physical Layer Discussion" document. **Note:** The `CAN_TERM` jumper on the Node 2 shield *must* be connected at this point.
6. Decide the CAN bus bit-timing by writing to the `CNFx` registers on MCP2515. Make sure to match the configuration in the `CAN_BR` register on ATSAM3X8E.
7. Test the system again, but now with the CAN controller of node 1 in normal mode. Node 2 should be able to reuse the upper level code generated in the previous exercise.
8. Make a joystick driver that can send joystick position from Node 1 to Node 2 over the CAN bus.

3.6.6 Tips

- For UART over USB to work, connect the micro USB-cable to the right USB-port (the one closest to the round power supply connector).
- Remember to enable peripherals in PMC.

- If the microcontroller keeps resetting, make sure to feed the *watchdog timer (WDT)* (See section 15 of the ATsam3X8E datasheet). You can also disable it by setting `WDT->WDT_MR` to `WDT_MR_WDDIS`.
- Sending CAN messages in normal mode will never succeed before there is at least one node that can acknowledge the transmission.
- Remember to terminate the bus in both ends – the shield has built-in termination that can be disabled by removing JP1.
- The order of writing to the CNFx registers in MCP2515 matters. Check that they are correctly configured after writing to all three.
- In this lab we will use JTAG, however the USB to serial bridge on the Arduino can be used to program it. The DTR (Data Terminal Ready) which signals a new connection in RS232, is connected to the Reset line of the ATsam3X8E. Therefore, the microcontroller is reset every time you connect to the COM port in Windows.

3.7 Controlling servo and IR

3.7.1 Pulse Width Modulation (PWM)

To generate analog voltages from a digital controller, PWM is often used. The principle involves high frequency pulsing of a digital output signal where the pulse width, or ON-time, can be manipulated. The ratio between the pulse width and signal period is called the duty cycle of the PWM signal. Passing the PWM signal through an lowpass filter will generate an analog signal that is proportional to the signal duty cycle. That is, if the voltage is 5 V while the signal is on and the PWM signal has a duty cycle of 20%, the average output signal will be 1 V.

3.7.2 Servo

The servo's position is controlled by means of a PWM signal. The signal period should be 20 ms and a pulse width of 1.5 ms controls the servo to its center position. Maximum deflection angles will be achieved by using pulses of 0.9 ms and 2.1 ms, respectively.

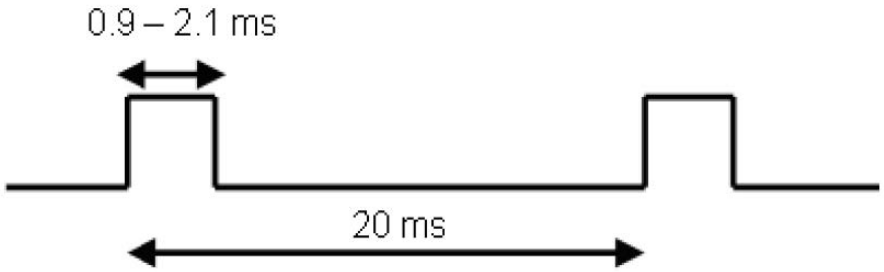


Figure 24: PWM for servo

Check that the signal is correct before connecting the servo! Otherwise you risk damaging the servo. Never use pulses outside the range 0.9 – 2.1 ms.

3.7.3 Converting voltage levels

Precautions must be made when connecting devices with different Transistor-Transistor Logic (TTL) levels as this might damage the lower-level chip or cause unexpected behaviour. The ATSAM3X8E, as well as many newer microcontrollers, uses 3.3V TTL while the ATmega162, and servo uses 5V TTL levels. As also stated in the previous exercise **Do not connect 5V devices to the ATSAM3X8E directly!**
There are multiple ways of converting the signals. For step-down conversion (i.e. 5V $\rightarrow$ 3V), a simple voltage divider of a 1 kΩ and a 2 kΩ resistor might be enough. For step-up-conversion of digital signals, you can use a Logic level converter or high frequency optocouplers. For analog signals an operational amplifier circuit will do the trick. *Note that you can only draw a limited amount of power from a normal Op-Amp. If you require power, there are power amplifiers available on the market. We will not need it for this project*

3.7.4 Detecting a lost ball - breaking an infrared (IR) beam

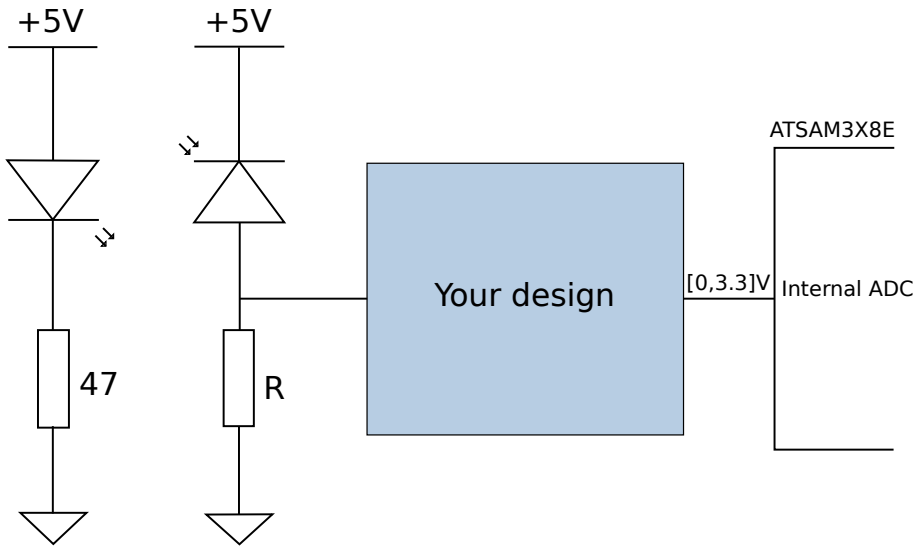


Figure 25: Suggested IR assembly

Choose an appropriate value for R based on the photodiode's datasheet. Design a circuit that converts the output voltage from the diode to the input range of the ADC. Use the operational amplifier.

3.7.5 Pinout for IO-header on blue ping-pong boards

solenoid servosig	solenoid servo5v	NC servognd	photo- photo+	IR- IR+

Table 7: Pinout for IO-header on blue ping-pong boards

3.7.6 Note on PWM generation on node two

There are two ways to generate PWM signals with the ATSAM3X8E: using the Timer/counter or by the dedicated PWM module. You **shall** use the PWM module, as previous experience with the timer counter have produced significant stuttering on of the servo. As an added bonus, it requires very little setup effort.

3.7.7 Exercise

1. The timer/counter module of the ATsam is a versatile module and can generate timer interrupts, PWM signals and more. Create a driver for the timer/counter module which allows you to use the PWM functionality. If you also implement the timer interrupt it might save you time when implementing the controller since most of you will most likely want to use timer interrupt there.
2. Create a PWM and/or servo driver which will use your controller output as an input and calculate the correct duty cycle/on-time which you will provide to your timer/counter driver. Also implement safety features which will never let the PWM go out of the servo's valid range.
3. Use an oscilloscope to verify that your driver in fact never goes outside the valid range of the servo (0.9-2.1ms).

4. Connect the servo on the game board to the PWM output on top of the Arduino shield.
5. Use the joystick position sent from Node 1 to control the servo position.
6. "Goals" are registered by blocking an IR beam. Install the IR-LED and IR-photodiode in the two holes located at the side walls of the game board.
7. Connect the IR diodes in a way that makes it possible to detect when the IR beam is blocked. An example is given in Figure 25. You might consider implementing an analog filter. **Check that the signal is correct before connecting it to the microcontroller**
8. As the IR signal is noisy and unstable you need to use the internal ADC of the Arduino to read the signal and filter out valid signal states.
9. Create a driver that will read the IR signal. You may want to implement a digital filter to reduce noise.
10. Create a function that is able to count the score. This will later be used for the game application.

3.7.8 Tips

- Read the SAM datasheet for information about how to set up PWM.
- There are multiple ways of doing the voltage conversion in the IR assembly. Think of what you have learned in previous courses. You should at least use an operational amplifier as a unity-gain buffer. For information on Op-amps, see *Curtis Johnson: Process Control Instrumentation Technology* or `https://www.electronics-tutorials.ws/opamp/opamp_1.html`
- Read values from the IR sensor and print them to the terminal to find the appropriate thresholds settings.
- If necessary, a simple RC filter can be used for low-pass filtering the IR-sensor/photodiode signal.
- Consider implementing delays for score detection that will eliminate spurious goals due to ball bouncing etc.
- Some digital cameras, such as the one in your cell phone, can detect IR light. You can use this to check if the IR-LED is active.

3.8 Controlling motor and solenoid

This exercise will implement motor control for the game board racket. The motor will be controlled by a PWM signal and a digital (i.e. 1/0) signal from the ATSAM3X8E via the A3959 motor driver using the so-called "phase/enable" control scheme. You must read the A3959 datasheet to find which pins to use and how to control the PWM signal for correct voltage output (functional description starts at page 6). For feedback on motor movement, you will use a timer/counter to register encoder output.

3.8.1 Encoder

An encoder is an electro-mechanical device that registers the movement of the motor shaft. As the motor shaft turns, the encoder outputs fractions of one revolution of the motor shaft in terms of square pulses. The pulses are generated internally by an LED and a photodetector separated by an opaque but slotted disk (codewheel) that rotates with the shaft, where the number of slots gives the resolution of the encoder. In our case, the encoder is incremental, meaning that it outputs relative motion only (in terms of pulses) and always starts at "0" when the system is powered up (as opposed to absolute encoders, which output the actual angle of rotation directly). Furthermore, this is a quadrature encoder, meaning that the encoder outputs *two* pulse signals which are offset by 90 degrees. The phase shift enables us to determine which direction the motor shaft is turning. You are encouraged to study the encoder's output in an oscilloscope to better understand how it works (if you haven't already, this is a good opportunity to get comfortable with the oscilloscope's acquisition function).

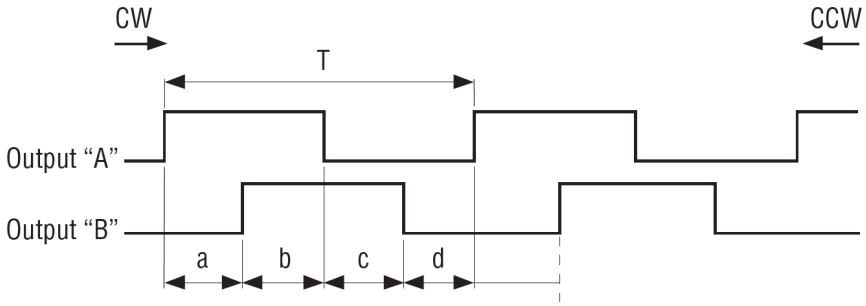


Figure 26: Example of a quadrature encoder pulse train, snipped from the RE30E datasheet

3.8.2 Solenoid

A solenoid will be employed to hit the ping pong ball. The solenoid requires 12 V to activate and you will therefore apply a relay to control it. A relay works as a current controlled electrical switch. By supplying current to its internal inductor the relay will push or pull mechanical contacts, and thus make or break electrical contact.

The relay draws too much current to be driven directly from an MCU I/O-pin and an intermediary transistor driver is therefore required to operate the relay, as shown in Figure 27. Inductors potentially create very high voltages when the current is turned off abruptly (inductive kick) which means that you need to connect a diode snubber as well. A capacitor may also be necessary.

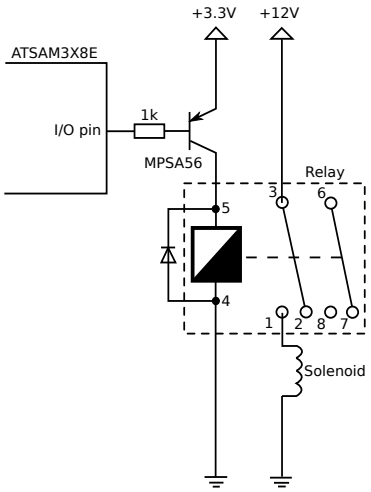


Figure 27: Connecting the solenoid to the ATSAM3X8E (note the pin numbering)

Internally, the relay is connected as shown in Figure 28. When there is voltage over the inductor the electromagnetic force will make the contacts move as indicated by the arrows. A spring will make the contacts return when there is no voltage. Assemble the solenoid circuit and test with 3.3 V to the resistor before connecting the Arduino Due.

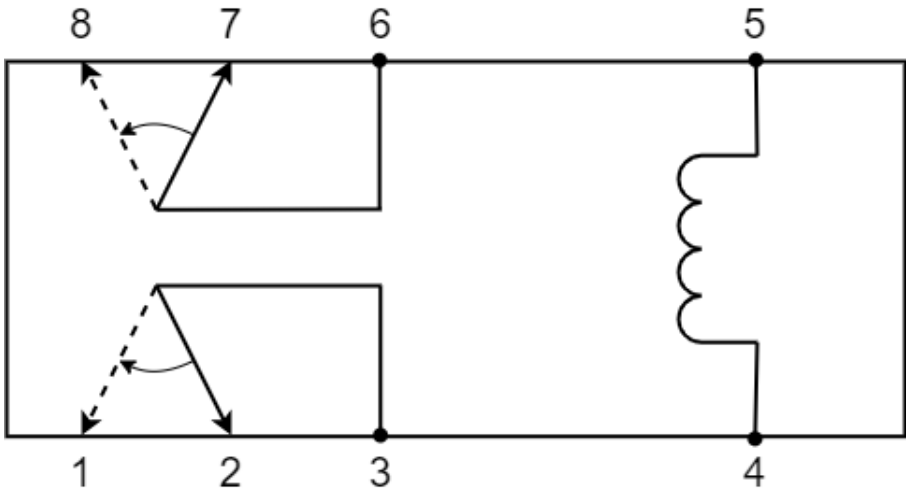


Figure 28: Relay (note the pin numbering)

3.8.3 Controller

A simple yet robust controller is called the PID controller. The continuous form of this controller is given in Equation 1

$$u = K_p e + K_i \int e dt + K_d \frac{de}{dt} \tag{1}$$

where e is the error (the difference between actual and target value). When working with microcontrollers we usually don't have any means to input continuous time equations, meaning we might prefer the discrete version given in Equation 2, where T is the sampling period.

$$u[n] = K_p e[n] + TK_i \sum_{i=0}^{n} e[i] + \frac{K_d}{T}(e[n] - e[n-1]) \tag{2}$$

If we wish to use this discrete version then we will have to know T. One common way to do this is by using a timer interrupt. The minimum requirement for the exercise is to implement a

closed-loop/feedback position based controller. However, you are free to use a PI, a PID or some extensions of these, though the D part is generally not needed so most groups end up using a PI controller.

3.8.4 Exercise

1. Connect the encoder input to the motor encoder by using the designated cable. Use TC2 of the ATSAM3X8E in quadrature decoder mode and verify that it reads and decodes the motor motion.
2. Connect the shield's 4mm jacks to a 12V power supply, and the shield's 2mm jacks to the motor.
3. Develop code so that the motor speed and direction can be set by joystick position (x-axis for motor, y-axis for servo) using the the A3959 motor driver in phase/enable mode. You should now be able to play the game, although it is hard to control.
4. Create a position based controller (closed-loop, using feedback from the encoder) for the motor so that the position is easier to control. A PI controller should be sufficient. Exact tuning is not important, but the game must be playable with the controller.
5. Being a course/lab provided by the cybernetics department, a closed-loop/feedback position based controller is **the minimum requirement for approval of the exercise and the lab exam**. If you prefer a speed based controller, a position based controller must still be implemented.
6. Extend the code to include solenoid triggering when one of the joystick buttons is pushed. It should generate a pulse just long enough to hit the ball.

3.8.5 Tips

- Review the *TTK4155 Motor Shield* document.
- If you have implemented a position regulator you might have noticed that the wagon will not reach the desired position at slow speeds. This is caused by friction that inflicts a dead band that has to be overcome for the motor to start moving. This "stick effect" can be avoided by integrating the position error, using smart/dynamic values for the gain parameters.
- When you are certain that the shoot-routine works you might increase the solenoid voltage to 16 V. This should not damage the solenoid since the pulse is rather short.
- The solenoid and relay might draw a lot of current. Ensure that proper decoupling is in place and consider connecting them to power supplies separate from the digital parts of the circuit.

3.9 Completion of project and extras

When the features of all previous exercises are implemented there is only the final touch left. Make sure that the game is fully operational and prepared for the evaluation, and optionally implement some extra features that will impress the evaluators. Some suggestions for extra features:

- Documented code (e.g. UML)
- Creative use of the display
- Online tuning of regulator
- Writing to the display over RS232/SPI/CAN $\longrightarrow$ AT90USB128 $\longrightarrow$ OLED
- Dual buffering display data in SRAM
- Utilizing more functionality on the USB multifunction card, like its CAN interface.

Examples of extensions that have been implemented earlier:

- Replay: all motor inputs saved to SRAM, so that the game can be replayed
- Difficulty levels: different regulators can be used to give varying difficulty
- Saving and statistics: create user accounts for the group members and save statistics for the ping pong skills
- Camera control: computer vision is used to detect the position of the ball and let the game "play itself"
- Remote motion control: the game can be controlled using wireless controllers with accelerometers
- Sound: sound effects or music can be implemented using PWM and a speaker. *Do not do unto others as you would expect they should do unto you. Their tastes may not be the same.*
- `http://www.youtube.com/watch?v=cdPKJv5O21g`
- `https://www.youtube.com/watch?v=X0gf8bat_Qo`

Also have a look in the "Extras" folder on Blackboard. It might also be possible to borrow extra components if needed. Ask the teaching assistants. Try to keep the extras within the realm of embedded systems. Good luck! :)

Congratulations! You have now made a complete embedded system. We feel sorry for you when having to pull it all apart after the evaluation!