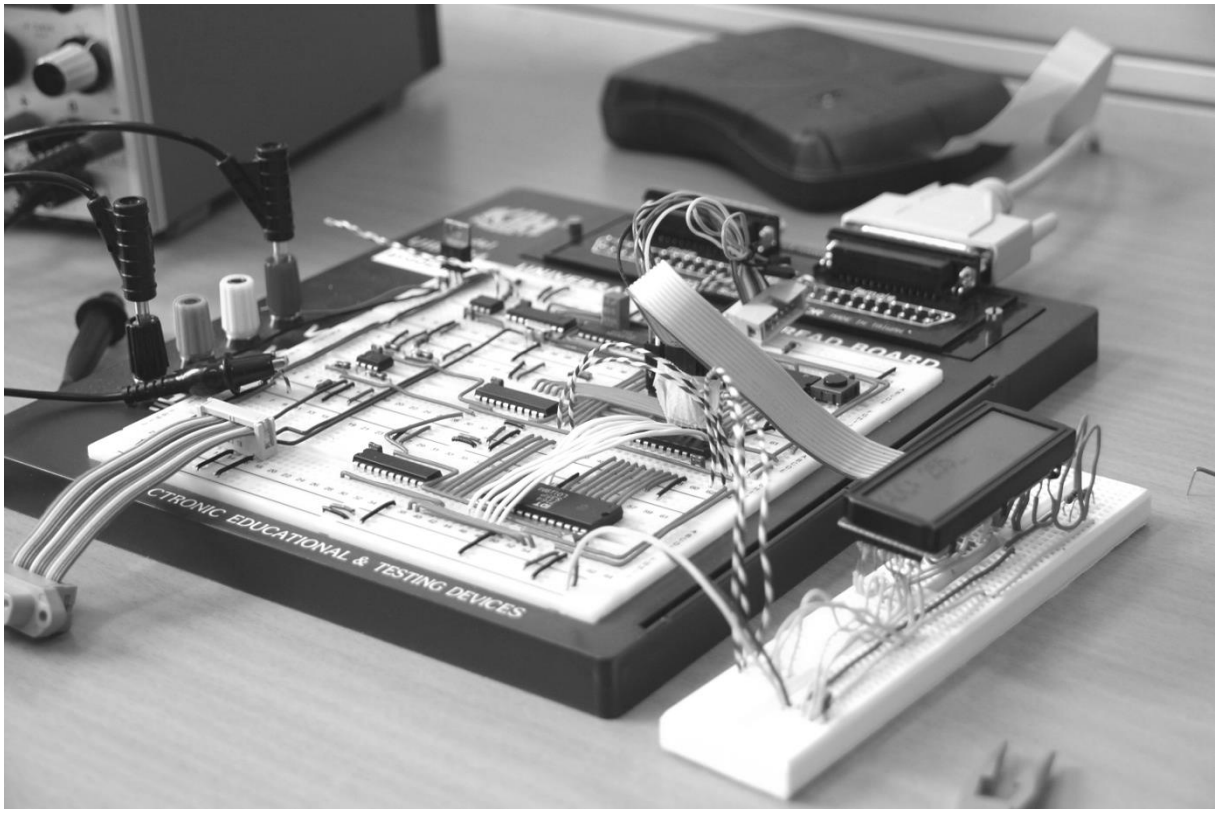
TTK4155
Industrial and Embedded Computer Systems Design

Term Project

Ping pong game with a distributed embedded control system

Version 3.4.1 – Aug 2017



Contents

1 Introduction

This term project aims for creating a computer controlled electromechanical ping pong game. The electronic components, game boards and development tools needed to realize the game will be handed out in the beginning of the semester. The main challenge will be to assemble the hardware components and develop software for the microcontrollers, making 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 which 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 which will ensure continuity in the work and necessary progress in the project.

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

1.1 System overview

As shown in Figure 1, the system is divided in two main parts: Node 1 and Node 2. Node 1 consists of the breadboard and the USB Multifunction card, and Node 2 of an Arduino Mega 2560 development board and a dedicated I/O-card.

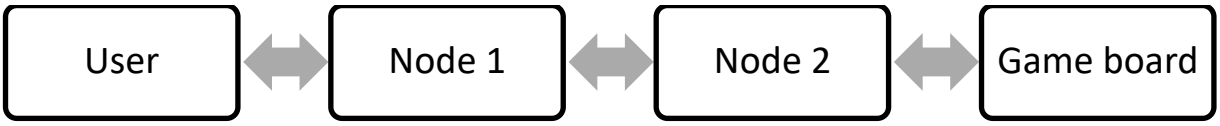


Figure 1 – system overview

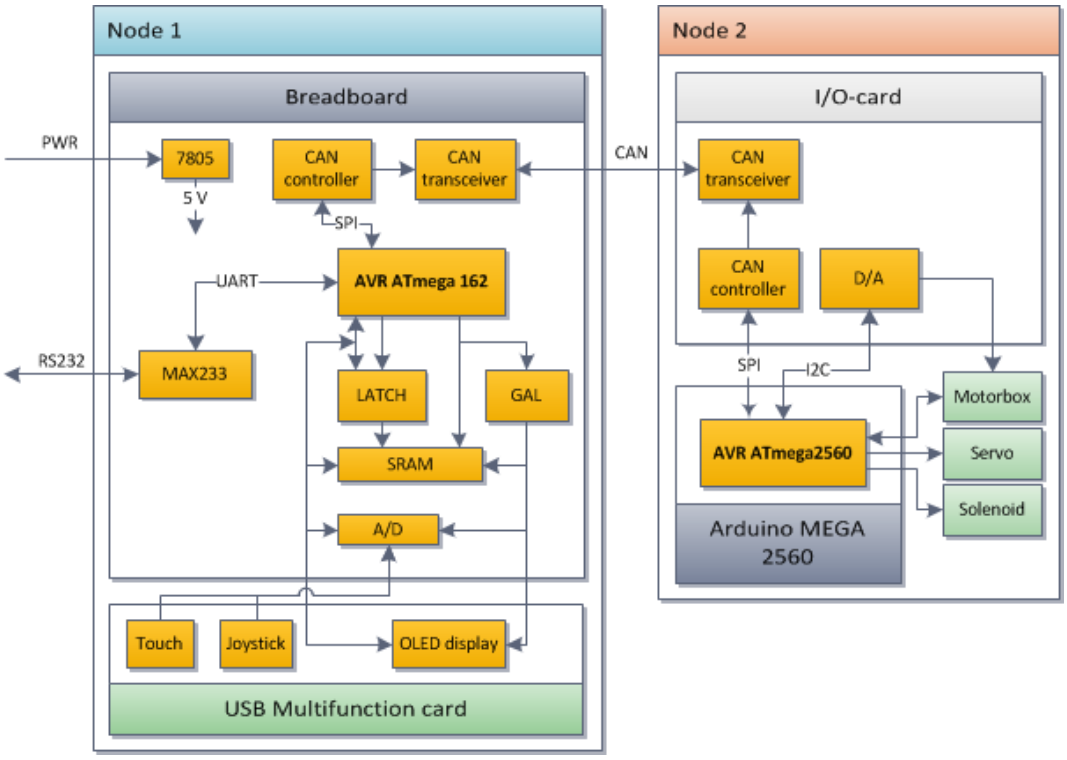


Figure 2 – Node 1 and 2 with their main components

In the first exercises, only Node 1 will be used. Node 2 will be added later on when including the game board and the CAN bus. Figure 2 gives an overview of which components each of the two

nodes consist of. As shown, the CAN bus enables communication between the two nodes and can also be used to connect two complete systems together for two-player mode.

1.2 Practical information

1.2.1 Groups

The project will be carried out and evaluated in groups, each consisting of two or three students. Registration will be taken care of in 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, voltage generators 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 assistants before each new exercise. The topics will be directly relevant to the upcoming project exercise, and will contain useful information, hints and tips.

The first lecture is particularly important; all students are required to participate here.

1.2.3 Schedule

The table below gives an overview of all project exercises and a schedule showing which week they are expected to be completed.

Week	Exercise to be completed
35	1: Initial assembly of microcontroller and RS-232
36	2: Address decoding and external RAM
37	3: A/D converting, joystick and touch input
38	4: OLED display and user interface
39	
40	5: SPI and CAN controller
41	6: CAN-bus and communication between nodes
42	*Catch up week*
43	7: Controlling servo and IR
44	8: Controlling motor and solenoid
45	9: Completion of the project and extras
46	*Catch up week*
47	*Evaluation*

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.

1.2.4 Approval of exercises

After completing an exercise, the result **must be shown to and approved by one of the teaching assistants** before you can start working on the next exercise. This will help you verify that it is performed with sufficient quality and make it easier for you to proceed.

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). Your student card will give you access to this room automatically if you are a

student of Engineering Cybernetics. Otherwise, send an e-mail to the teaching assistant as soon as possible with your full name and card number (below the return address, not the one starting with "NTNU"). Some cards may have two number sets, in this case send both.

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 these periods. Beyond that, you are free to use the lab when it's not reserved for other courses.

There are lockers just 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 Atmel AVR 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 TA 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 and deposit fee

The development kit required to carry out the project will be handed out at the component store (see below) following the first lab lecture. Note that each group has to sign out the kit and provide a deposit fee of NOK 200.

The kit consists of relatively fragile components and should be handled with utmost care and **be returned to the component store in good order immediately after the final evaluation. This is a requirement in order to get access to the final exam.** Damaged components should be handed to the teaching assistants.

An up to date list of the contents of each kit will be handed out together with the components.

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.

Preliminary component list 2016

Component	Name	Amount
ATMEL microcontroller	ATmega 162	1
Voltage regulator 5V	MC7805CT	1
Maxim RS232 transceiver	Max233	1
Lattice GAL	GAL16V8D-15QP	1
CAN controller	MCP2515	1
CAN transceiver	MCP2551	1
National Semiconductor ADC	ADC0844CNN	1
Opamp	MCP602	2
SRAM 64kx8	IDT 7164S20TPG	1
Transistor	MPSA56	1
D-latch	SN74ALS573	1
Crystal	4,9152MHz	1
Crystal	16MHz	1
IR emitter	SFH484 (purple)	1
Photo diode	SFH203-FA (black)	1
LED	HLMP 1340	3
Diode	1N4005	1
Kuhnke 5V relay	RG2L-5VDC	1
Push button	Multimec 3CTL-9	1
Capacitor	22pF (yellow) (marked with “220”)	4
Capacitor	100nF (yellow) (marked with “104”)	11
Capacitor	1uF Wima (red)	1
Capacitor	10uF tantalum (yellow)	1
Capacitor	470uF electrolyte	1
Resistor	47Ω	1
Resistor	120Ω	2
Resistor	470Ω	3
Resistor	1KΩ	1
Resistor	10KΩ	5
Resistor	22KΩ	1
Resistor	47KΩ	2
Resistor	100KΩ	1
Resistor	330KΩ	1
Resistor	1MΩ	1
Breadboard (big)		1
Breadboard (small)		1
Wire box		1
Wire cutter		1

Preliminary Tool list 2016

- Arduino mega 2560
- I/O Module for Arduino mega 2560
- Mascot voltage supply, 9V DC
- Atmel-ICE programmer
- Cables
    - 25-9 pin DSUB cable (between breadboard and RS-232)
    - 9-9 pins DSUB cable (if needed)
    - 10 pin split ribbon cable (JTAG if needed)
    - 6 pin ribbon cable (if needed)
    - 10 pin ribbon cable (miscellaneous) (4X)
    - 10 pin ribbon cable 40 cm to MJ1 and MJ2 (motor control box) (2X)
    - 2 pin cable 15 cm (4X)
    - 2 pin cable 40 cm (4X)
    - 10 pin split ribbon cable ca. 1m (ping pong board)
    - 6 pin ribbon cable ca. 1m (ping pong board)
    - Red and black/blue cable (between motor and motor control box) 2 x 1m

USB Multifunction card

- USB Multifunction card
- Jumper Wire - 0.1", 6-pin, 12" (2x)
- Single row adjusted 12 pin header (might be available at the lab)
- 3 m USB mini cable

1.2.7 Documentation and datasheets

All necessary datasheets will be 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.

Atmel Studio's help system also contains a lot of valuable information. For information about the various development tools and debugging devices, see Help > AVR Tools User Guide. Also, the Atmel Studio User Guide describes the software itself.

Useful links

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

Atmel YouTube Channel
http://www.youtube.com/user/AtmelCorporation

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

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

Whatis (technical dictionary):
http://whatis.techtarget.com/

1.2.8 Evaluation

At the end of the semester, hopefully after completion of all exercises, groups are required to present their project results in a demonstration and oral presentation to the course teacher for evaluation. Every group will get their own time slot allocated (approx. 15 minutes) on the evaluation day. Source code you have developed must be uploaded to Blackboard prior to the presentation. No reports or other written materials are required, but groups are recommended to plan a short (3-5 min) presentation thoroughly and possibly work out material that will make their work and efforts easier to comprehend for the evaluators. The remainder of the presentation will be a Q & A session. All group members must be present during the lab days in the evaluation week! More information on the details of the evaluation will be provided later on.

Completion and functionality of the project exercises are weighed 80 % of the project score while the remaining 20 % can be achieved through implementation of extra features and creativity. See chapter 3.9 for suggestions. Mind that the score is set on the evaluation day based on, but not restricted to, game functionality, general quality impression, code readability, code structure, functionality, breadboard coupling, logical color coding, presentation (does every group member understand the implementation, does the entire group contribute to the presentation) and ability to answer questions. Completing and having all exercises approved does **not** automatically yield 80 % score on the project. It is important to have the exercises approved, so if the system fails on the evaluation day, we can at least see that isolated parts of the system were working at some point.

Group members will get a joint score. The **project counts 50% of the final grade** while the written exam counts 50%.

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).**

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 and software in an efficient manner (see chapter 0 and 2.2.8). 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 go 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 create extra features, which will improve the final result and probably give a better score.

Make sure you visualize the project in a larger context than just the scope 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 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 for other tasks that may come?

2 General background information

2.1 Hardware and electronics

2.1.1 Breadboard

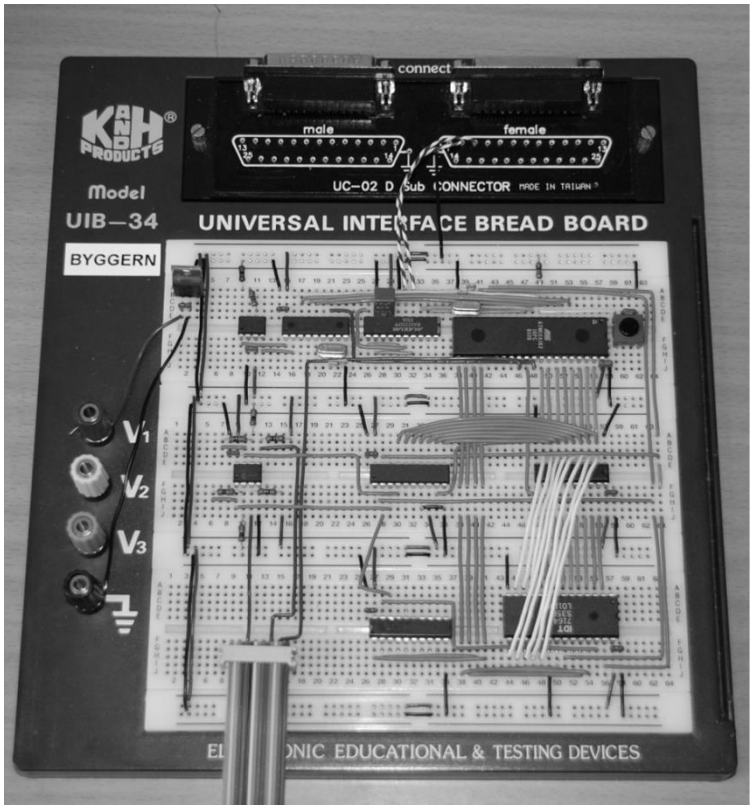


Figure 3 – breadboard

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. Try to understand how holes are connected by using the multimeter.

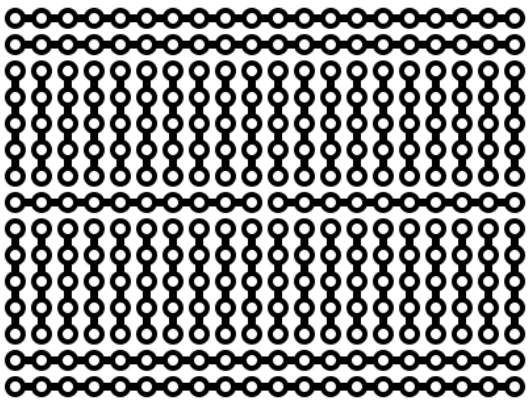


Figure 4 – typical hole pattern of a breadboard [Wikipedia]

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.** Also, make sure to place the components in the correct direction – note the arrows on the ICs in the figure.

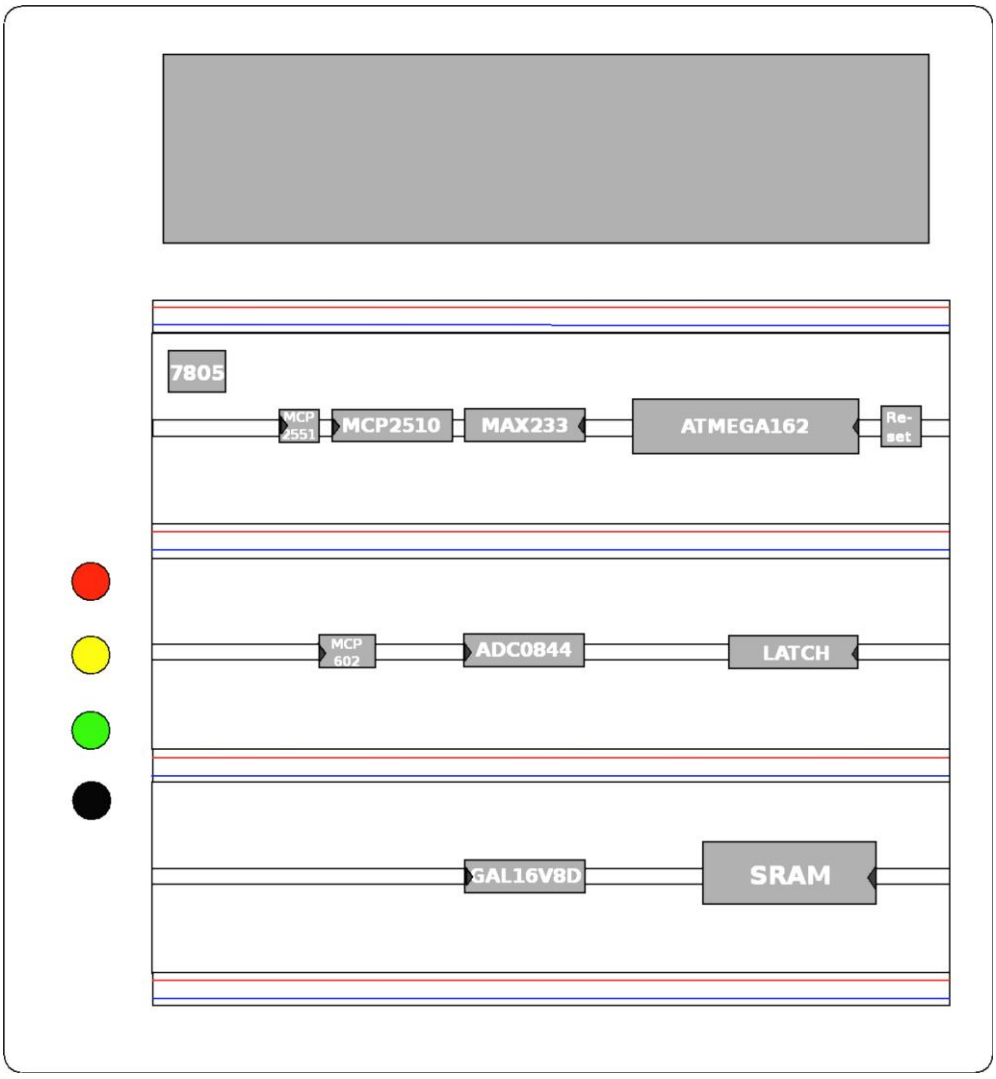


Figure 5 – Component placement on node 1. Note that we use MCP2515, NOT MCP2510 this year.

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. 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.

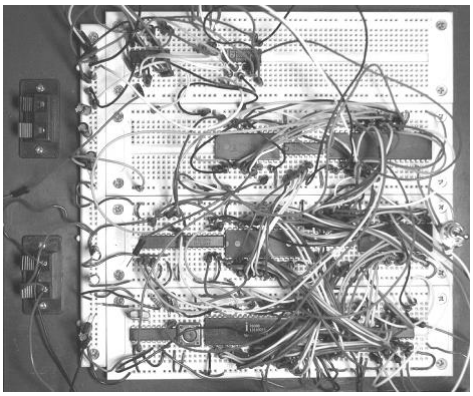


Figure 6 – messy assembly [Wikipedia]

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 crystal 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 current 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 ($1/j\omega C$) 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 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. Figure 8 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.

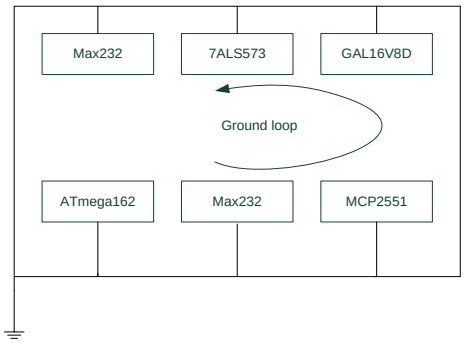


Figure 7 – ground loop (wrong)

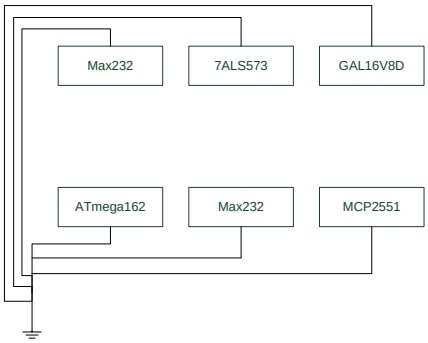


Figure 8 – star grounding (correct)

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 X iteration of the main program loop, provides a visual indication that the code is in fact executing.

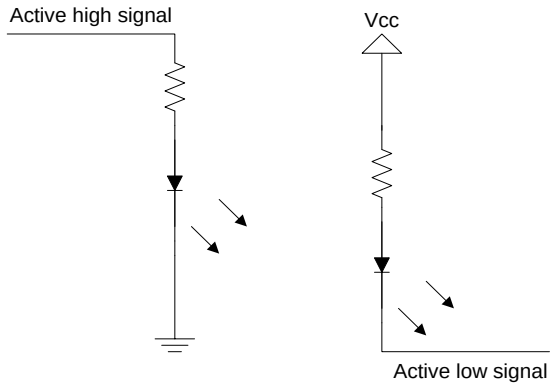


Figure 9 – LED connections

Figure 9 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. 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 10 k - 100 k 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.

2.1.5 The Atmel AVR microcontrollers

The Atmel AVR microcontrollers used in this project are based on an 8-bit RISC 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 in an instruction, and results must be explicitly stored back from the register file to data memory (by 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, SPI and TWI (I^2C) 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 Atmel's product pages and datasheets.

We will use minimum two different AVR microcontrollers in this project, an ATmega162 in Node 1, optionally the AT90USB1287 on the USB Multifunction card and an ATmega2560 in Node 2. All devices have similar methods for programming, but the AT90USB1287 and ATmega2560 also comes with an USB bootloader. Most of the information required to start developing with them is contained in their respective datasheets.

2.1.6 USB Multifunction card

The USB multifunction card contains an Atmel AT90USB1287 microcontroller. It is not mandatory to program the MCU, but the card has a lot of features, so this is a great piece of hardware to use in the creation of additional features.

We are only going to use the joystick, pre-processed touch signals and the OLED display in the exercises.

Some of the onboard features:

- 128 x 64 pixels graphic OLED display
- Thumb Joystick w. button

- LEDs and buttons
- USB boot loading: Use Atmel FLIP software to program the MCU over USB.
- Touch interface
- CAN interface
- Header for Atmel's RZ600 wireless transceivers
- Speaker

2.1.7 Arduino Mega 2560

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 (simplifying porting of e.g. the CAN code), debugging is vastly simplified and one avoids the struggle with many Arduino libraries which are poorly implemented.

2.1.8 Atmel-ICE

The Atmel-ICE is a powerful development tool that lets you debug programs that are actually running on an AVR microcontroller (on-chip debugging). Together with Atmel 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.

2.1.9 JTAG on the Arduino

The interface and language used to communicate with an Atmega during debugging is called JTAG. This is one of many programming and debugging standards, most Atmegas 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. In the Arduino this is done to prevent reprogramming of the built in bootloader, and to free up the pins used by JTAG.

To be able to use debugging on the Atmega2560 we therefore have to enable JTAG again. Luckily this can be done via the ISP interface which is left active. To enable JTAG, go to device programming with the board powered, and connected through ISP to the Atmel-ICE. Once there, read the device ID in the top bar by pressing the right read button, and go to fuses. If you got an error message, you might have to turn the connector around. In Fuses, tick the JTAG enable box, and press program. Your Arduino Mega 2560 should now support JTAG programming and debugging through the angled 10-pin connector on the shield.

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 IC's 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.
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 correct.
12. Borrow IC's from another group to eliminate code errors.

2.1.11 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 values. Ensure that all unused input pins of an IC are connected to either supply voltage or ground.

2.2 Programming in C with AVR

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/

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 instruction set, and one interface driver for sending and receiving CAN messages.

For each driver, make one header file (.h) and 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 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 I2C (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 score.

Automated tools that will generate on-line documentation in HTML from the comments in your source files are freely available, for example Doxygen. See http://www.stack.nl/~dimitri/doxygen/ 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 MCUs 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 PA6 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 pull-up 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. 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 one needs some of these pins.

2.2.4 Bit manipulation

Configuration and control of a microcontroller and its peripherals is 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 positions. Example:

```
PORTA |= (1 << PA0);
```

Some useful expressions for manipulating bits:

Only the bit in position CS02 set: `(1 << CS02)`

Only the bits in position COM01, COM00, and CS02 set:

```
(1 << COM01) | (1 << COM00) | (0 << WGM01) | (1 << CS02)
```

Set the bits CS02 and COM01 in TCCR0 register, clear the other bits:

```
TCCR0 = (1 << CS02) | (1 << COM01);
```

Set the bits CS02 and COM01 in TCCR0, leave the other bits unchanged:

```
TCCR0 |= (1 << CS02) | (1 << COM01);
```

Clear the bit CS02 in TCCR0, leave other bits unchanged:

```
TCCR0 &= ~(1 << CS02);
```

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 ) )
```

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 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 the event requires a short response 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.

Interrupts are a technique of automatically diverting the processor from the execution of the current program so that it may deal with some event that has occurred, for instance when an ADC conversion is complete or when 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 current 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 have 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 upon 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 at any time using the functions `sei()` or `cli()`, respectively. 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 is described in the AVR datasheets.

```
// POLLING EXAMPLE (ATmega162)
// 8 buttons connected to PORTA, 8 leds to PORTB

#include <avr/io.h>

int main()
{
        // Buttons input (the whole port)
        DDRA = 0;

        // Leds output (the whole port)
        DDRB = 0xFF;

        while(1){
                // Read A, set it out on B
                PORTB = PINA;

                //Do something else in X ms
        }
}
```

```
// 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!
}
```

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;
```

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:

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

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).

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
`stdint.h`	Standard Integer Types

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.

Note that the delay functions 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 following lines of code.

```
#define F_CPU 16000000 // clock frequency in Hz
#include "util/delay.h"
```

2.2.8 Software debugging

Atmel 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 microcontroller chip is possible by using the Atmel Studio in conjunction with the Atmel-ICE. The AVR Tools User Guide gives an in-depth description of debugger.

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.

2.3 Tools and software

To develop software for the MCUs, you are free to use the development environment and tools of your own choice, but Atmel Studio (Windows only) together with AVR-GCC is recommended because of its good debugging facilities, integration with compiler and programmer tools, and good support from the teaching assistants. Alternatively, AVR-GCC together with any text editor in Windows or Linux can be used. In Linux, AVRDUDE can be used for programming the MCUs (transferring the binary program to the microcontroller).

In addition, you will need terminal software to communicate with the MCUs over RS-232.

2.3.1 Atmel Studio 6.2

Atmel Studio is built specifically for AVR microcontrollers and together with WinAVR and the Atmel-ICE, Atmel Studio constitutes a complete development environment for the AVR microcontrollers. Spending some time exploring the various features that Atmel Studio provides is 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. Atmel Studio can also simulate all microcontroller devices in the AVR family.

Atmel Studio 6 and updates can be found at http://www.atmel.com/tools/atmelstudio.aspx and AVR-GCC at http://winavr.sourceforge.net/.

2.3.2 Version control

Files and catalogs stored on the computers in the real-time 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. Another popular version control system, widely used in multiple industries, is SVN http://subversion.apache.org/ . And many other alternatives exist as well.

2.3.3 Terminal

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

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.

Creating a new project in Atmel Studio

After opening Atmel 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.

RS-232

RS-232 is a serial communication interface standard that is widely used in embedded systems. It has been on the market for a very long time and due to its simplicity and abundance, it still has a 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 10. Figure 11 shows the relation between the standard contacts DB9 and DB25, where the latter is the connector found on the breadboard.

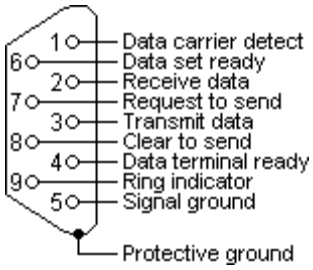


Figure 10 – [Lammert Bies]

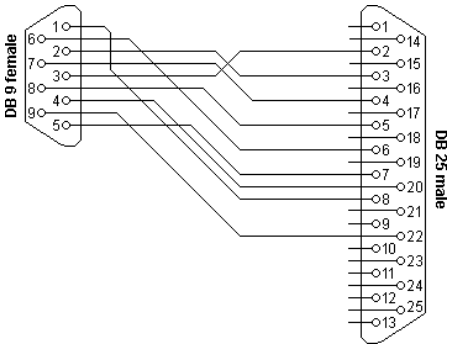


Figure 11 – [Lammert Bies]

DB9 pin	DB25 pin	Name	Description
1	8	DCD	Data carrier detect
2	3	RX	Receive data
3	2	TX	Transmit data
4	20	DTR	Data terminal ready
5	7	SGND	Signal ground
6	6	DSR	Data set ready
7	4	RTS	Request to send
8	5	CTS	Clear to send
9	22	RI	Ring indicator

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 might use a 1 µF capacitor on the input instead of the recommended 0.33 µF. Use a voltage supply of 8-12 V and verify using a multimeter that the output voltage is a stable 5 V. 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 0.1 µF 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 in Atmel Studio 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.

Now, you have connected the basic components needed to get the MCU up and running. The following steps will enable the 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 $T1_{IN}$ and $R1_{OUT}$ on the MAX233, respectively. Correspondingly, $T1_{OUT}$ and $R1_{IN}$ are to be connected to the serial line to the computer by using a DB9 – DB25 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.

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.
- Before programming the MCU, verify its fuse bits. This check can be executed from Atmel 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)
- 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 behavior 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 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).

External memory

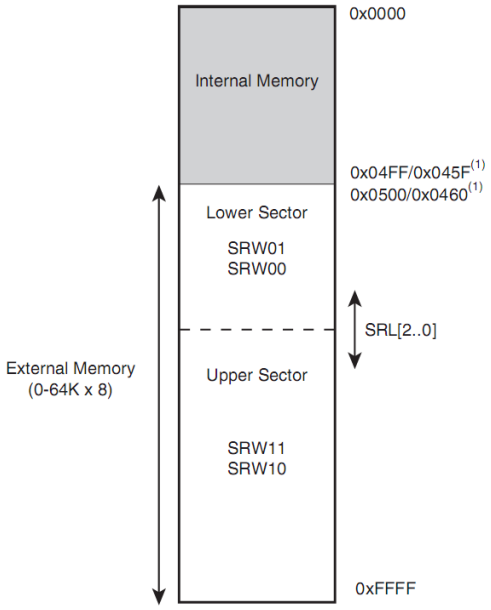


Figure 12 – external memory [ATmega162 datasheet]

The organization of the ATmega162 data address space is shown in Figure 12. 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 behavior can be exploited when organizing 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.

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 three different devices to the external bus: an SRAM, an A/D converter and an LCD display. 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.

In this project, the address decoder will be implemented in a programmable logic IC, called a GAL (General Array Logic). The GAL makes it possible to write the address decoding logic in a hardware description language and subsequently program the IC to implement the logical functions via a dedicated programming tool.

Also, note that the ATmega162 supports sector select, meaning that different memory areas can have individual wait-state (timing) settings.

VHDL

VHDL ((Very High Speed Integrated Circuit) Hardware Description Language) is a hardware description language originally written for the US Department of Defense, but is now considered an industrial standard for describing the function of programmable logic ICs.

The code needed for implementing the address decoder is very simple.

A VHDL tutorial can be found at http://www.seas.upenn.edu/~ese201/vhdl/vhdl_primer.html. Reading the full tutorial should not be necessary; the first few chapters should be sufficient.

The following example will get you started.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity address_decoder is
    Port (
        a11 : In std_logic;
        a10 : In std_logic;
        a9  : In std_logic;
        a8  : In std_logic;

        ram_cs  : Out std_logic;
        adc_cs  : Out std_logic;
        oled_cs : Out std_logic;
        oled_dc : Out std_logic  -- Not necessary…
    );
    attribute LOC : string;
    attribute LOC of ram_cs  : signal is "P19";
    attribute LOC of adc_cs  : signal is "P18";
    attribute LOC of oled_cs : signal is "P17";
    attribute LOC of oled_dc : signal is "P16"; -- Not necessary…

    attribute LOC of a11     : signal is "P1";
    attribute LOC of a10     : signal is "P2";
    attribute LOC of a9      : signal is "P3";
    attribute LOC of a8      : signal is "P4";
end address_decoder;

architecture behave of address_decoder is begin
    -- implement the functionality here
end behave;
```

The following is worth noting:

- Comments start with "--", assignments are written as "<=".
- The `LOC` attributes specifies which pins the different signals are assigned to.
- To get the desired functionality, write logical expressions in the behavior part. Input signals are combined to determine output signals. The logic can be written using Boolean expressions or truth tables. A Boolean expression can for example be
  `output <= (NOT input1) OR (input2 AND input3);`
- The example gives a recommended pin layout with subsequent exercises in mind. You are recommended to follow this layout to make your code compatible with other group's hardware for testing purposes, and it will also save you the hassle of reorganizing the wires when new signals are required.

Programming the GAL IC

When the desired logic functions are implemented in the VHDL file, it should be compiled and burned to the GAL IC. The burning is done by using the "Dataman-Pro Pg4uw" software on the "GAL-pc" by the door in the lab. Note that the VHDL coding and compiling should be done on your lab PC and then transferred to the PC by the door.

The following procedure creates a new project:

1. Start "ispLever Classic Project navigator" on your workspace computer (not the burner PC).
2. File > New Project …
3. Choose name and location for the project. Select VHDL and Synplify as synthesis tools.
4. In the next window, select the following:
    a. Family: GAL Device
    b. Device: GAL16V8D
    c. Speed Grade: -15
    d. Package Type: 20PDIP
    e. Part name: GAL16V8D-15QP
5. In the next window, select nothing and proceed to the next one, where you click Finish. The project is now set up.
6. Source > New …
7. Select VHDL module. A window will appear where the name of the VHDL module can be entered. Here, entity, architecture and port mapping can also be configured. This information can also be written directly into the VHDL file.
8. An "empty" VHDL file will be generated where you can enter your code.
9. When finished writing the code, save and close the VHDL file and text editor.
10. Select the "chip" in the project (GAL16V8D-15QP)
11. In the right window, by double clicking Chip Report, a report with inputs and outputs is displayed.
12. Double click JEDEC File. This will generate a .jed-file that will be used to burn the GAL IC.

Now, the address decoder is compiled and ready to be written to the GAL IC by using the EPROM burner. The GAL16V8D IC is rewriteable so there is no need to erase it in a UV eraser. Transfer the .jed-file to the burner PC using a USB-stick, or shared folder.

13. Start "Dataman-Pro Pg4uw". If you are asked for what kind of programmer to use, select "Dataman-40Pro" and click connect.
14. Click the "Select" button and find Lattice GAL16V8D.
15. Click "Load" and find the .jed-file you created earlier.
16. Insert the GAL IC to the burner.
17. Click "Program" and "Device operation options". Enable the functions you need, and select "Device" under "Programming parameters". Click OK and then Yes.
18. The GAL IC is now ready for use.

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.

Suggested memory mapping for this project and code example

As shown in Figure 12, 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 OLED display at addresses 0x1000 – 0x13FF, the ADC at 0x1400 – 0x17FF, and the SRAM at 0x1800 – 0x1FFF, as shown in Figure 13. This will allow us to use only a few bits of the address to perform the address decoding. We will only use 0x800 * 8 bit = 16 kbit of the SRAM, and at the same time we will "waste" a lot of addresses on the OLED and ADC. That is the disadvantage of partial address decoding, but it gives very simple address decoding logic. We could of course have included additional address bits in the decoder to make more efficient use of the address space.

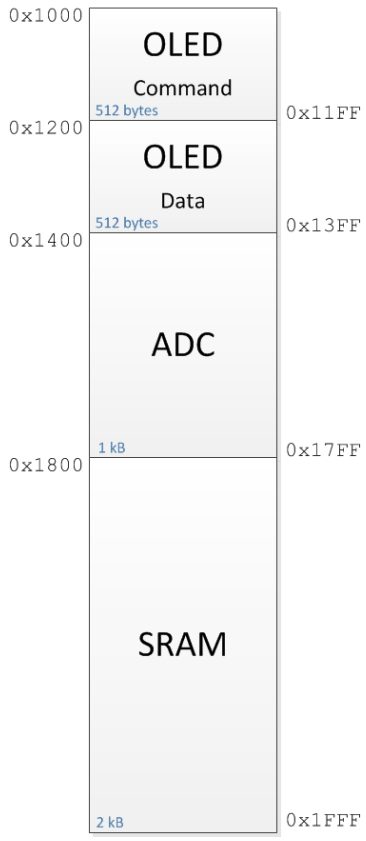


Figure 13 – recommended memory mapping

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.

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:

```
#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);
}
```

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. Do not worry if there is a small amount 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:
    - OLED display (needs at least two addresses)
    - ADC (needs at least one address)
    - 16 kb SRAM (needs at least 2048 addresses (8 bit x 2048 addresses = 16384 bits = 16kb = 2 KB))

    Implement the address decoder using the GAL IC. Connect it, and verify that the correct chip select signals are generated by running a test program and measuring the resulting output signals.

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 USB multifunction card.

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 14.

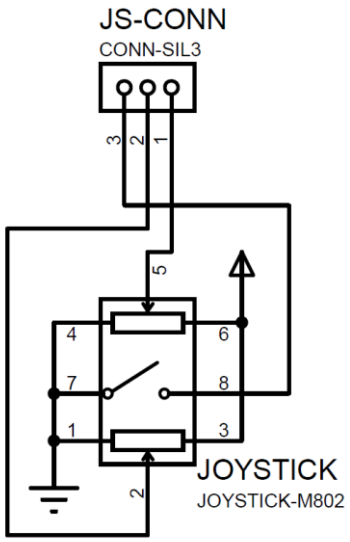


Figure 14 – joystick and internal schematics

Touch

The card has two touch sliders and two touch buttons. The AT90USB1287 uses Atmel's free QMatrix library to read and process the touch signals. It then outputs the slider values as PWM signals and the buttons as standard output.

ADC0844CCN

The ADC0844CCN 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.

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

Exercise

1. Connect the ADC0844CCN, 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 USB multifunction card and check that the positions of sliders and buttons are shown on the OLED. Make sure the jumper across “EXTSEL” is connected.
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 PWM signals to the ADC through the low pass filter onboard the USB multifunction card. Compare the signal before and after the low pass filter. What is the cutoff frequency and the slope of the filter?
6. Connect the buttons to the microcontroller.
7. Write an expression explaining the relationship between joystick angle and voltage.
8. Make a software driver that can return the joystick and slider positions.

Tips

- If the positions and buttons are not shown on the OLED, reflash the card with your JTAG and use the file “Touch to PWM.hex” on Blackboard. Make sure pin 1 lines up with pin 1 on the JTAG.
- “EXTSEL” switches between internal and external control of the OLED.
- The potentiometers inside the joystick works as voltage dividers.
- Find the filter construction in the USB Multifunction card schematic. There are many online filter calculators.
- Use the oscilloscope to find the relationship between the joystick position and its internal resistance. Find the output voltage as a function to 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.
- 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.

3.4 OLED display and user interface

In this exercise you will connect the OLED display on the USB Multifunction card to the MCU's parallel bus interface to enable display of text and information to the user, e.g. to print the score while playing and to display a menu. This includes making the software drivers necessary to operate the display.

LY190-128064

The display to be used is a monochrome 128*64 dot matrix display module. It can be connected to the MCU's parallel bus and operated in the same way as the SRAM and ADC. In addition to the 8-bit parallel bus interface, which is what we will use, the display can also be controlled using SPI or I^2C.

The display is hardware configured to use 8080-parallel interface with only write operations possible (!RD always high). In that mode the control signals we need to use are !WR, !CS and D/!C (data/command). The display has two data registers; one for command and one for data. The D/!C pin decides which register is to be accessed; command (D/!C = 0) or data (D/!C = 1).

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 creating additional features that will earn you some extra score in connection with the evaluation, e.g. making animation, own characters, dual buffering, drawing functions (circles, cubes, …) etc.

Exercise

1. Extend the logic in the GAL IC, so that it will generate correct control signal(s) for the display based on address.
2. Connect the display to the data bus and the required control signals from the GAL and the MCU. Is it possible to connect the !WR on the OLED directly to the !WR from ATmega162? Check table 7-2 in the OLED advanced datasheet up against the external memory timing diagrams in the ATmega162 datasheet to find out.
3. How is it possible to write to the whole screen with the given memory mapping (512 bytes) when the screen data is 1kB?
4. Make a driver for the display, including an initialization routine to set up the display, a write function that prints one character, and a command function that controls the OLED configuration registers.
5. The font is too big to store in data memory. Store the font in PROGMEM and use it from there.
6. Implement `printf`-support for the display. Alternatively, you might create your own function to print strings.
7. 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.
8. Optional: Create your own characters.
9. 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 SRAM.
10. Optional advanced: Create drawing functions like oled_line(x0,y0,x1,y1), oled_circle(x,y,r) etc.

Tips

- See table 7-1 in the OLED advance datasheet to get an overview over the control signals in 8080-mode.
- You need to connect a jumper on the USB Multifunction card to access the screen externally. Look in the schematics!
- 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: http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=38003
- 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 hardware- and software-wise before you start.¨

3.5 SPI and CAN controller

This exercise will make the basis for enabling CAN network communication in our system. It will be realized using a stand-alone 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 implement the SPI bus to enable communication between the MCU and the CAN controller. Then, you will make a driver that lets you access to the CAN controller's control registers. In this exercise, the CAN controller will be tested using it in a loopback mode (sending to itself). In the next exercise, you will enable communication between Node 1 and Node 2, as described in chapter 1.1.

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.

MCP2515

This project uses the CAN controller MCP2515 which implements the data link layer and some of the physical layer of the CAN protocol. Each node will be physically connected to the bus via the CAN transceiver MCP2551.

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.

The controller has several registers and a header file with register names and addresses is provided on the manufacturer's homepage.

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.

SPI

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.

Modularization

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 understand and maintain, as well as it opens the possibility of reuse. Remember that both nodes use the same CAN controller and reusability will save you a lot of time. 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

The high level message sending interface will be used by your application, and in principle, it should be designed so general that it is possible to use it on other CAN controllers without any modifications (although it will not happen in this project). 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 chapter 2.2.1.

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 SPI communication.
4. 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
5. Create a driver for CAN communication using the MCP2515 driver. It should have the following features:
    - Initialization of the CAN controller. Use the loop-back 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.
6. Test this driver by sending/receiving some messages (in loop-back mode, you will receive what you sent immediately).

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.
- 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 MCUs used in this project will start transmission when the SPDR register is written to. When the transmission completes you will find the received byte in the same register.
- 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.

3.6 CAN-bus and communication between nodes

This is the exercise where Node 2 will be introduced for the first time. As shown in chapter 1.1, it consists of the Arduino Mega 2560 development kit, an expansion-card and an I/O-card.

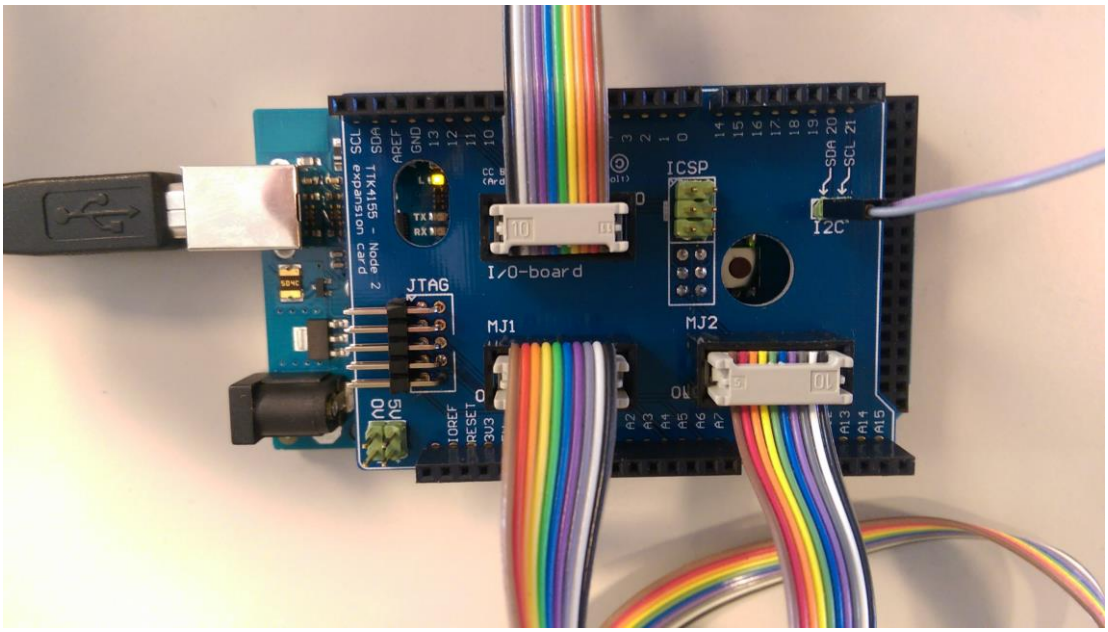


Figure 15 The Arduino 2560 with expansion card

MCP2551

The MCP2551 is a CAN transceiver which serves as the physical interface to the actual wires that constitute the CAN bus. It will be connected between the CAN controller and the wire pairs of the bus.

Arduino USB Communication

The Arduino mega 2560 allows us to use the USB connection as a USB to serial bridge via a separate ATmega chip onboard. This extra chip is the unit seen to the right in the Arduino schematic. The Atmega2560 is connected to the USB bridge through USART0 via PE0 and PE1.

Expansion card

The expansion card sits on top of the Arduino Mega 2560 providing 2x5 headers to simplify the connection between the Arduino, I/O card and motor box. In addition, it allows access to the Atmega2560's ISP and JTAG programming interfaces. Note the markings on some of the headers. The status LEDs and reset-button of the Arduino board are accessible through holes in the expansion card. Please do not remove the card.

I/O card

The I/O card contains the ICs required to complete Node 2, including a CAN interface and a DAC. See the schematic for more information.

Exercise

1. Connect the CAN transceiver MCP2551 on Node 1 using the information provided in the datasheet. You can use the 22 k resistor for slew-rate limiting.
2. Connect Node 2 to the I/O card. Use a ribbon cable from PORTB<->I/O-Board and use a single/smaller cable for MCP2551 interrupts if desired.
3. Connect Node 1 and Node 2 together using the CAN bus, conforming to the “AN228: A CAN Physical Layer Discussion”.
4. It is useful to implement serial communication also on Node 2. This can be done by reusing the RS-232 communication library from Node 1.
5. Test the system again, but now with the CAN controller in normal mode. Node 2 should be able to reuse most of the code generated in the previous exercise.
6. Make a joystick driver that can send joystick position from Node 1 to Node 2 over the CAN bus.

Tips

- 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 I/O card does not have built-in termination.
- See the schematics of the I/O card to understand how the components are connected, and which connectors to use.
- Keep in mind that the SPI and UART is located at different pins on Node 1 and Node 2.
- The USB to serial bridge on the Arduino is used to program it, the DTR (Data Terminal Ready) which signals a new connection in RS232, is connected to the Reset line of the Atmega2560. Therefore, the microcontroller is reset every time you connect to the COM port in Windows.

3.7 Controlling servo and IR

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 averaging 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.

Servo

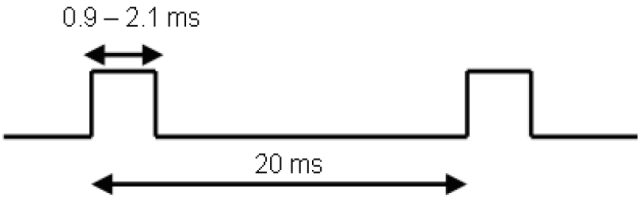


Figure 16 – PWM for servo

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.

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.

Detecting a lost ball – breaking an infrared (IR) beam

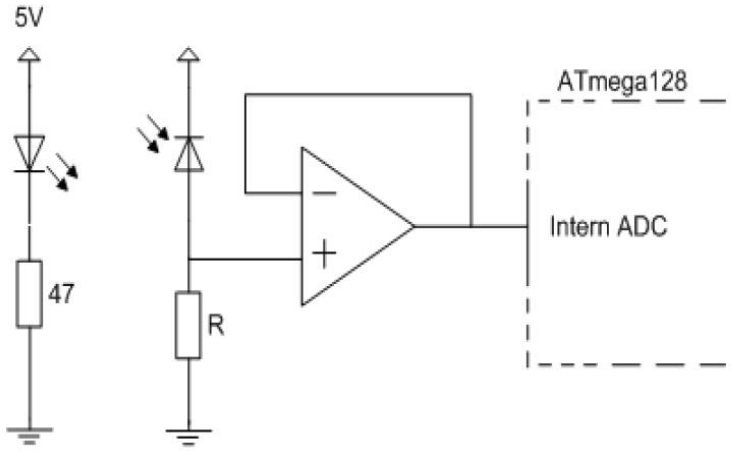


Figure 17 – example of IR assembly

Choose an appropriate value for R based on the photodiodes datasheet.

Exercise

1. The timer/counter module of the ATmega is a versatile module, it can generate timer interrupts, pwm signals and more. Create a driver for the timer/counter module which allows you to use the pwm functionality (fast-pwm). If you also implement the timer interrupt it might save you time when implementing the controller since most of you would 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 time/counter driver. Also implement safety features which will never let the pwm go out of valid range for the servo.
3. Use an oscilloscope to verify that your driver in fact never go outside valid range for the servo (0.9-2.1ms)
4. Connect the servo on the game board to one of the PWM outputs of the Arduino.
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 sidewalls 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 17, and you might also consider implementing an analog filter.
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. The motor will take up some pins of PORTF, but you should have some vacant pins for ADC usage.
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.

Tips

- Read the AVR datasheets for information about how to set up PWM, examples etc. Fast PWM is the mode you most likely want to use, try searching for “fast pwm” in the datasheet if you have a hard time finding the right information.
- If you still are having a hard time configuring PWM take a look at Figure 17-7 in the ATmega2560 datasheet, it explains how the different counter registers interact with each other. Table 17-2 show some of the same info for all the different modes, you probably want to use mode 14.
- 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. A motor interface box will power the motor and control its speed according to a 0-5 V signal provided by the DAC on the I/O card of Node 2. It will also decode the output from the motors quadrature counter. The DAC of type MAX520 will be controlled by the Arduino Mega 2560 using the TWI/I^2C bus.

TWI/I^2C

TWI, also known as I^2C, is a serial master/slave bus for inter-IC communication. It uses 7 bit addressing and thus supports 128 units on the bus. Generally, all nodes may take the role as the bus master and all nodes can both transmit and receive data.

The transmission starts with the node sending the START condition on the bus. If the bus is free the node becomes the master and can transmit its data. Depending on data sent, the receiver (slave) might be requested to send data back to the master. The bus is kept busy by the master during the entire bus cycle. When the transmission completes the master sends a STOP condition which frees the bus for other nodes.

The bus consists of two wires, a clock signal (SCK) and a data signal (SDA), and supports transmission rates of up to 400 kHz. The SCK and SDA pins available on the Arduino Shield.

A driver for this bus can be created in the same manner as we did for UART, SPI and CAN; but in this exercise you are allowed to use a driver from Atmel. The app note containing the driver is called AVR315 and is found on Blackboard.

Encoder

An encoder is an electro-mechanical device that registers the movement of the motor shaft. The motor box decodes the output from this encoder, and provides the opportunity to obtain the number of rotations performed by the motor, and hence the position or speed of the carriage can be computed. The motor box user guide provides more information.

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 a MCU I/O-pin and an intermediary transistor driver is therefore required to operate the relay, as shown in Figure 18. 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.

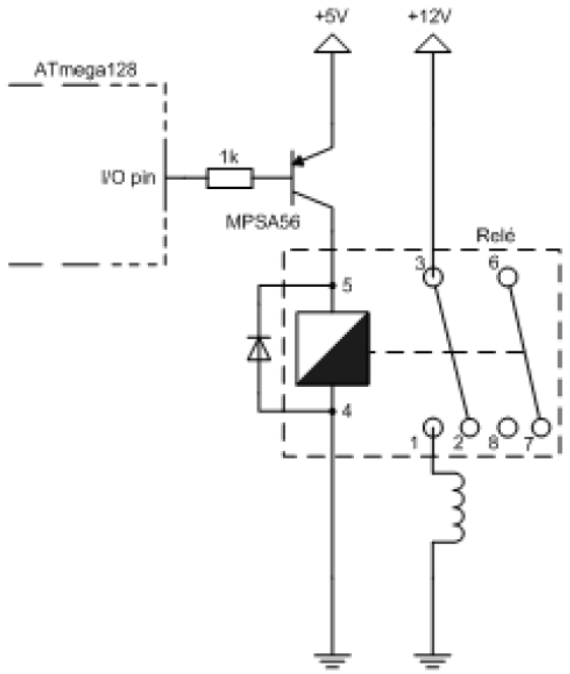


Figure 18 – connecting the solenoid to the ATmega128 (note the pin numbering)

Internally, the relay is connected as shown in Figure 19. 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 5 V to the resistor before connecting the Arduino Mega 2560.

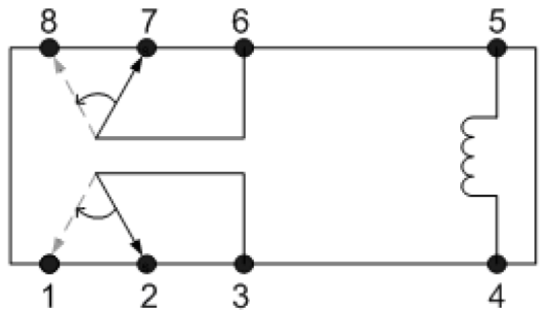


Figure 19 – relay (note the pin numbering)

Controller

A simple yet robust controller is called the PID controller. The continuous form of this controller is given as $u = K_p e + K_i \int e\, dt + K_d \frac{de}{dt}$ 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 $u(n) = K_p e(n) + TK_i \sum_{i=0}^{n} e(i) + \frac{K_d}{T}(e(n) - e(n-1)$, where T is the sampling period. 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. You are free to use a PI, a PID or some extensions of these, however the D part is generally not needed so most groups end up using a PI controller.

Exercise

1. Connect the Arduino Mega 2560 expansion shield to the motor interface box's MJ1 and MJ2 ports with 10-lead flat cables.
2. Connect the DAC output to DA1 on the MJEX port. Make sure you have a common ground connection somewhere, preferably through the MJEX pin.
3. Connect the encoder input to the motor encoder by using the designated cable. Both the encoder and cable is marked with a white spot, which should be at the same end. Also connect the motor voltage supply, M+ and M-, to the motor.
4. Browse Atmel's application notes, download and integrate the TWI driver in your code.
5. Develop code so that the motor speed can be controlled by the joystick's position (open loop), together with the control of the servo. You should now be able to play the game, although it is hard to control.
6. Create a position/speed controller (closed loop, using feedback from the encoder) for the motor so that the position/speed is easier to control. A PI controller should be sufficient. Exact tuning is not important, but the game must be playable with the controller.
7. 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.

Tips

- Read the quadrature decoder often enough to prevent it from overflowing.
- 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, and greasing the rail with oil.
- 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.
- Check out AVR221 for an app note on implementing a discrete PID controller on an ATmega

3.9 Completion of the 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 prepare for the evaluation as described in chapter 1.2.8. In addition, if there is time left you might work on extra features to get a higher score at the evaluation.

Some suggestions for gaining extra credit:

- Documented code (e.g. UML)
- Connecting two boards for multi-player operation
- Creative use of the display
- Online tuning of regulator
- Writing to the display over RS232/SPI/CAN -> AT90USB1287 -> OLED
- Dual buffering display data in SRAM
- Utilizing more functionality on the USB multifunction card. Like RZ600 Wireless or CAN.

Examples of extensions that have been implemented earlier:

- Replay: all motor inputs saved to SRAM, so that the game can be replayed
- Sound: sound effects or music can be implemented using PWM and a speaker
- 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
- http://www.youtube.com/watch?v=83D4P8sokFc
- http://www.youtube.com/watch?v=cdPKJv5O21g

It might 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! :)