ODENSE
UNIVERSITY COLLEGE
OF ENGINEERING

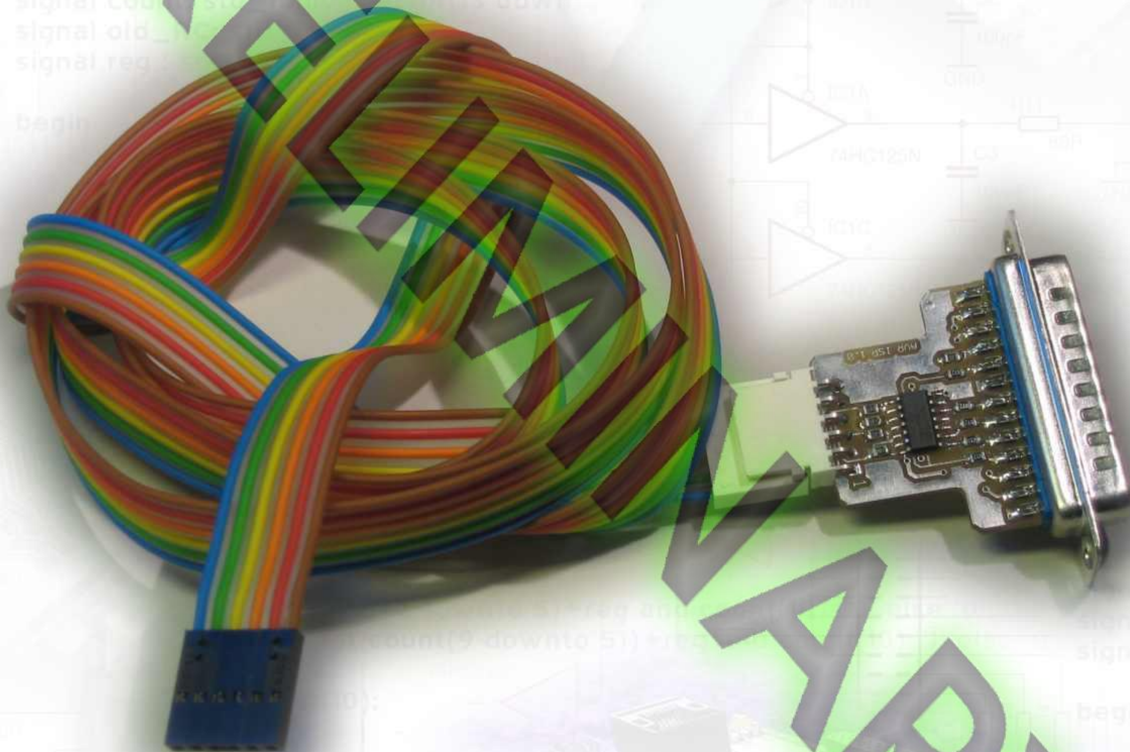
AVR IN SYSTEM PROGRAMMER

USERS GUIDE

Anders Stengaard Sørensen

November 8, 2004

PRELIMINARY



The AVR **I**n **S**ystem **P**rogrammer (ISP) of Odense University College of Engineering (IOT), enable you to program AVR micro controllers, via the parallel port of your PC. We have based our AVR-ISP on the popular **STK-200** interface, which can be used with all available AVR download-software. This document describes the design, the functionality, and how to assemble and use our AVR-ISP.

Anders Stengaard Sørensen

Contents

1 Introduction **5**

2 Theory of operation **5**
- 2.1 The interface hardware . . . 7
- 2.2 The connector and cable . . . 10

3 Assembling the AVR In System Programmer **11**
- 3.1 Identifying the PCB . . . 11
- 3.2 Identifying the components . . . 12
- 3.3 Mounting the components . . . 13
- 3.4 The cable . . . 14

4 Using the AVR-ISP **15**
- 4.1 Connecting to the micro controller . . . 15
- 4.2 Connecting with PonyProg . . . 16
- 4.3 Connecting with uisp . . . 17

5 Pitfalls and common problems **19**
- 5.1 BIOS configuration of the parallel port . . . 19

A Schematic **20**

B PCB layout **21**

C Bill of materials **22**

D Component placement **23**

E Test circuit with an ATmega8 **24**

F Litterature **26**

WARNING!

Connecting external circuits directly or indirectly to the parallel port may cause damage to your computer, if the external circuit it is not properly designed and tested. This is especially true if the external equipment operate with negative voltages, or voltages in excess of 5V.

Although AVR-ISP is designed to protect your computer against excessive external signals, neither Odense University College of Engineering, nor it's employees can take any responsibility for damage caused to your computer when using the AVR-ISP programmer.

You use the AVR-ISP entirely at your own risk, so be careful!

Copyright notice

Everyone can copy and/or use the AVR-ISP design presented here, in any way they see fit. You are also welcome to copy and distribute this document in its entirety, or to use text and figures from it, provided you include a proper reference to the original document and author.

About the HOPE projects

Hands **O**n **P**rogrammable **E**lectronics — or HOPE , is a series of projects, aimed at promoting the use of programmable electronic components in research, development and students projects, related to Odense University College of Engineering.

While it is good educational practice to teach classical electronic design, based on discrete components and simple integrated circuits, it is also necessary to enable students to gain practical experience with the highly flexible and complicated devices used in practical electronics today.

As I began teaching in 2003, I was surprised to see the complex circuits students were designing with 74.. and 40.. type IC's, to realize registers, counters, decoders and other small digital systems, that could be realized much easier (and cheaper) in a Programmable Logic Device (PLD) or even in a micro controller. I was even more surprised to learn that most of the students had actually followed courses in PLD's and micro controllers, but thought it too abstract or troublesome to transfer their experience with PLD or micro controller demonstration systems to a practical design in its own contexts.

In order to reduce the *entry barrier* towards programmable electronics, I have initiated a number of small projects, resulting in a series of tools, that should make it easier to begin working with selected PLD's, micro controllers etc. I have launched these projects under the common title *Hands On Programmable Electronics*, with subtle reference to the first commandment of the Hacker Ethic:

> *Access to computers — and anything which might teach you something about the way the world works — should be unlimited and total. Always yield to the Hands-On Imperative!* *(MIT students* ~ 1960*)*

It is my HOPE that the tools provided by the HOPE projects will result in increased use of CPLD's, micro controllers, FPGA's, FPAA's and other programmable electronics in students projects, as well as R&D projects in corporation with Odense University College of engineering.

Anders Stengaard Sørensen — 2004

1 Introduction

The AVR-ISP programmer described here will enable you to work with most of Atmels AVR micro controllers, using the parallel port of your computer, and a *download program* such as uisp or Pony Programmer.

I have chosen to include tools for the AVR micro controller in the HOPE program, for the following reasons:

- Atmel has a large inventory of AVR processors, which cover a broad area of applications.
- AVR processors are relatively powerful, inexpensive, easy to use and easy to come by.
- There exist a large user community, promoting free software, reference designs, discussion groups etc. on the Internet.
- Due to previous AVR projects in the local community, we can utilize existing experience.

During 2003 and 2004, I have used the ATmega8 and ATmega128L, in our summer course *Design and construction of a robot car*. Replacing the MC68HC11 used previously, the AVR processor proved much easier to use, as the students had working micro controllers up and running 1 or 2 days into the course.

You will probably be assembling the AVR-ISP as part of one of your first projects with AVR processors, very likely planning to use it with an ATmega8 or ATmega128, which we usually recommend — and keep on stock — for students projects at Odense University College of Engineering.

If you follow the guidelines and reference designs given below, you should have a micro controller up and running later to-day.

2 Theory of operation

Most modern programmable devices offer the possibility of *in system programming*, allowing the user to up- and download configuration, program code and data while the device is mounted in a circuit. Many different schemes exist for in system programming, some are specific for a single type of devices, while others follow international standards like JTAG.

The AVR micro controllers we use at Odense University College of Engineering (IOT[1]), offer in system programming through a scheme, known as AVR-ISP, which — to my knowledge — is specific for AVR micro controllers. AVR-ISP is thoroughly described in Atmels *AVR910 - In-System Programming* application note [1], but it can be briefly summarized as a standard 3-wire synchronous serial SPI interface, overlaid by a RESET signal and a power supply, resulting in the 6-wire interface shown in figure 1

[1] In Danish: "Ingeniørhøjskolen Odense Teknikum"

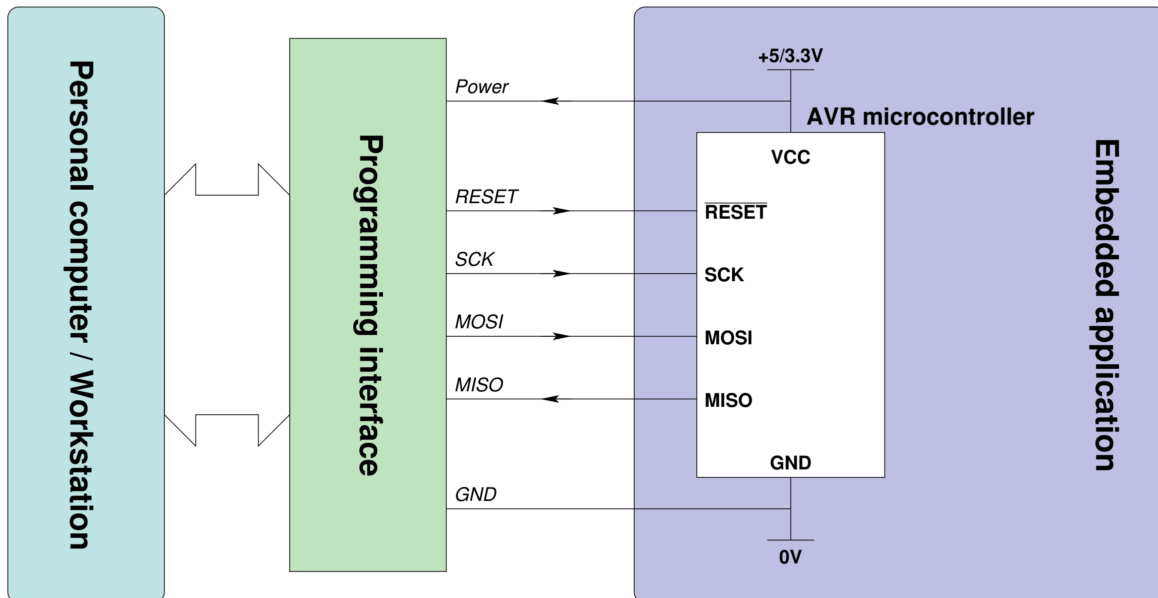


Figure 1: The programming interface and its connection to the micro controller

The function of each of the 6 signals are:

Power: Is used to supply the programming interface. By using the same power supply as the micro controller, it is ensured that the programming interface and the micro controller will always have compatible signal levels, so the same programming interface can be used with both 5.0V and 3.3V micro controllers.

RESET: Is used by the programming interface to place the micro controller in it's *reset* state, when communicating with it.

SCK: (*Serial ClocK*) is the master clock the programming interface use when communicating with the micro controller. One bit is passed on MISO / MOSI for each clock cycle on SCK, thus the term "synchronous serial interface".

MOSI: (*Master Out, Slave In*) is the signal used to pass bits from the programming interface — which is the master — to the micro controller.

MISO: (*Master In, Slave Out*) is the signal used to pass bits from the micro controller — which is the slave — to the programming interface.

GND: Is the zero-reference for the signal lines, and 0V for the power supply.

With the hardware of the 6-wire AVR-ISP interface in place, up- and downloading code and data, is simply a matter of reading and placing the right bits on the synchronous serial interface in the right order. This is accomplished in cooperation between the programming interface and the programming software running on the PC / workstation. Relevant protocols and algorithms for this is described in e.g. application notes AVR910 [1] and AVR911 [2], but usually we leave the details to off the shelve programming software.

2.1 The interface hardware

There are many different approaches to realizing an AVR programming interface, and thus a multitude of different interfaces exist in the AVR community. Most of the difference have to do with the way the programming interface connects to the PC, and which types of micro controllers they can program.

I have chosen an implementation which is both very simple to build, while it is compatible to one of the most popular programming interfaces in the community — the STK-200. This have resulted in a small and simple circuit, which can be assembled in an hour or less, and which can be used with all available programming software.

The STK-200 type interface, utilizes the parallel (printer) port of a PC, using one pin in the parallel port for each digital signal in the AVR-ISP interface, and letting the programming software control the pins of the parallel port in the correct sequences.

In principle, the parallel port of a PC can be connected directly to an AVR micro controller, but having a simple circuit in between offer some important benefits:

- Level shifting between the 5V TTL compatible parallel port, and an AVR micro controller running on other voltages than 5V.
- Suppression of signal reflections in the cable, due to impedance mismatch in the signal paths.
- Protection of the PC's parallel port against minor over voltages, spikes etc.

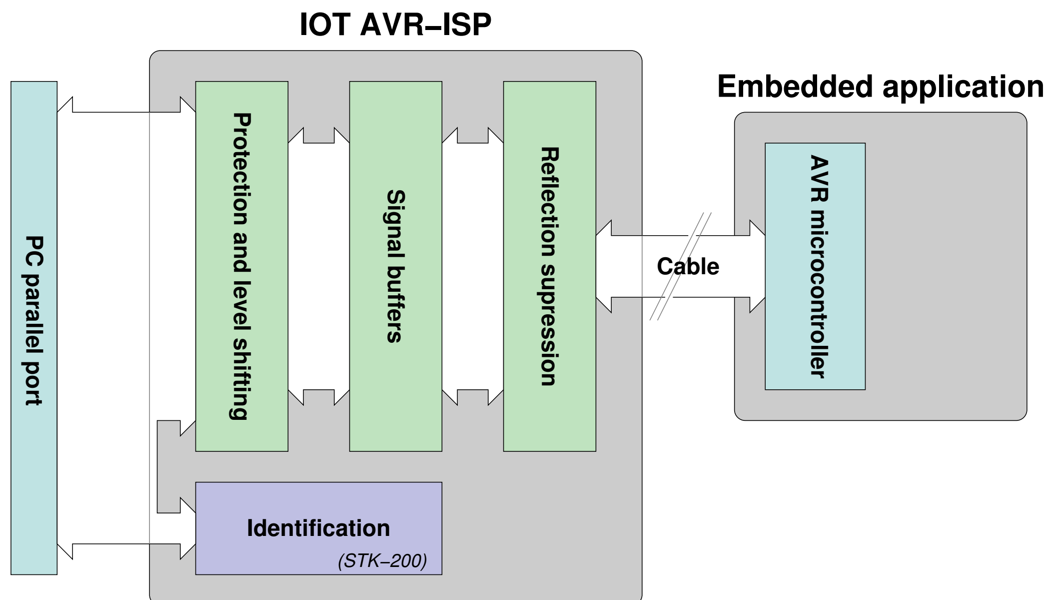


Figure 2: Block diagram of the IOT AVR-ISP

The block diagram of the IOT AVR-ISP, shown in figure 2, indicate the overall structure of the programming interface, which consist of four major blocks:

- Identification
- Protection and level shifting
- Signal buffers
- Reflection suppression

For a detailed schematic of the interface, please refer to figure 12 in page 20

2.1.1 Identification

In order to allow the programming software on the PC to check if the correct programming interface is connected to the PC, the interface identifies itself by drawing pin 15 to logic 1, and by connecting pin 2 to pin 12 and pin 3 to pin 11. These connections allow the programming software to check if an STK-200 type interface is connected to the parallel port, by checking if the input pins 11 and 12, follow the level on the output pins 3 and 2. It can also determine if the programming interface is powered, by checking the status of input pin 15.

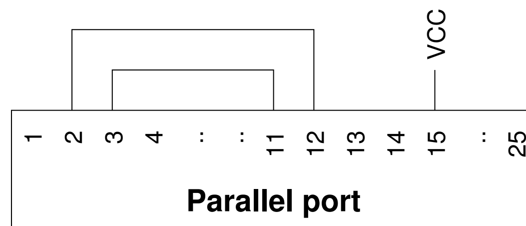
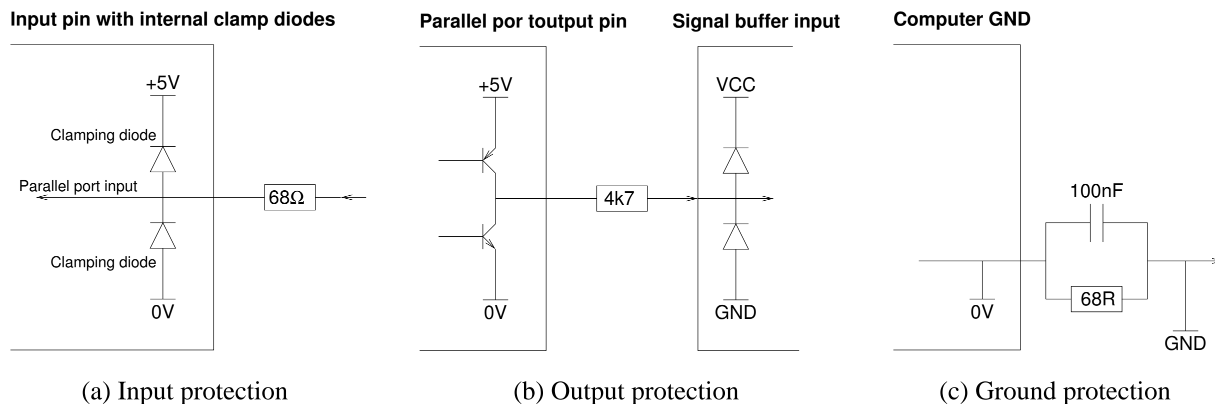


Figure 3: Identification the programming interface as an STK-200 on the parallel port

2.1.2 Protection and level shifting

In order to protect the parallel port against minor over voltages, due to errors in the embedded application, none of the parallel port pins are connected directly to a signal which is in contact with the embedded application.

All input pins on the parallel port is connected to the rest of the programming interface through 68Ω resistors (R1 and R8). As the parallel port inputs are protected by clamping diodes as sketched in figure 4(a), the resistors will protect the inputs from over voltages up to $I_m \times 68\Omega$, where I_m is the maximum current the clamping diodes can withstand. Assuming the diodes can handle a continuous current of approximately $50mA$, the resistors will enable the circuit to tolerate voltages in the range of approximately -3.5V to +8.5V, without causing damage to the parallel port. The level of protection might be increased by increasing the value of the resistors,



(a) Input protection

(b) Output protection

(c) Ground protection

Figure 4: The electrical protection of the parallel port

but if the resistors become too high, the input signals might deteriorate and cause transmission errors. The value of 68Ω is chosen because it is also used in other parts of the circuit, and I would like to keep the number of different components to a minimum. The parallel port input levels are compatible to both 3.3V and 5V CMOS output levels, so no level shifting is necessary for the parallel port inputs.

All output pins on the parallel port are connected to signal buffer inputs through $4.7k\Omega$ resistors (R3, R4, R5, R6 and R7), as shown in figure 4(b). The resistors provide ample protection for the output pins, in the unlikely case the signal buffers should become low impedance (defective). Assuming the parallel port outputs can withstand 10mA continuous current, the resistors will make the outputs tolerate low impedance connection to a voltage in the range $\pm 45V$. The $4.7k\Omega$ resistor also provide level shifting in cooperation with the input clamping diodes of the signals buffers, effectively shifting the 5V TTL compatible outputs down to an acceptable level if the programming interface is powered by a voltage lower than 5V.

In order to further reduce the risk of excessive currents flowing through the parallel port in case of accidents, even the ground reference is wired through a 68Ω resistor (R13), in parallel with a $100nF$ capacitor (C5), as shown in figure 4(c). The resistor inhibit current from flowing between the parallel port and the programming interface, and will even act as a fuse if the current approaches 100mA. The capacitor will allow high frequency currents to pass uninhibited between the PC and the programming interface, maintaining the integrity of the flanks of the pulses traveling between PC and programming interface.

2.1.3 Signal buffers

Passing the signals through a set of buffers, play a role in adapting the logic levels, as the output voltages of the signal buffers can not exceed its supply voltage, which is drawn from the same supply as the micro controller. The buffer also maintain signal integrity even though the signals

are passed through resistors in order to provide protection against excessive currents in the face of accidental wrong connections etc.

The buffer used in the IOT AVR-ISP is the 74HC125 *High speed CMOS quad three state buffer* (IC1). The use of *high speed CMOS* (74HC) technology ensure compatibility with supply voltages from 3V to 5V, and **the 74HC125 should not be substituted with other logic families** such as the more popular 74LS or 74HCT.

The use of three-state buffers allow the programming software to disable the signals connected to the micro controller, when it is not being accessed, so the programming interface can remain connected to the micro controller after programming, even if the programming pins of the micro controller is used for I/O.

In order to protect the programming interface from wrong connections, the 74HC125 is protected from wrong polarization of the power supply, by routing the power supply through a diode (D1). D1 is a shottky diode which will introduce a voltage drop of approximately $0.3V$

2.1.4 Reflection suppression

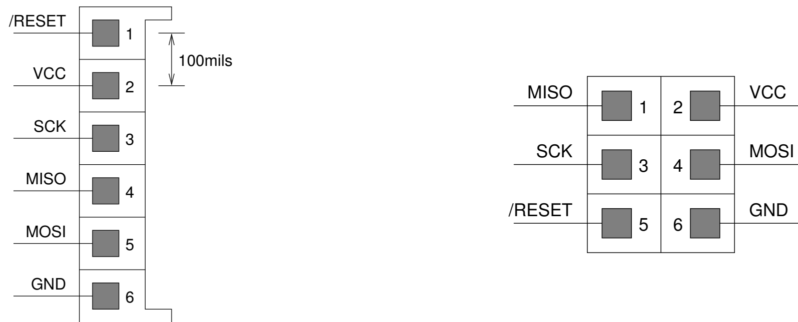
As it is impractical to implement impedance matching to the programming cable near the micro controller, the programming interface is designed to suppress reflections on the cable using simple RC stages (R9/C2, R11/C4 and R10/C3) that will absorb most of the high frequency content of reflected flanks.

2.2 The connector and cable

In the application note *AVR910: In-System Programming* [1], Atmel recommend that the 6 wires connecting the programming interface to the micro controller, should be configured in a 2×3 pin header with 100mil spacing.

Instead of following Atmels recommendation, I have equipped the IOT AVR-ISP with a 1×6 pin header. The main reasons for this is to be compatible with prior AVR designs in the local community, because 2×3 IDC (ribbon cable) connectors are hard to come by, and because it is easier for most students to mount a 1×6 non-ribbon-cable connector than a 2×3.

If it should become necessary to connect to an embedded application with an Atmel type layout, it will be quite easy to make a cable with an IOT layout in one end, and an Atmel layout in the other.



(a) IOT connector layout

(b) Atmel connector layout

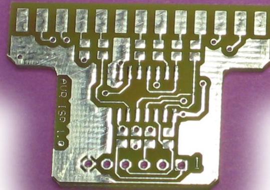
Figure 5: Common connector layouts

3 Assembling the AVR In System Programmer

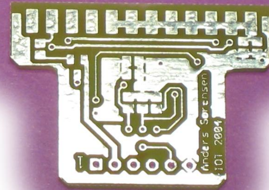
This section refers to version 1.0 of the IOT AVR-ISP, which is currently the only existing version.

3.1 Identifying the PCB

The PCB for the AVR-ISP, is polygonal, with a 37×27mm outline, designed to fit inside common shields for 25-way sub-D connectors. Figure 6 show a photo of both sides of the PCB. The top side — also referred to as the component side — marked by the text **'AVR ISP 1.0'**, while the bottom (solder) side, is marked with the text **'Anders Sørensen - IOT 2004'**. The exact layout of the PCB is available in figure 13 on page 21



(a) Top (component) side



(b) Bottom (solder) side

Figure 6: The PCB

3.2 Identifying the components

All the components used in the AVR-ISP, except the cable, are shown with the photograph in figure 7. A detailed list of the components can be found in table 1 on page 22

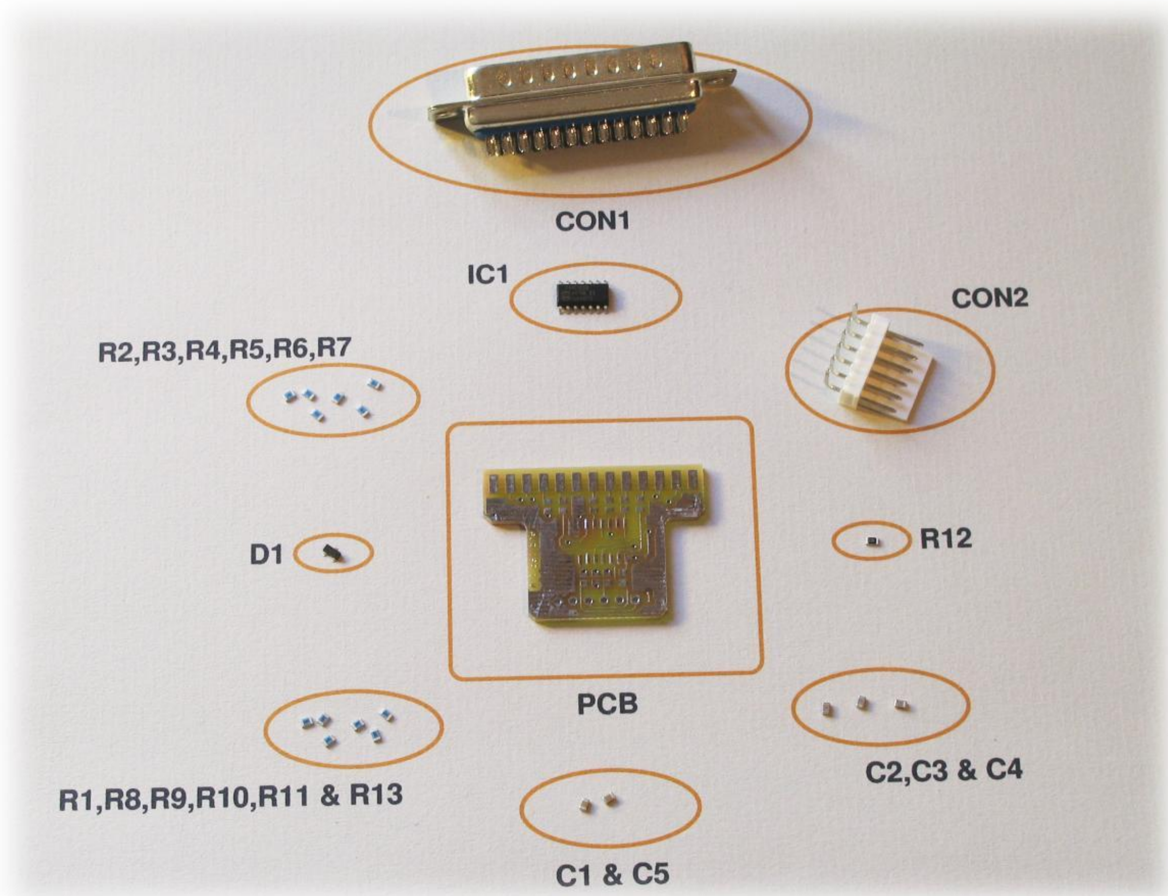


Figure 7: Overview of components

The two connectors should be easily identified from figure 7, check that CON1 is a *plug* type (with 25 small pins), which fit the *socket* type connector (with 25 small holes) in the back of your computer.

IC1 is also easy to identify, as the text `74HC125D` should be printed clearly on top of it.

The resistors can be identified by close inspection, as their values are printed on them in fine print — you may have to use a magnifying glass. 68Ω is printed as `68R0`, 4.7kΩ as `4701`, and 100kΩ as `1003`.

As the value of capacitors is usually not printed directly on the components, there is no direct way of identifying the capacitors once they have been removed from their storage. The capacitors

can only be identified from the printing on the container that stored them. Be careful not to mix the capacitors after they have been removed from their storage.

D1 can not be uniquely identified from the naked component as the component is not large enough to hold its entire name. Instead the diode is marked with the code `D96`. Be sure to identify the diode from the printing on its container.

3.3 Mounting the components

If you have no experience with mounting **s**urface **m**ount **d**esign (SMD) components, you might want to practice your skills by soldering a few spare resistors between the pads of a piece of prototyping PCB, or ask someone with more experience for a demonstration. Once you have practiced a bit, it shouldn't be too hard to master, using thin solder and a set of pointed tweezers to hold the components.

- Start by soldering one pin while you hold the component in place with the tweezers.
- Let go of the component.
- Solder the remaining pins of the component.
- Re solder the first pin to remove any mechanical stress caused by force or vibration while you held it with the tweezers.

All the components are either bidirectional, or asymmetrical, so they can not be turned the wrong way. One possible exception is IC1, as SMD IC's are not always marked near pin 1. Please refer to the data sheet of the specific version of 74HC125D you use, if you have any doubt about the pin placement or direction.

The position of components can be seen in figure 14. The mounting sequence is arbitrary, but I find it easier to begin with IC1, continue with all the SMD components on the top side, move on to all the SMD components on the bottom side, and finish with CON2 and CON1.

Note that the PCB slides in between the two pin-rows of CON1, with 13 pins on the top side and 12 pins on the bottom side. CON2 should be mounted on the top side, and it is a god idea to cut off the pins sticking out on the bottom side after mounting it.

Remember that IC1 and D1 can be vulnerable to static discharges. Avoid buildup of static charges while working with the components — do not rub your shoes on the floor, and remember to touch the metal frame of your table before you start working to get rid of any static charge. If you have access to a proper antistatic workbench with a wristband, that is even better.

3.4 The cable

The cable connecting the AVR-ISP to the circuit board containing the micro controller should simply have a 1×6 connector in each end, connecting pin 1 to pin 1, pin 2 to pin 2 etc. As indicated by figure 7, I propose to use a MOLEX type connector for the AVR-SIP, as MOLEX connectors offer both friction lock and mechanical polarization. The cable could simply be soldered directly to the PCB, without using a connector, but wires soldered directly in a PCB are prone to metal fatigue, so a connector offer much better mechanical stability. Direct soldering can only be recommended if you mount the AVR-ISP in an enclosure that ensures that the cable wires can not move with respect to the PCB.

In many cases, a simple pin header without friction lock and polarization will be used on the micro controller PCB, as it is more compact than a MOLEX connector, so you should consider using a narrow pin-header female connector in the micro controller end of the cable, as shown in the picture on the front page. Using a connector without polarization is not a problem, as it is harmless to the micro controller and AVR-ISP to turn the cable the wrong way — you will just not be able to communicate with the micro controller.

There are no special requirements for the cable, and no need to use e.g. shielded cable. The simplest way is to twist or braid 6 single wires, or to use a 6-way ribbon cable. Using double spacing (100 mils[2]) ribbon cable as shown in the front page, is probably the most elegant and robust solution.

The individual terminals used in MOLEX- and other types of pin-header connectors are designed to be crimped onto the wires, using a special crimp tool. Although they can be mounted using a set of pliers or by soldering, using the correct crimp tool offer much better mechanical and electrical integrity, drastically reducing the risk of cable malfunction over time.

[2] 2.54mm

4 Using the AVR-ISP

This chapter will give an introduction, that should allow you to make the proper electrical connections to an AVR micro controller, and use one of the popular download-programs to connect to the micro controller.

4.1 Connecting to the micro controller

One of the major benefits of AVR micro controllers is the simplicity of the external circuitry needed to use them. One of the simplest possible circuit to demonstrate an AVR micro controller is shown in figure 8, and contain:

- The micro controller.
- A decoupling capacitor.
- A pull-up resistor to keep $\overline{\text{RESET}}$ high.
- A 6×1 connector/pin-header for programming.
- A Light emitting diode (LED) with a current limiting resistor — so we can see the micro controller doing something useful, like blinking with the LED.
- A voltage supply, fitting the micro controller (5/3.3V).

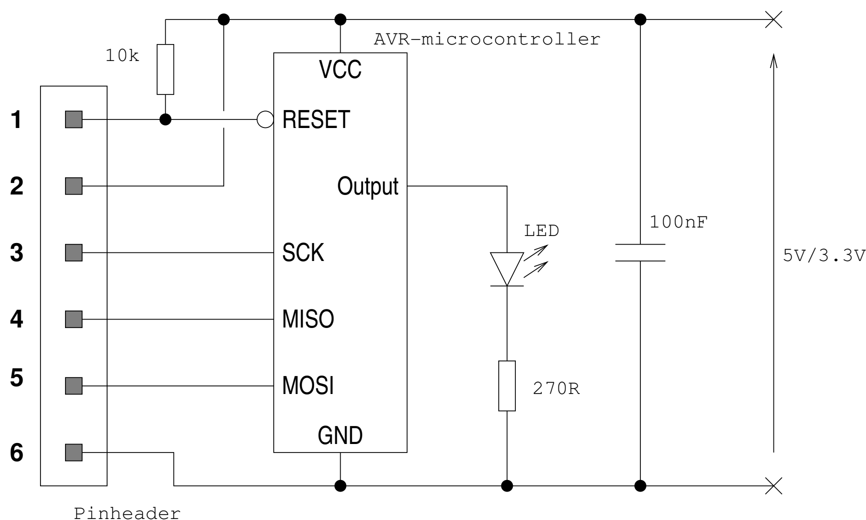


Figure 8: A simple test circuit with an AVR micro controller

An example such a test circuit, based on an ATmega8 can be seen in figure 15

4.2 Connecting with PonyProg

One of the most popular download programs in the AVR community is PonyProg [3] of Lanconelli Open Systems. PonyProg is a generic download program, intended for use with many different programmable devices that use serial communication, among which are the AVR micro controllers.

PonyProg versions for Linux as well as windows can be downloaded from PonyProgs Internet site [3], and are easily installed on a PC.

After installing PonyProg must be set up for:

- Using the parallel port
- Using the STK-200 type interface (*AVR ISP I/O*)
- The type of device used (*AVR micro*)
- The specific AVR micro controller you use (eg. *ATmega8*)

Before using the interface, it must also be calibrated using the *calibration* option in the *Setup* menu.

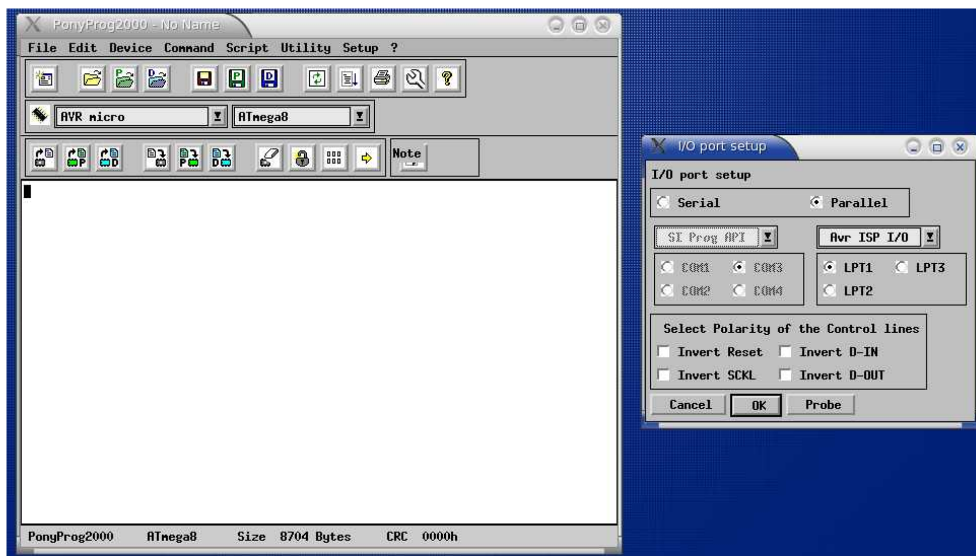


Figure 9: Setup for the IOT AVR-ISP

If you don't yet have a program to download, and your micro controller is still empty, a convenient way to test if you can connect to it, is to read the configuration and security bits of the micro controller.

Use the *Configuration and security bits* entry of the *Command* menu to get the window shown in figure 10. Then use the *Read* button to read the bits from the micro controller.

If the AVR-ISP works properly, you should see two small *status* windows pop up and vanish in rapid secession, to be replaced by the *Configuration and Security bits* window as shown in figure 10.

If the AVR-ISP is not working or not connected properly, you will get an *Alert* stating that: *Device missing or unknown device (-24)*.

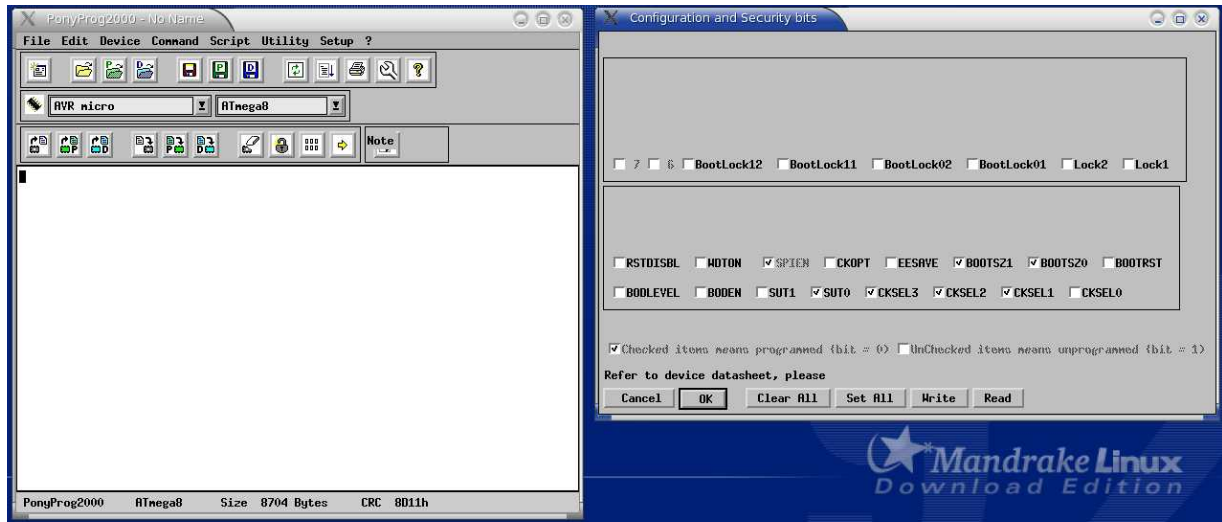


Figure 10: Looking at the micro controllers configuration and security bits

Figure 10 show how the bits are configured in an unused ATmega8 processor. **You shouldn't change** (*write*) the configuration and security bits **unless you know what you are doing**, as a wrong configuration of e.g. the clock bits may disable the internal oscillator, taking the programming interface offline, until an external clock source is connected.

Once you have verified that PonyProg can communicate with your micro controller, it should be straight-forward to program the micro controller with the `.hex` files that is generated by your compiler/linker. If you need the source-code for an example program, figure 16 shows a program for a blinking LED.

4.3 Connecting with uisp

While PonyProg is clearly intended for people who prefer graphical point and click user interfaces, uisp is intended for people who prefer a command oriented interface, in order to acess software tools from a command prompt or a Makefile.

UISP is developed as free software at the uisp development website: http://savannah.nongnu.org/projects/uisp. The source code can be downloaded from the website and compiled on your local computer, or compiled binaries can be downloaded from e.g. http://cdk4avr.sourceforge.net/

Although uisp may seem modest in comparison to PonyProg, it is at least as powerfull when it comes to programming AVR micro controllers. You can learn all about the features of uisp by using it's integral manual, which can be summoned by giving the command: `uisp --help`

In order to use uisp with our STK-200 compatible AVR-ISP interface, using the normal parallel ports, you can run uisp with the options (assuming the parallel port of your computer is located at address 0x378)

```
uisp -dlpt=0x378 -dprog=stk200
```

In order to simply check the connection between your computer and an AVR micro controller, you can use the uisp command to read the configuration and security bits, also known as *fuses*. You do this with the command:

```
uisp -dlpt=0x378 -dprog=stk200 --rd_fuses
```

The result should look similar to figure 11

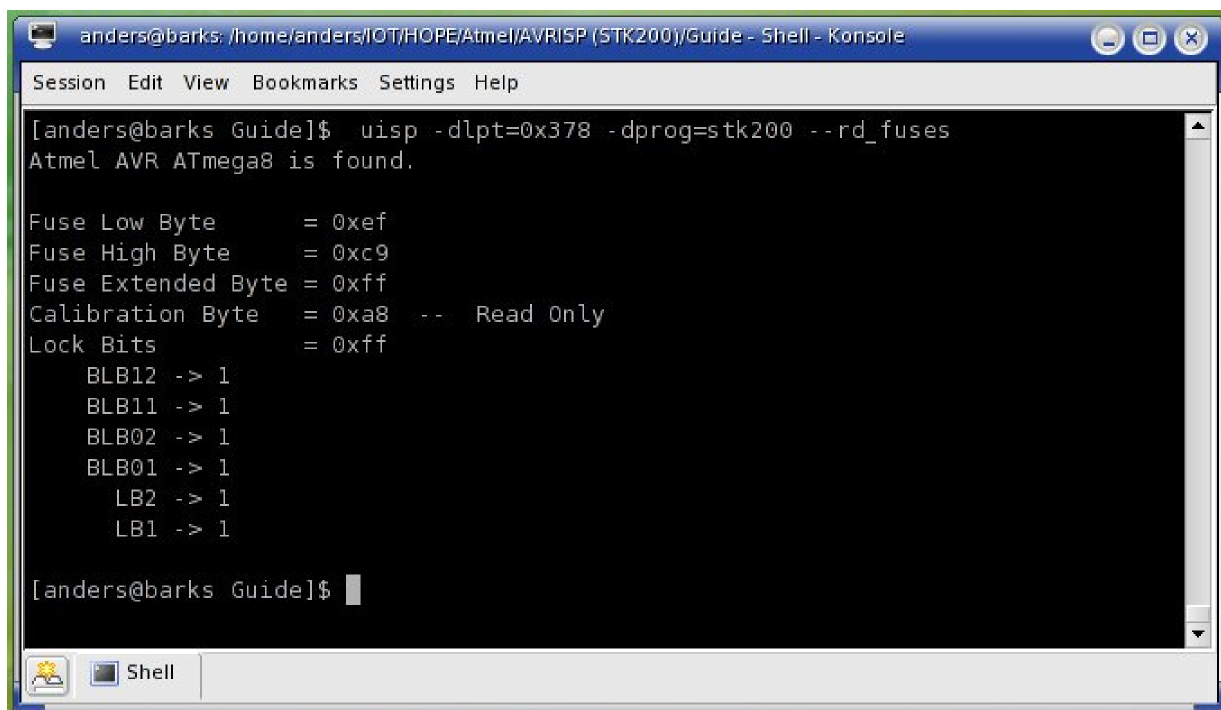


Figure 11: Looking at the micro controllers *fuses*

Naturally, uisp can also up- and download binary (.hex) files. This is done with commands like:

```
uisp -dlpt=0x378 -dprog=stk200 --upload if=program.hex
uisp -dlpt=0x378 -dprog=stk200 --download of=flashdump.hex
```

As mentioned above, uisp is an excellent tool when using Makefiles. A sample makefile, using uisp, can be found in figure 17.

5 Pitfalls and common problems

In this section I describe some of the pitfalls and common mistakes that have previously been encountered by users of the AVR-ISP. If you encounter a pitfall or make a mistake that could have been avoided with better documentation, please send me a note so I can update the next revision of this document.

5.1 BIOS configuration of the parallel port

If you can't connect to the micro controller, and you are fairly certain that you have assembled and connected AVR-ISP correctly, you should determine if the problem is with your computer, the AVR-ISP or the micro controller. You can do this by exchanging each of them with a counterpart that is known to work, e.g. by swapping gear with someone that already has a working system.

If you can determine that the problem is not with the AVR-ISP or micro controller, but rather with your computer, there is a good chance that your BIOS setup for the parallel port is wrong.

Enter the BIOS setup, and verify that the parallel port is configured for bidirectional communication. If the port have several different settings, you might have to try them in turn until you can reach the micro controller.

A Schematic

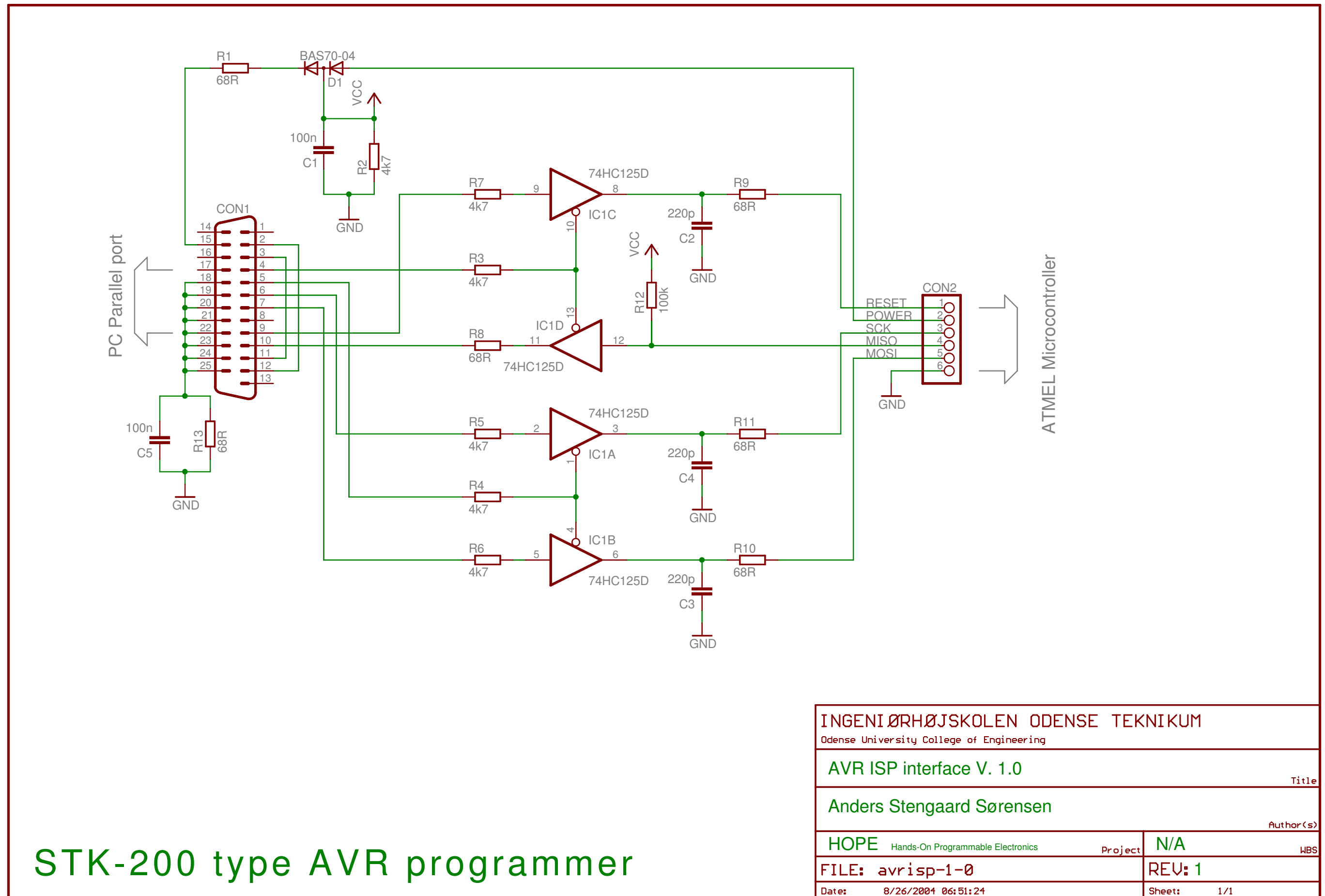
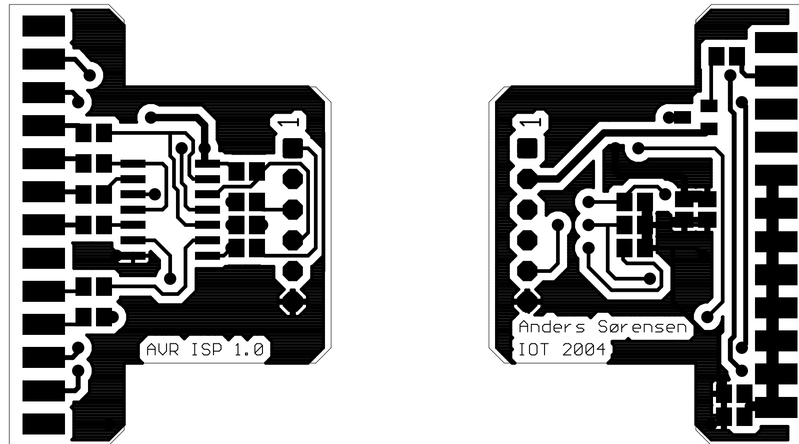


Figure 12: Schematic of the IOT AVR-ISP

B PCB layout



(a) Top (component) side

(b) Bottom (solder) side

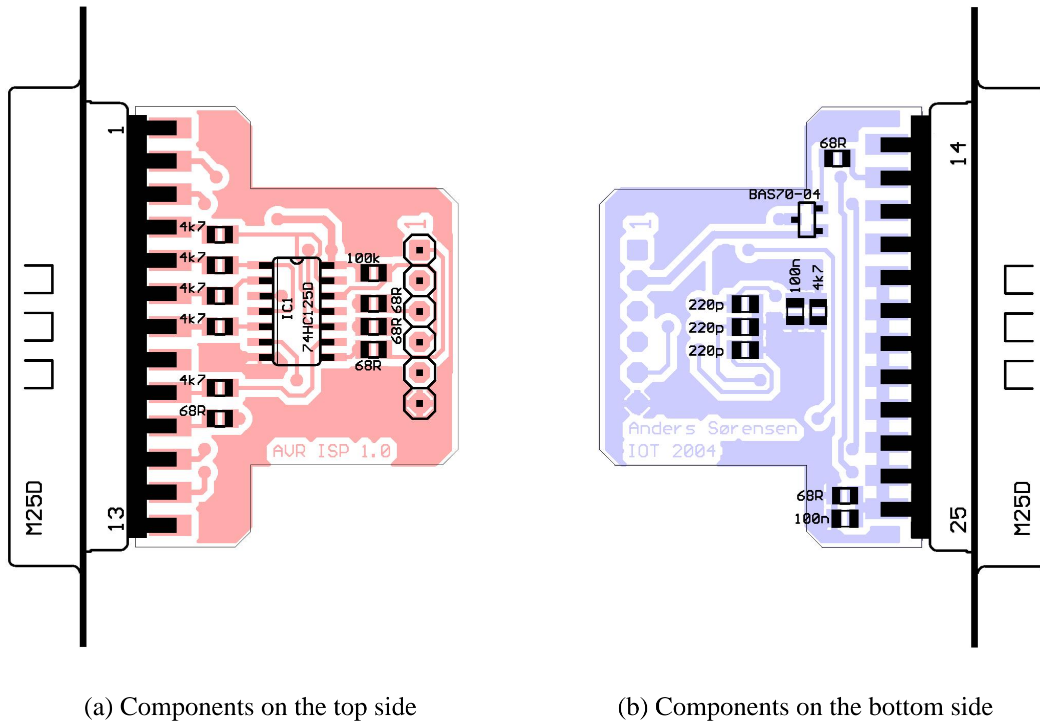
Figure 13: PCB layout of the AVR-ISP

C Bill of materials

Part	Value	Package	Description
IC1	74HC125D	SO14	Quad bus BUFFER, 3-state
D1	BAS70-04	SOT23	Silicon Schottky Diodes
C1	100nF	0805	CAPACITOR
C2	220pF	0805	CAPACITOR
C3	220pF	0805	CAPACITOR
C4	220pF	0805	CAPACITOR
C5	100nF	0805	CAPACITOR
R1	68Ω	0805	RESISTOR
R2	4.7kΩ	0805	RESISTOR
R3	4.7kΩ	0805	RESISTOR
R4	4.7kΩ	0805	RESISTOR
R5	4.7kΩ	0805	RESISTOR
R6	4.7kΩ	0805	RESISTOR
R7	4.7kΩ	0805	RESISTOR
R8	68Ω	0805	RESISTOR
R9	68Ω	0805	RESISTOR
R10	68Ω	0805	RESISTOR
R11	68Ω	0805	RESISTOR
R12	100kΩ	0805	RESISTOR
R13	68Ω	0805	RESISTOR
CON1	M25D	SUB-D	25 way sub-D, plug
CON2	PINHD-1X6R	PIN HEADER	MOLEX

Table 1: Bill of materials

D Component placement

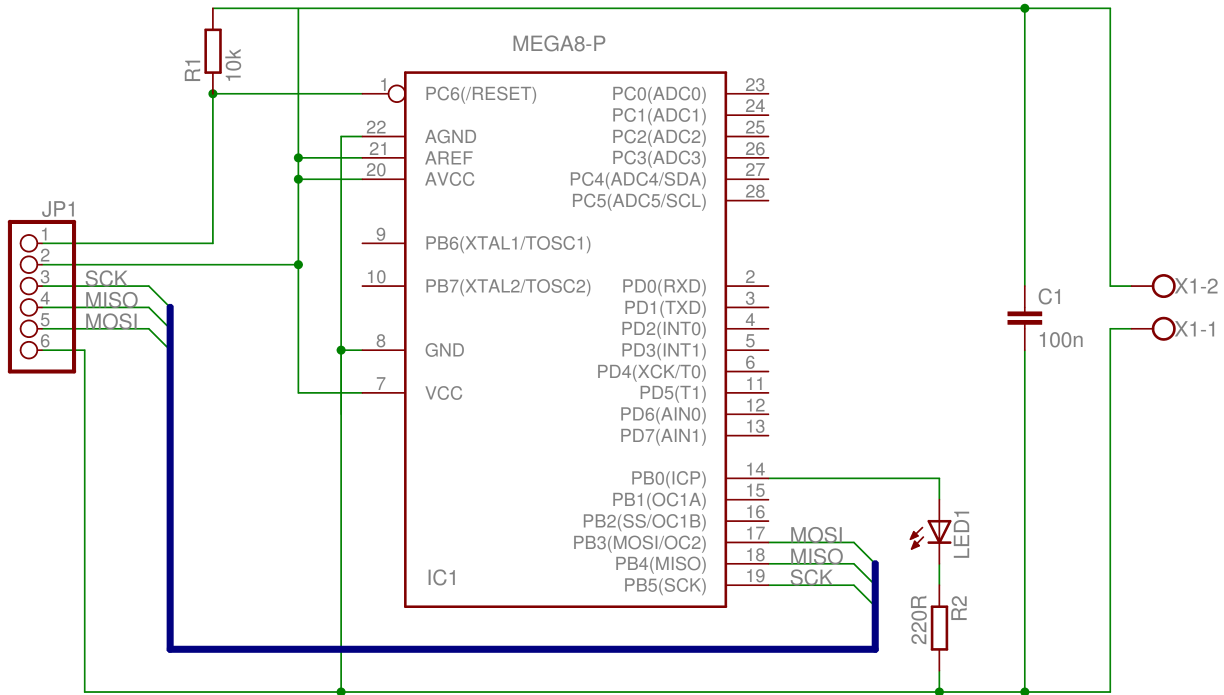


(a) Components on the top side

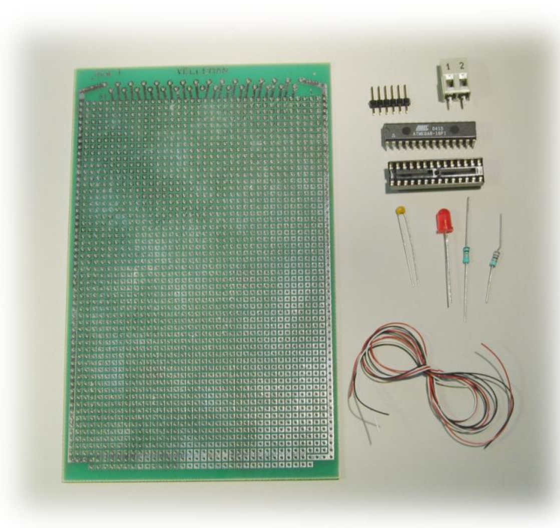
(b) Components on the bottom side

Figure 14: Component placement

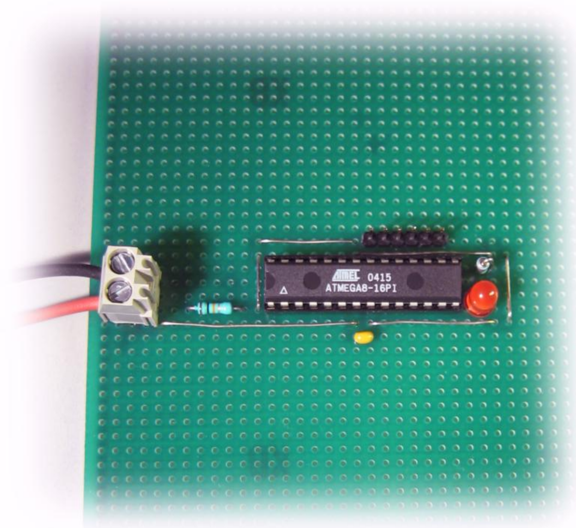
E Test circuit with an ATmega8



(a) Schematic



(b) Components



(c) Completed test circuit.

Figure 15: A test circuit with an ATmega8

```
#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>

#if defined(__AVR_ATmega8__)

#else
#  error "Don't know what kind of MCU you are compiling for"
#endif

int main (void)
{
  volatile long int i;     /* i must be volatile to prevent the compiler from
                              optimizing thee pause loop away                 */

  DDRB=0xff;               /* Set all port B bits to output */

  while(1) {               /* An infinite loop */
    PORTB=000;             /* Set all port B bits low  */
    for(i=0;i<10000;i++);  /* An empty (pause) loop    */
    PORTB=0xff;            /* Set all port B bits high */
    for(i=0;i<2500;i++);   /* An empty (pause) loop    */
  }

    return (0);
}
```

Figure 16: Code example to blink with a LED on port B

```
MCU=atmega8
CC=avr-gcc
OBJCOPY=avr-objcopy
CFLAGS=-g -mmcu=$(MCU) -Wall -Wstrict-prototypes
#-------------------
all: demo.hex
#-------------------
demo.hex : demo.out
     $(OBJCOPY) -R .eeprom -O ihex demo.out demo.hex
demo.out : demo.o
     $(CC)  $(CFLAGS)  -o demo.out -Wl,-Map,demo.map demo.o
demo.o : demo.c
     $(CC) -I /usr/include/avr  $(CFLAGS) -Os -c demo.c
# you need to erase fist before loading the program.
# load (program) the software into the eeprom:

load: demo.hex
     uisp -dlpt=0x378 --erase  -dprog=stk200
     uisp -dlpt=0x378 --upload if=demo.hex -dprog=stk200  -v=3 --hash=32
#-------------------
clean:
     rm -f *.o *.map *.out
#-------------------
```

Figure 17: Example Makefile for use with avr-gcc and uisp (on a UNIX system)

F Litterature

References

[1] Atmel Semiconductors, 8-bit AVR RISC Microcontroller Application Note — AVR910: In-System Programming

[2] Atmel Semiconductors, 8-bit AVR RISC Microcontroller Application Note — AVR911: AVR Open-source Programmer

[3] Lanconelli Open Systems, PonyProg - Serial device programmer http://www.lancos.com/prog.html