

Principle of operation and environment  
of the AVR Boot-loader  
optiboot

Karl-Heinz Kübbeler  
`kh_kuebbeler@web.de`

June 14, 2016

# Contents

<b>1</b>	<b>Principle function of a boot-loader</b>	<b>3</b>
<b>2</b>	<b>The Hardware of the AVR 8-bit micro controllers</b>	<b>5</b>
2.1	CPU and memory access . . . . .	5
2.2	Input and Output function . . . . .	7
2.3	The start of AVR micro-controllers . . . . .	8
2.4	Writing to the AVR memories . . . . .	10
2.4.1	Parallel programming . . . . .	11
2.4.2	serial download with ISP . . . . .	11
2.4.3	Self programming with serial interface . . . . .	12
2.4.4	Diagnostic Tools . . . . .	13
<b>3</b>	<b>The optiboot boot-loader for AVR Micro-controllers</b>	<b>14</b>
3.1	Changes and enhancements to the version 6.2 . . . . .	14
3.2	Automatic size adaption in the optiboot Makefile . . . . .	14
3.3	target selection for the optiboot Makefile . . . . .	15
3.4	The Options for the optiboot Makefile . . . . .	17
3.5	Usage of optiboot without a boot-loader area . . . . .	18

# Preface

My interest for the AVR boot-loaders begun, as some users had told me their interest to run the transistor tester software at some boards of the Arduino family. Of course the transistor tester software does not run as Arduino Sketch. The Arduino development environment Sketch is only used to show the output of the serial interface. The transistor tester software do not use the Arduino library. This is not necessary to use the boot-loader.

The boot-loader is a little program, which can receive program data from a serial interface from a host (PC) and can put this data in the instruction storage (flash) of the micro-controller. Because the transistor tester software use very much program storage, the boot-loader should use as less program memory as possible for his program. The boot-loader should also be able to write data to the nonvolatile data storage of the AVR, the EEprom. So the target was specified. I search a boot-loader, which support the writing of flash and EEprom, but use only little space in the flash memory for it's own code.

# Chapter 1

## Principle function of a boot-loader

A boot-loader is a little program, which can receive program data from a interface and store this data in the instruction memory of a processor. Typically the received program is also started at the end of transmission. With this method a computer with writable instruction memory is able to run any application program.

In principle the BIOS of a PC is also a boot-loader, but the BIOS is extended for the function to select a interface to fetch the program data. You can select a chain of peripheral equipment, which is tested for existence of program data. Often is a second stage of loader started, which can choose more selections like different operating systems or boot options.

For the micro-controllers the function of a boot-loader is designed more simple. Only one interface is preselected and there are no further options selectable during operation. A characteristic for the mode of operation of a boot-loader is the type of instruction memory. If the instruction memory of the computer is build with RAM (Random Access Memory), the boot-loader must be sure to start any application program only, if it is loaded just before.

For a micro-controller with non-volatile instruction memory (flash), the boot-loader can assume, that a application program has been loaded some time ago to the instruction memory. Therefore the boot-loader try to start a application program every time after waiting for new program data a appropriate time. It doesn't matter, if new program data are received before or not. Even if there was never loaded any application program, the result is not fatal. The facility to load a application program later is still available. The lack of any application program let the boot-loader resume with the next try to get program data from the serial interface.

The figure 1.1 shows the principle function of boot-loaders, which receive their data from a serial interface.

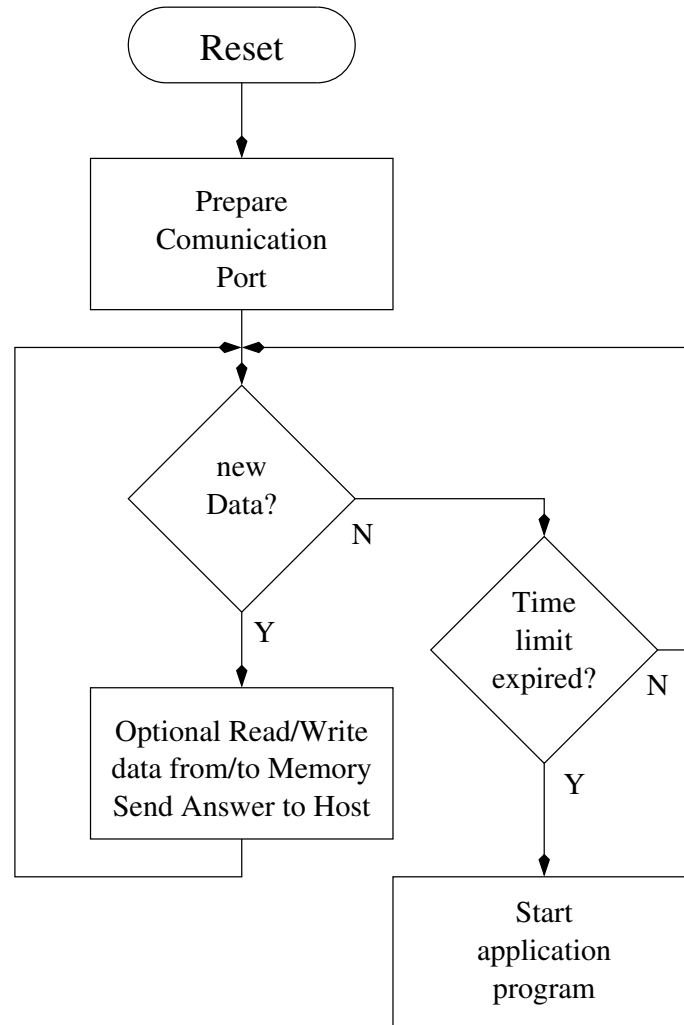


Figure 1.1. Principle function of a boot-loader

The transmitter process at the PC will reset the AVR target processor at the beginning of transmission. If the reset is not done automatically, you must reset the AVR processor manually. The PC tries to start the communication with the AVR processor by sending a data byte to the serial output and wait for any serial response of the AVR processor. If the answer is not received in a appropriate time, the procedure is repeated some times. The boot-loader program at the AVR processor wait only a limited time for new data. If the wait time is exceeded, the boot-loader ties to start a application program in the flash memory.

# Chapter 2

## The Hardware of the AVR 8-bit micro controllers

### 2.1 CPU and memory access

You can find any thing at the chip of a AVR 8-bit micro-controller, what is needed to run a digital computer. You find a clock generator, registers, data storage (RAM), program storage (flash), input registers and output registers. The content of registers and data storage is loosed with every restart. The content of the instruction memory (flash) and the often available additional nonvolatile data storage (EEProm) is preserved for long time. The figure 2.1 shows a simplified block diagram of a 8-bit AVR micro-controller.

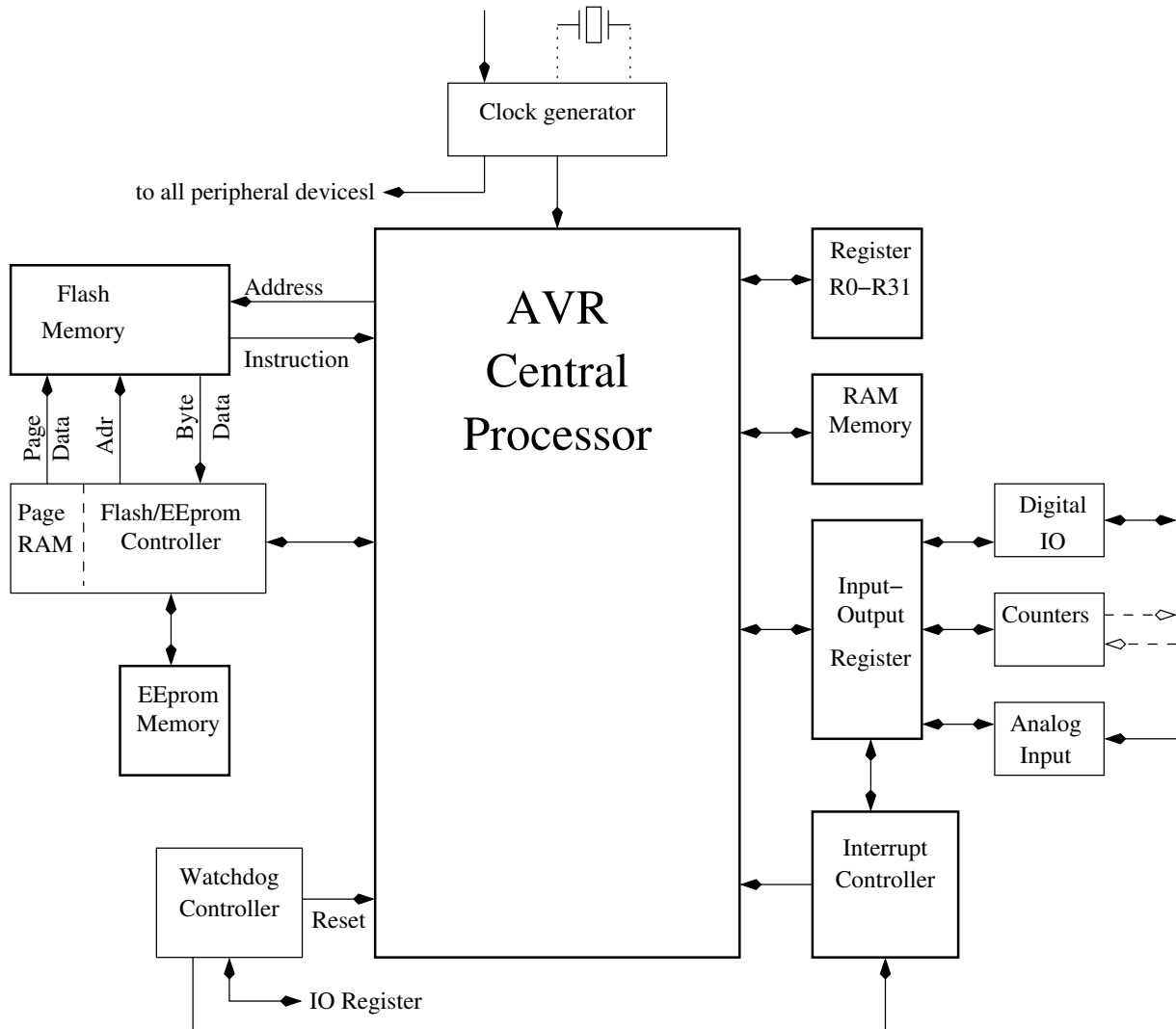


Figure 2.1. Simplified block diagram of a AVR mikrocontroller

You can see at the diagram, that the CPU (Central Processor Unit) can access easy the registers R0-R31 and the RAM memory. Also the access to the input and output registers is easy possible. But the access to the instruction memory (flash) is only possible with a special controller and more complex.

Only the instruction engine can easy access the flash data for the selected program address. With the Load Immediate (LDI) instruction you can transfer parts of the instruction word to the upper registers (R16-R31). Also with the instructions ADIW, ANDI, CPI, ORI, SBCI, SBIW and SUBI parts of the 16-bit instruction word are processed.

Usually after every instruction the program counter will be increased by one word or two words, depending on the instruction length. An exception to this rule for the normal operation is only caused by the conditional or unconditional jump instructions (RJMP, JMP, IJMP, RCALL, CALL, ICALL, RET, RETI).

Also a reset event or interrupt event can be the reason for a discontinuity of the program counter increase. A Reset will reset the whole processor and the program counter will be set to a previous selected address. An interrupt will set the program counter to an associated address. Normally the start address for the reset event is set to 0. For starting the boot-loader many AVR processors have special configuration bits in fuses to select a different start address.

A random address to the instructing memory content is only possible with the flash-controller. For that access you must tell the controller the requested byte address. Afterward the requested byte can be read with a special instruction.

More complex is a write operation to the flash memory. The write access is only possible for a total page of flash. The flash page must be erased before any write access and you must load the total page data to the controller buffer storage before you can select the write operation. You can compare this method with printing a page with a stamp. The stamp can be configured with replaceable letters before the next print. So you can print any text. But you need a empty sheet of paper and you must configure the whole text for the page. Only if all is prepared right, you can stamp a page.

Also the access to the nonvolatile data memory (EEPROM) is only possible with the special controller. The writing to the EEPROM is more simple compared to the flash memory access, but you need the controller access too. You can not use the EEPROM and flash memory access together, because some common parts of the controller is used.

## 2.2 Input and Output function

The CPU can access the external pins with IO-registers. The IO-register are organized with Byte access, so that you can select up to 8 pins with one IO register. The figure 2.2 shows the structure of a port pin circuit.

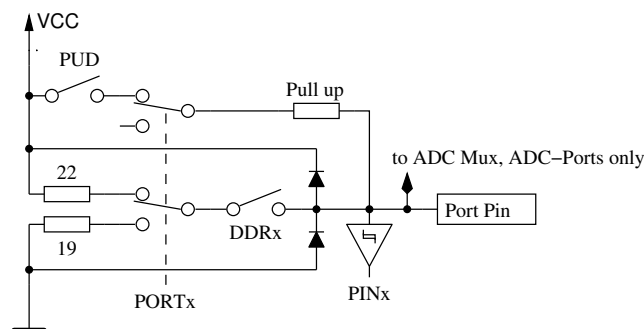


Figure 2.2. Simplified circuit of every AVR Port pin

Every pin is associated with one port bit and can be used as output pin (PORT) or as input pin (PIN). The 8 bits of the Data Direction Register (DDR) are used to select the mode of every port bit. For every pin you will find a associated bit number in three different registers. The DDR register is used to select the direction of the associated bits. The PIN register shows the voltage level of the associated bits. If the voltage is below the half operating voltage, the PIN bit is switched to 0 and above the half operating voltage the PIN bit is switched to 1. If the associated bit in the DDR register is set to 1, the associated bit in the PORT register select the level of the output signal. A 0 select a voltage level near to GND and a 1 select a voltage level near to the operating voltage (VCC). If the output mode for one bit is deselected with a 0 in the DDR register, a setting of the corresponding bit in the PORT register enables a Pull-Up resistor. But if the PUD bit in the MPU configuration register MCUPR is set, all Pull-Up registers are deactivated. All associated registers and bits of a group (8-bit) use always the same code letter. For the second group this letter is a B. The output port of this group has the name PORTB, the input port can be accessed with the name PINB and the register for the data direction has the name DDRB. For the identification of a single bit a number between 0 and 7 will be appended. For example the bit 0 of the input port B would be named PINB0. This terminology is used in the Atmel documentation and is used also with program languages.



## 2.3 The start of AVR micro-controllers

With the factory set configuration of the AVR micro-controller a first pass over of the minimal operating voltage will cause a reset of the processor. All IO-register are set to predefined values and after waiting some time to stabilize the operating voltage the instruction unit is started with the flash address 0. Normally all pins are set to input mode. Beside this reason for a reset event there exist three other reasons for a reset of the processor. The reason for the reset event is saved in the MCU status register (MCUSR) with four bits.

Name of Flag	Reason for the Reset
PORF	Power-on Reset This Reset is caused by switching on the operating voltage. This reason can not be deactivated.
BORF	Brown-out Reset This reason can only occur, if the function is selected with the BODLEVEL bits of a Fuse and no Brown-out Interrupt is selected.
EXTRF	External Reset is caused by a 0 level at the Reset Pin, if the fuse RSTDISBL is not activated.
WDRF	Watchdog Reset can only be set, if the corresponding Interrupt is not enabled.

Table 2.1. Different Reset reasons in the MCUSR register

By setting the right configuration bits in the fuses of the AVR micro-controller you can select another start address as the usual 0. The figure 2.3 shows the options for a ATmega168. This processor has a total instruction memory (flash) capacity of 16384 Byte. The instruction interpreter of the micro-controller can access a 16-bit parallel instruction code of the flash memory. So the largest program counter is only 8190 for optiboot! It can not be 8192 because the counting begin with 0, but it is one word less because the last word of the flash memory is used to hold the version number of optiboot.

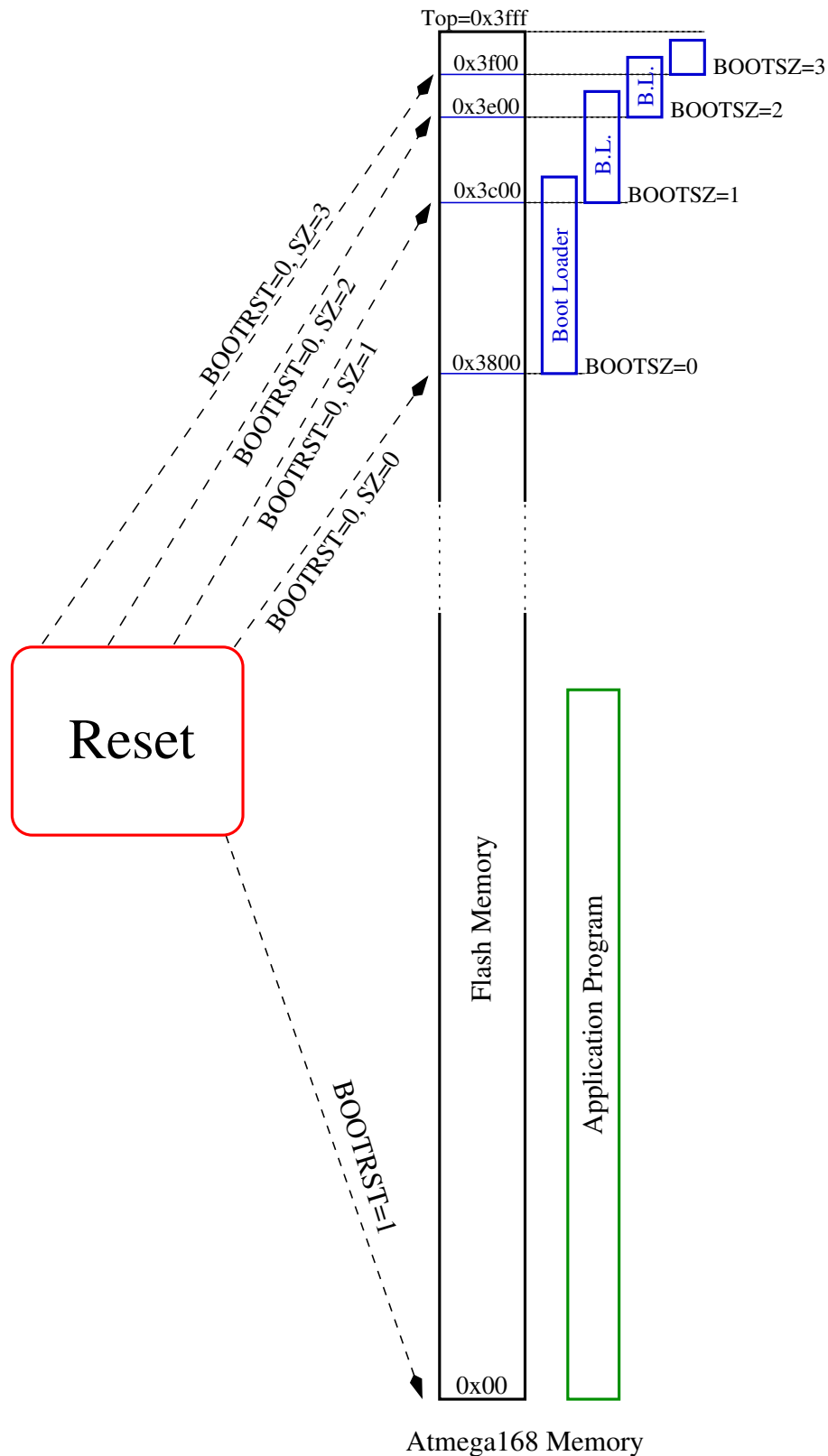


Figure 2.3. The different Start-Options for the ATmega168

The ATmega168 can select a boot-loader size of 256 bytes (BOOTSZ=3), 512 Bytes (BOOTSZ=2), 1024 Bytes (BOOTSZ=1) and 2048 Bytes (BOOTSZ=0). The application program would like to use as many program space as possible, so the boot-loader space should be as low as possible. The boot-loader code is placed at the highest starting address possible. The activating of Lock-bits of the AVR micro-controller can protect the boot-loader area against overwrite. Once activated Lock-bits can only be reset with a total erase of the AVR memories.

For this processor, also for the Mega48 and Mega88, the control bits for the boot-loader start are located in the extended fuse (efuse). This is also true for the BOOTRST fuse, which can be used to switch the start address from 0 to the boot-loader start. For most other AVR micro-controller, also for the ATmega328, the same control bits are located in the high fuse (hfuse). The table 2.2 shows the memory sizes of different AVR-micro-controllers and additionally the options for the Boot-loader area. The boot-loader options are located in the same bit numbers, whatever fuse is selected, the high or extended fuse.

Processor Type	Flash size	EEProm size	RAM size	UART	Boot Config Fuse	BOOTSZ			
						=3	=2	=1	=0
ATmega48	4K	256	512	1	Ext.	256	512	1K	2K
ATtiny84	8K	512	512	-	-	(N x 64)			
ATmega8	8K	512	1K	1	High	256	512	1K	2K
ATmega88	8K	512	1K	1	Ext.	256	512	1K	2K
ATmega16	16K	512	1K	1	High	256	512	1K	2K
ATmega168	16K	512	1K	1	Ext.	256	512	1K	2K
ATmega164	16K	512	1K	1	High	256	512	1K	2K
ATmega32	32K	1K	2K	1	High	512	1K	2K	4K
ATmega328	32K	1K	2K	1	High	512	1K	2K	4K
ATmega324	32K	1K	2K	2	High	512	1K	2K	4K
ATmega644	64K	2K	4K	2	High	1K	2K	4K	8K
ATmega640	64K	4K	8K	4	High	1K	2K	4K	8K
ATmega1284	128K	4K	16K	2	High	1K	2K	4K	8K
ATmega1280	128K	4K	8K	4	High	1K	2K	4K	8K
ATmega2560	262K	4K	8K	4	High	1K	2K	4K	8K

Table 2.2. Boot-loader configurations for different micro-controllers

By the way the boot-loader will work for the first time, even if the BOOTRST fuse bit is not activated. In this case the reset vector is still set to address 0, where usually the application program is located. Because for the first time no application program is loaded, the CPU execute the instructions in the cleared flash memory until it reach the boot-loader code. For the ATmega168 this are less than 8000 instructions, which executes the CPU in less than 1 ms with a 8 MHz clock. But if any application program was loaded, the Reset will start the application program, if the BOOTRST bit is not activated (still set). The boot-loader program can no longer work, because it is not addressed by the reset.

## 2.4 Writing to the AVR memories

The AVR micro-controllers know three different nonvolatile memories. The most important is the instruction memory, the so called Flash memory.

In addition there are some configuration bits, which can be used to select some features of the processor. This configuration bits are organized in some bytes, the lfuse (low fuse), the hfuse (high fuse), the efuse (extended fuse), the lock byte and the calibration byte. The calibration byte is used to calibrate the frequency of the internal RC oscillator. The lock byte can be used to restrict the access to the memories. Once activated lock bits can not be reset by rewriting the lock byte. The only way to deactivate the lock bits is a complete clear of all memories. Note, that the lock function will

be activated by clearing the appropriate bits (write to 0). With the complete clear of all Memories the lock bits will be set to 1 (full access). The layout of the configuration bits to the different fuse bytes differ for the several AVR processor models and should be read in the specific data sheet. You can select the way of clock generation, the delay of start and a monitoring of operating voltage with the fuses.

Most AVR micro-controllers are also equipped with a nonvolatile data memory, the EEPROM. This memory type has no special function for the processor. It is just a way for a application program so save data for the next program start.

### 2.4.1 Parallel programming

All three nonvolatile memory types can be written or read with different technique. Usually the parallel programming method is rare used for writing the nonvolatile memories. Sometimes this method is the only way to reactivate processors, which can not accessed with other methods. For example you can not use the serial programming, if the fuse bit for the Reset pin usage is deactivated (RSTDISBL=0). The voltage at the reset pit is raised to higher voltage level (12V) for this parallel programming method. Therefore this method is also called HV-programming.

### 2.4.2 serial download with ISP

The normal way to program any non volatile memory is the serial programming. The Atmel documentation call this method also serial download. For that way a SPI (Serial Parallel Interface) interface is used. The SPI interface is build with three signals, MOSI, MISO and SCK. Additionally the Reset pin of the AVR processor must hold to 0V to force this special download mode. Together with two additional power signals GND and VCC (about 2.7 to 5V) this four signals build a ISP (In System Programming) interface, which is often integrated at many boards. The figure 2.4 shows the layout of two usual plugs, which are often integrated at boards with AVR micro-controllers.

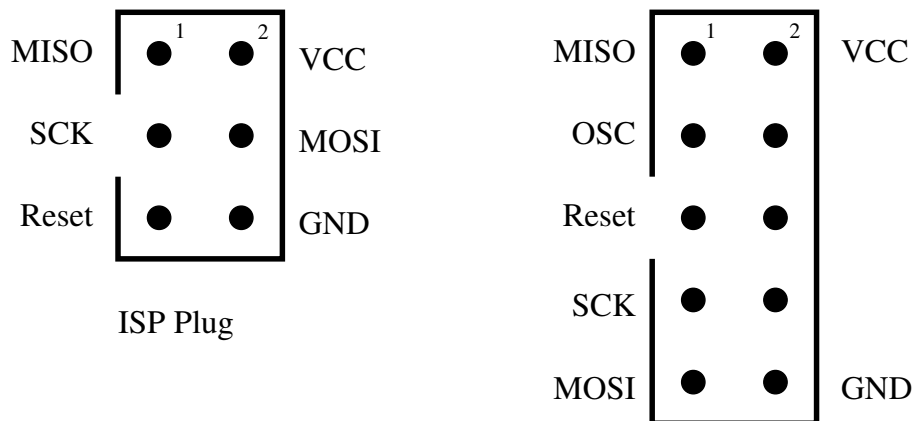


Figure 2.4. Two different types of ISP Plugs

The 10-pin version of the ISP-plug can additionally support a clock signal OSC for feed-in a clock signal to the AVR micro-controller. One of this two plug versions are usually required to program a boot-loader in the flash memory of a AVR micro-controller. The program data for the flash memory are usually created with a PC. To transfer the program data to the AVR micro-controller a ISP programmer is required, which use often a USB interface to the host computer side. But the host computer can also use a serial or parallel interface to connect a ISP programmer. The USB interface has the advantage, that the power (5V or 3.3V) for the micro-controller can be taken easy from the interface. You can choose some types of specific ISP programmers at the electronic market,

the manufacturer of the controllers offer the Atmel AVR-ISP MK2 programmer for example. But you can also use a Arduino UNO or a similar Arduino with a special program for the connected ISP interface. I use a DIAMAX ALL-AVR, which is equipped with both plug types and has some additional features.

### 2.4.3 Self programming with serial interface

Because the AVR processor can write flash and EEPROM memories with special instructions, you can write a little program to the flash memory with one of the two programming methods, which receive data from a serial interface and can write this data to the flash or EEPROM memory. Exactly this is the feature of the boot-loader optiboot. Setting of fuses or lock bytes is often not possible with this method and is not supported by the boot-loader. You must set the fuses and the lock byte with one of the other methods. The STK500 Communication Protocol from Atmel is used for the serial data transfer. Because up-to-date computers often have no more any serial interfaces, a USB - serial converter like the FTDI chip of Future Technology Devices International Ltd is used. A module with this chip is for example a UM232R.

The Chips PL2303 from Prolific Technology Inc. and CP2102 from Silicon Laboratories Inc. satisfy the same purpose. Also a suitable programmed ATmega16U2 can be used for the same function. All of these chips have a selectable Baud-rate and a TTL level for the serial signals. You have to add level converter to get real RS232 signals. But the AVR micro-controllers don't need RS232 signal level. One of these chips is integrated at any Arduino board with USB interface. For a fast answer the serial interface should also connect the DTR signal of the converter with a serial 100nF capacitor to the Reset input pin of the AVR processor. The figure 2.5 shows a typical way of connection.

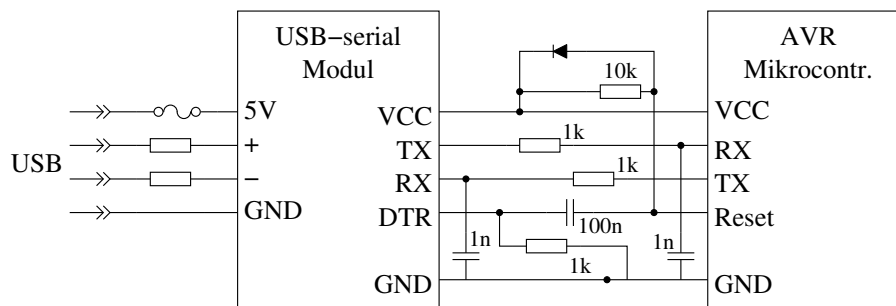


Figure 2.5. Connection of a USB-serial converter to the micro-controller

You should select the right power voltage for the USB - serial converter. Most modules can select a 3.3V or a 5V signal level with a jumper. If you have a Arduino UNO with a ATmega328p at a socket, you can remove the ATmega328p and use the board as USB - serial converter. So you can transfer program data to another AVR processor with a already installed boot-loader. If you frequently use this serial interface, a separate USB - serial interface make sense.

You can select with the Arduino development tool Sketch with the menu entry "Tools - serial Port" a detected serial port. Then you can open a monitor window with the menu entry "Tools - Serial Monitor", which can show you the serial output of your AVR micro-controller at the screen. The Baud rate of the serial interface can be selected at the monitor window. Both 1 nF capacitors at the RX-inputs removes spikes from the serial signals. The test of the software UART program was only successful with a little capacitor at the RX input of the AVR processor. The probe of a scope was sufficient as "filter" for the spikes. The hardware UART tolerates the spikes without any filter and runs proper with or without the capacitor.

The running Serial Monitor of the Arduino Sketch can disturb the program download with the serial interface, if the same USB - serial module is used for download and the monitor program. But

you can insert a additional USB - serial module to the host computer and connect only the RX-Signal to the TX-Port of the AVR micro-controller. This second serial input listen to the output of the AVR without any problems for the serial communication of the program download, if you select this second interface with the Serial Monitor tool.

## 2.4.4 Diagnostic Tools

At the Linux operating system you can install a tool with the strange name jpnevulator, which can monitor two serial inputs at the same time. Any received data are shown in a hexadecimal format and with the option -a also as ASCII characters. With the option -timing-print the system time of the serial data packets are shown. To prevent any affect to the data communication, you should select two separate USB - serial modules for this monitoring. You should connect the serial input (RX) of one module to the TX signal and the serial input of the other module to the RX signal of the AVR micro-controller. Together with the module for the program download there are three USB - serial modules connected to the PC. Of course all three modules must be set to the same baud rate (stty ... -F /dev/ttyUSB1). The full command line with the start of the protocol can be look like:

```
jpnevulator -a --timing-print --read --tty "/dev/ttyUSB1" --tty "/dev/ttyUSB2"
2016-05-29 11:05:06.589614: /dev/ttyUSB0
30 20                                0
2016-05-29 11:05:06.589722: /dev/ttyUSB1
14 10                                ..
2016-05-29 11:05:06.593593: /dev/ttyUSB0
41 81 20                            A.
2016-05-29 11:05:06.594581: /dev/ttyUSB1
14 74 10                            .t.
2016-05-29 11:05:06.597583: /dev/ttyUSB0
41 82 20                            A.
2016-05-29 11:05:06.598574: /dev/ttyUSB1
14 02 10                            ...
2016-05-29 11:05:06.601586: /dev/ttyUSB0
42 86 00 00 01 01 01 01 03 FF FF FF FF 00 80 04 B.....
00 00 00 80 00 20                    .....
2016-05-29 11:05:06.603608: /dev/ttyUSB1
14 10                                ..
2016-05-29 11:05:06.605639: /dev/ttyUSB0
45 05 04 D7 C2 00 20                E.....
2016-05-29 11:05:06.606576: /dev/ttyUSB1
14 10                                ..
...
```

# Chapter 3

## The optiboot boot-loader for AVR Micro-controllers

The optiboot Boot-loader has been created with C language by Peter Knight and Bill Westfield. I have used the version 6.2 as base for the here described revised Assembler version. I would like to underline, that I did not reinvent the optiboot boot-loader. I have just done some optimizing. Many adaptations to several target processors and special board level systems are present with the version 6.2. The program use parts of the STK500 communication protocol, which is released with AVR061 [7] from Atmel.

### 3.1 Changes and enhancements to the version 6.2

Basically I have translated the total program in the assembler language and have adapted the Makefile, that the program length will be processed automatically to select the start address of the boot-loader and set the right fuses for the program length. The selected solution generates some small files during some interim steps, which are required to solve the following steps to select the right start address and the right fuses. The start address of the boot-loader for any target processor depends on the present flash size, the flash requirement of the boot-loader code and the tile size, which is supported by the target processor for bootlace. The tile size is the smallest boot-loader size, which can be supported by the selected target processor.

For processors like the ATtiny84, which don't support the boot-loader start function, the page size of the flash memory is used for this calculation. For the ATtiny84 this are 64 Bytes. Therefore the start address of the boot-loader is always located at the begin of a flash page.

For all other supported target processors the boot-loader area can be selected with the fuse bits BOOTSZ1 and BOOTSZ0 (each with the values 0 and 1). If you put together the both bits, you get a coded boot-loader size with values between 0 and 3. Always the value of 3 select the smallest possible boot-loader area. A value of 2 select a double size, the value 1 the quadruple size and the value 0 select a size of eight times the smallest size. The table 2.2 at page 10 shows a overview for the several target processors.

### 3.2 Automatic size adaption in the optiboot Makefile

The boot-loader start address and the required boot-loader size will be adapted automatically with the Makefile. For the calculation some interim files are created, which is only possible together with some Linux tools:

**bc** a simple calculator, which can operate with input and output- values in decimal and hexadecimal values.

**cat** put the file content to the standard output.

**cut** can select part of lines of a text.

**echo** shows the specified text at standard output.

**grep** shows only lines of a text file which contain the specified string.

**tr** can replace or erase characters.

Until now this function of the Makefile is only tested with a Linux system. Probably a use with the Windows system is only possible, if you install the Cygwin package.

You don't handle the different interim files in normal case. Here I would like to refer the names and the meaning:

**BootPages.dat** hold the count of required pages by the boot-loader. For processors which support the boot-loader start feature, the value can be only 1, 2, 4, or 8 and specifies how many times the minimum size of the boot-loader area (tile) is used. With the virtual boot-loader support the number can be any value and specifies the count of required flash pages.

**BOOTSZ.dat** hold a number between 0 and 3 for selection of the BOOTSZ0 and BOOTSZ1 bits.

**BL\_StartAdr.dat** hold the start address of the boot-loaders with hexadecimal format. The start address is computed with the flash size of the selected target processor and the count of required page or tile size.

**EFUSE.dat** hold the value for the efuse in hexadecimal format. The Makefile determine depending of the target processor type, if this file is used or not.

**HFUSE.dat** hold the value for the hfuse in hexadecimal format. The Makefile determine depending of the target processor type, if this file is used or not.

### 3.3 target selection for the optiboot Makefile

The control of steps for generating the program data from the source code is defined in the Makefile. Except the main Makefile there are three additional extension Makefiles, which are included by the main Makefile: Makefile.1284, Makefile.atmel, and Makefile.extras . There can exist different configurations for the same processor type. The table 3.1 shows different basic configuration for several target processors. In principle this list can be extended. You can select some parameters also with the make call or by setting a environment variable of the shell.



Name	MCU	AVR_ FREQ	total Flash size	Flash page size	BP_ LEN	LFUSE	HFUSE	EFUSE
attiny84	t84	16M?	8K	64	(64)	62	DF	FE
atmega8	m8	16M	8K	64	256	BF	CC	-
atmega88	m88	16M	8K	64	256	FF	DD	04
atmega16	m16	16M	16K	128	256	FF	9C	-
atmega168	m168	16M	16K	128	256	FC	DD	04
atmega168p	m168p	16M	16K	128	256	FC	DD	04
atmega32	m32	16M	16K	128	256	BF	CE	-
atmega328	m328	16M	32K	128	512	FF	DE	05
atmega328p	m328p	16M	32K	128	512	FF	DE	05
atmega644p	m644p	16M	64K	256	512	F7	DE	05
atmega1284	m1284	16M	128K	256	512	F7	DE	05
atmega1284p	m1284p	16M	128K	256	512	F7	DE	05
atmega1280	m1280	16M	128K	256	1K	FF	DE	05

Table 3.1. Processor targets for optiboot Makefile

All size values are shown in byte units, the values for fuses are shown with hexadecimal values. The frequency values must be specified in Hz units, 16M is the same as 16000000 Hz. The standard baud rate of the serial interface is always 115200.

Additional to the universal processor configurations you can also select configurations for special boards or operational environment. The table 3.2 shows the different adjustments.

Name	MCU	AVR_ FREQ	BP_ LEN	L FUSE	H FUSE	E FUSE	BAUD_ RATE	LED	SOFT_ UART
luminet	t84	1M	64v	F7	DD	04	9600	0x	-
virboot8	m8	16M	64v						
diecimila	m168	(16M)		F7	DD	04		3x	-
lilypad	m168	8M		E2	DD	04	-	3x	-
pro8	m168	16M		F7	C6	04	-	3x	-
pro16	m168	16M		F7	DD	04	-	3x	-
pro20	m168	16M		F7	DC	04	-	3x	-
atmega168p_lp	m168	16M		FF	DD	04	-		-
xplained168pb	m168	(16M)					57600		
virboot328	m328p	16M	128v						-
atmega328_pro8	m328p	8M		FF	DE	05	-	3x	-
xplained328pb	m168	(16M)					57600		
xplained328p	m168	(16M)					57600		
wildfire	m1284p	16M					-	3xB5	
mega1280	m1280	16M		FF	DE	05	-		-

Table 3.2. configured targets for the optiboot Makefile

### 3.4 The Options for the optiboot Makefile

With the options you can select the feature of the optiboot boot-loader. For example you can select with the option `SOFT_UART`, that a software solution is used for the serial communication. Without this option a integrated hardware UART is used for serial communication. The pin TX (Transmit) is used for serial output and the pin RX (Receive) is used for serial input. If more than one UART is present at the target processor, the first interface with the number 0 is used. But you can also select every other present UART by specify the number with the option `UART` (`UART=1` for the second present UART). For the hardware UART interfaces the pins for transmit and receive are fixed to the specific pins. For the serial communication with software you can select any pins, which are able to do digital input and output. More details for the available options you can find in the tables 3.3 and 3.4

Name of the Option	Example	Function
F_CPU	F_CPU=8000000	Tell the program the clock frequency of the processor. The value is specified in Hz units (cycles per second). The example specifies a frequency of 8 MHz.
BAUD_RATE	BAUD_RATE=9600	Specifies the baud-rate for the serial communication. Always 8 data bits without parity is used.
SOFT_UART	SOFT_UART=1	Select a software solution for the serial communication.
UART_RX	UART_RX=D0	Specifies the port and bit number used for the serial input. The example select bit 0 of PIND as serial input. You can use this option only with the software UART.
UART_TX	UART_TX=D1	Specifies the port and bit number used for the serial output. The example select bit 1 of PORTD as serial output. You can use this option only with the software UART.
UART	UART=1	Select a hardware UART used for the serial communication. You can only select a UART if more than one is present. This option can not be used with the <code>SOFT_UART</code> Option.
LED_START_FLASHES	LED_START_FLASHES=3	Select a repetition count of flashing cycles for the control LED.
LED	LED=B3	Select a port and bit number for the control LED. The example would select the bit number 3 of the port B for the LED connection. With the option <code>LED_START_FLASHES</code> this LED will flash the specified count before the communication start. With the option <code>LED_DATA_FLASH</code> the LED will glow during wait for serial input.
LED_DATA_FLASH	LED_DATA_FLASH=1	The control LED will glow during waiting for serial input data.

Table 3.3. Important options for the optiboot Makefile

More options are listed in table 3.4 . Some of these options are only interesting for software checks and for processors without the boot-loader support.

Name of the Option	Example	Function
SUPPORT_EEPROM	SUPPORT_EEPROM=1	Select the EEPROM read and write function for the boot-loader. If the assembly language is selected as source, the EEPROM support is enabled without this option, but can be switched off by setting the SUPPORT_EEPROM Option to 0. For the C-source the function must be switched on (default = off).
C_SOURCE	C_SOURCE=1	Select the C language as source instead of the assembly language (option 0 = assembly). The assembly version requires less program space.
BIGBOOT	BIGBOOT=512	Select additional space usage for the compiled program. This is used only for tests of the automatic adaption to the program size.
VIRTUAL_BOOT_PARTITION	VIRTUAL_BOOT_PARTITION	Changes the interrupt vector table of a user program, that the boot-loader is called with a Reset. For the start of the user program another interrupt vector is used.
save_vect_num	save_vect_num=4	Choose a interrupt vector number for the VIRTUAL_BOOT_PARTITION method.

Table 3.4. More options for the optiboot Makefile

### 3.5 Usage of optiboot without a boot-loader area

For processors without a special boot-loader area in the flash memory, for example the ATtiny84, a solution is selectable to use the optiboot anyway. This function can be selected with the VIRTUAL\_BOOT\_PARTITION option. To start the boot-loader first with every Reset of the processor, the interrupt vector table of the application program is changed. At the reset vector location a jump to the optiboot program is registered. The original start address of the application program will be moved to another interrupt vector the "replacement reset vector". This interrupt vector should not be used by the application program. If the boot-loader does not receive any data from the serial interface within a appropriate time, the boot-loader jump to the location of the replacement reset vector and start the application program. The figure 3.1 should illustrate these changes.

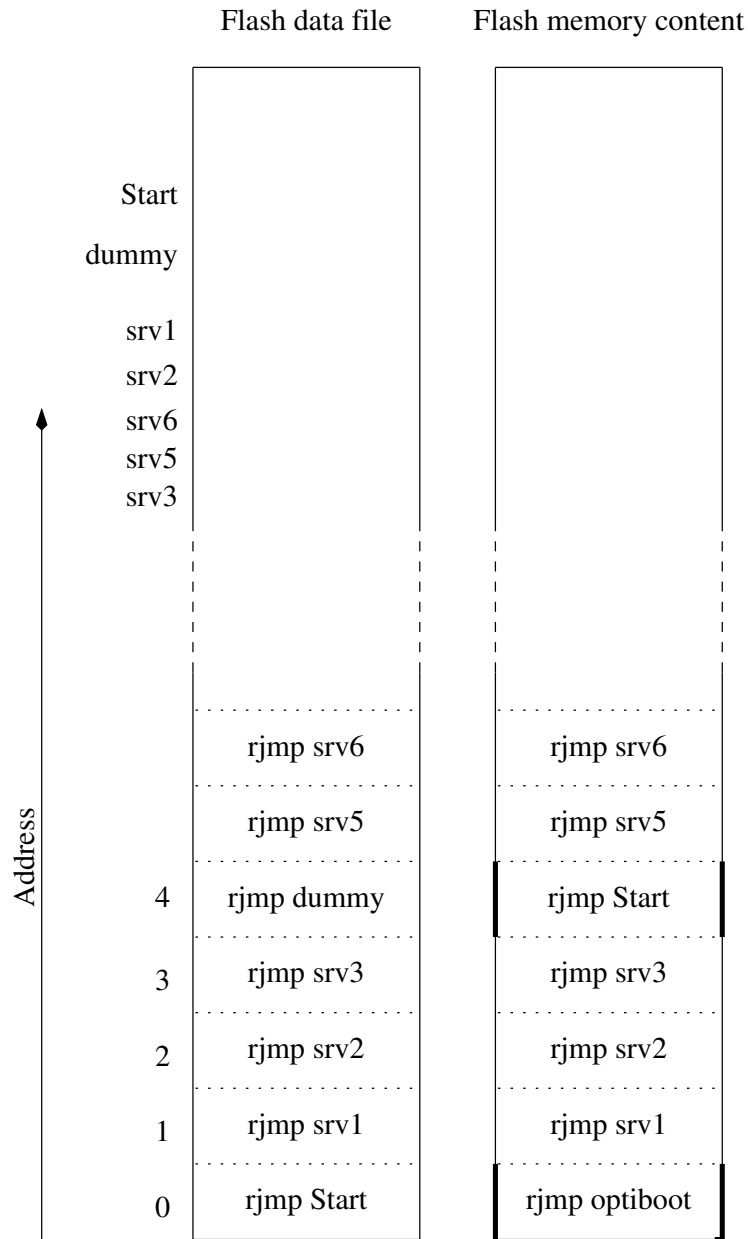


Figure 3.1. Changes of program data by optiboot

At the left side the content of the program data file (.hex) is shown. Just to the right the content of the flash memory is shown, as it is modified by the optiboot boot-loader. At two interrupt vector addresses the content is changed. At the reset vector address 0 the jump is modified to select the optiboot start address as jump target. At the "replacement vector address" 4 the original jump target address of the application program's reset vector is used as new jump target address of this vector. One of the problems with this modification is, that usually the program data is verified by the host after write is finished. To provide any error message by verify the program data, the optiboot return the program data without its own modification, not the real content of the interrupt vector table. The jump target address of the reset vector can be reconstructed with the content of the replacement vector address. But the original content of the replacement vector would be lost because there is no place to save the original content in the flash memory. Therefore optiboot use the last two places of the EEprom memory to save this original content of the replacement vector. So the verify of the program data is possible without errors, as long as the application program do not use one of the last two EEprom locations. Even if the application program use one of the last two EEprom locations, the boot-loader will be unaffected. Only the program verify by the host is no longer possible without

a error message. An error message will occur at the location of the replacement interrupt vector.

For processors with more than 8 kByte flash memory two instruction words are used for every interrupt vector. Normally every of this double words hold one JMP instruction with the proper jump target address. The optiboot program can respect these JMP vector table too. But if you use the linker `avr-ld` with the option `-relax`, all JMP instructions are replaced by a RJMP, if this is possible for the target address. This replacement of JMP instruction in the vector table by RJMP is not respected by the optiboot program. The optiboot program assume, that all interrupt vector numbers of a processor with more than 8 kByte flash hold a JMP instruction. For that reason a optiboot program with the `VIRTUAL_BOOT_Partition` option will not work with a application program, which is linked with the `-relax` option. The same problem exist, if the application program itself use a RJMP instruction in one of the two critical interrupt vector positions.

Further you should notice, that you don't activate the `BOOTRST` fuse together with with the usage of the `VIRTUAL_BOOT_PARTITION` option. The reason is, that the start address of the boot-loader can be located to other addresses with the `VIRTUAL_BOOT_PARTITION` option than without this option. With the `VIRTUAL_BOOT_PARTITION` the start address can be placed to every begin of a flash page. For the normal boot-loader support of the AVR the start address can only respect the single, double, quadruple or octuple size of a minimum boot-loader size as shown in figure 2.3 at page 9.

# Bibliography

- [1] Atmel Corporation *8-bit AVR with 8KBytes In-System Programmable Flash - ATmega8(L)*,. Manual, 2486AA-AVR-02/13, 2013
- [2] Atmel Corporation *8-bit AVR with 16KBytes In-System Programmable Flash - ATmega16A*,. Manual, 8154C-AVR-07/14, 2014
- [3] Atmel Corporation *8-bit AVR with 32KBytes In-System Programmable Flash - ATmega32(L)*,. Manual, doc2503-AVR-07/11, 2011
- [4] Atmel Corporation *8-bit AVR with 4/8/16/32KBytes In-System Programmable Flash - ATmega48 - ATmega328*,. Manual, 8271J-AVR-11/15, 2015
- [5] Atmel Corporation *8-bit AVR with 16/32/64/128KBytes In-System Programmable Flash - ATmega164 - ATmega1284*,. Manual, 8272G-AVR-01/15, 2015
- [6] Atmel Corporation *8-bit AVR with 2/4/8KBytes In-System Programmable Flash - ATtiny24-ATtiny44-ATtiny84* ,. Manual, doc8006-AVR-10/10, 2010
- [7] Atmel Corporation *STK500 Communication Protocol* ,. Application Note, AVR061-04/03, 2003
- [8] [http://en.wikibooks.org/wiki/LaTeX/LaTeX\\_documentation](http://en.wikibooks.org/wiki/LaTeX/LaTeX_documentation),. Guide to the LaTeX markup language, 2012
- [9] <http://www.xfig.org/userman> *Xfig documentation*,. Documentation of the interactive drawing tool xfig, 2009
- [10] <http://www.cs.ou.edu/~fagg/classes/general/atmel/avrdude.pdf> *AVRDUDE User-guide*,. A program for download/uploading AVR microcontroller flash and eeprom, by Brian S. Dean 2006
- [11] <http://www.mikrocontroller.net/articles/AVRDUDE> *Online Dokumentation des avrdude programmer interface*, Online Article, 2004-2011