AVRDUDE

Release 0.1

2003-2005 Brian Dean, 2006-2022 Jörg Wunsch

Jan 06, 2022

CONTENTS

This file documents the avrdude program for downloading/uploading programs to Microchip AVR microcontrollers.

For avrdude version 6.99-20211218, 6 January 2022.

Send comments on AVRDUDE to avrdude-dev@nongnu.org.

Use https://github.com/avrdudes/avrdude/issues to report bugs.

CHAPTER

ONE

INTRODUCTION

AVRDUDE - AVR Downloader Uploader - is a program for downloading and uploading the on-chip memories of Atmel's AVR microcontrollers. It can program the Flash and EEPROM, and where supported by the serial programming protocol, it can program fuse and lock bits. AVRDUDE also supplies a direct instruction mode allowing one to issue any programming instruction to the AVR chip regardless of whether AVRDUDE implements that specific feature of a particular chip.

AVRDUDE can be used effectively via the command line to read or write all chip memory types (eeprom, flash, fuse bits, lock bits, signature bytes) or via an interactive (terminal) mode. Using AVRDUDE from the command line works well for programming the entire memory of the chip from the contents of a file, while interactive mode is useful for exploring memory contents, modifying individual bytes of eeprom, programming fuse/lock bits, etc.

AVRDUDE supports the following basic programmer types: Atmel's STK500, Atmel's AVRISP and AVRISP mkII devices, Atmel's STK600, Atmel's JTAG ICE (the original one, mkII, and 3, the latter two also in ISP mode), appnote avr910, appnote avr109 (including the AVR Butterfly), serial bit-bang adapters, and the PPI (parallel port interface). PPI represents a class of simple programmers where the programming lines are directly connected to the PC parallel port. Several pin configurations exist for several variations of the PPI programmers, and AVRDUDE can be configured to work with them by either specifying the appropriate programmer on the command line or by creating a new entry in its configuration file. All that's usually required for a new entry is to tell AVRDUDE which pins to use for each programming function.

A number of equally simple bit-bang programming adapters that connect to a serial port are supported as well, among them the popular Ponyprog serial adapter, and the DASA and DASA3 adapters that used to be supported by uisp(1). Note that these adapters are meant to be attached to a physical serial port. Connecting to a serial port emulated on top of USB is likely to not work at all, or to work abysmally slow.

If you happen to have a Linux system with at least 4 hardware GPIOs available (like almost all embedded Linux boards) you can do without any additional hardware - just connect them to the MOSI, MISO, RESET and SCK pins on the AVR and use the linuxgpio programmer type. It bitbangs the lines using the Linux sysfs GPIO interface. Of course, care should be taken about voltage level compatibility. Also, although not strictly required, it is strongly advisable to protect the GPIO pins from overcurrent situations in some way. The simplest would be to just put some resistors in series or better yet use a 3-state buffer driver like the 74HC244. Have a look at http://kolev.info/avrdude-linuxgpio for a more detailed tutorial about using this programmer type.

Under a Linux installation with direct access to the SPI bus and GPIO pins, such as would be found on a Raspberry Pi, the 'linuxspi' programmer type can be used to directly connect to and program a chip using the built in interfaces on the computer. The requirements to use this type are that an SPI interface is exposed along with one GPIO pin. The GPIO serves as the reset output since the Linux SPI drivers do not hold slave select down when a transfer is not occuring and thus it cannot be used as the reset pin. A readily available level translator should be used between the SPI bus/reset GPIO and the chip to avoid potentially damaging the computer's SPI controller in the event that the chip is running at 5V and the SPI runs at 3.3V. The GPIO chosen for reset can be configured in the avrdude configuration file using the *reset* entry under the linuxspi programmer, or directly in the port specification. An external pull-up resistor should be connected between the AVR's reset pin and Vcc. If Vcc is not the same as the SPI voltage, this should be done on the AVR side of the level translator to protect the hardware from damage.

On a Raspberry Pi, header J8 provides access to the SPI and GPIO lines.

Typically, pins 19, 21, and 23 are SPI MOSI, MISO, and SCK, while pins 24 and 26 would serve as CE outputs. So, close to these pins is pin 22 as GPIO25 which can be used as /RESET, and pin 25 can be used as GND.

A typical programming cable would then look like:

@multitable @columnfractions .15 .15 .3 * *J8 pin* @tab *ISP pin* @tab *Name* * *21* @tab *1* @tab *MISO* * - @tab *2* @tab *Vcc - leave open* * *23* @tab *3* @tab *SCK* * *19* @tab *4* @tab *MOSI* * *22* @tab *5* @tab */RESET* * *25* @tab *6* @tab *GND* @end multitable

(Mind the 3.3 V voltage level of the Raspberry Pi!)

The *-P `portname`* option defaults to */dev/spidev0.0:/dev/gpiochip0* for this programmer.

The STK500, JTAG ICE, avr910, and avr109/butterfly use the serial port to communicate with the PC. The STK600, JTAG ICE mkII/3, AVRISP mkII, USBasp, avrftdi (and derivatives), and USBtinyISP programmers communicate through the USB, using *libusb* as a platform abstraction layer. The avrftdi adds support for the FT2232C/D, FT2232H, and FT4232H devices. These all use the MPSSE mode, which has a specific pin mapping. Bit 1 (the lsb of the byte in the config file) is SCK. Bit 2 is MOSI, and Bit 3 is MISO. Bit 4 usually reset. The 2232C/D parts are only supported on interface A, but the H parts can be either A or B (specified by the usbdev config parameter). The STK500, STK600, JTAG ICE, and avr910 contain on-board logic to control the programming of the target device. The avr109 bootloader implements a protocol similar to avr910, but is actually implemented in the boot area of the target's flash ROM, as opposed to being an external device. The fundamental difference between the two types lies in the protocol used to control the programmer. The avr910 protocol is very simplistic and can easily be used as the basis for a simple, home made programmer since the firmware is available online. On the other hand, the STK500 protocol is more robust and complicated and the firmware is not openly available. The JTAG ICE also uses a serial communication protocol which is similar to the STK500 firmware version 2 one. However, as the JTAG ICE is intended to allow on-chip debugging as well as memory programming, the protocol is more sophisticated. (The JTAG ICE mkII protocol can also be run on top of USB.) Only the memory programming functionality of the JTAG ICE is supported by AVRDUDE. For the JTAG ICE mkII/3, JTAG, debugWire and ISP mode are supported, provided it has a firmware revision of at least 4.14 (decimal). See below for the limitations of debugWire. For ATxmega devices, the JTAG ICE mkII/3 is supported in PDI mode, provided it has a revision 1 hardware and firmware version of at least 5.37 (decimal).

The Atmel-ICE (ARM/AVR) is supported (JTAG, PDI for Xmega, debugWIRE, ISP modes).

Atmel's XplainedPro boards, using EDBG protocol (CMSIS-DAP compliant), are supported by the 'jtag3' programmer type.

Atmel's XplainedMini boards, using mEDBG protocol, are also supported by the 'jtag3' programmer type.

The AVR Dragon is supported in all modes (ISP, JTAG, PDI, HVSP, PP, debugWire). When used in JTAG and debugWire mode, the AVR Dragon behaves similar to a JTAG ICE mkII, so all device-specific comments for that device will apply as well. When used in ISP and PDI mode, the AVR Dragon behaves similar to an AVRISP mkII (or JTAG ICE mkII in ISP mode), so all device-specific comments will apply there. In particular, the Dragon starts out with a rather fast ISP clock frequency, so the *-B `bitclock`* option might be required to achieve a stable ISP communication. For ATxmega devices, the AVR Dragon is supported in PDI mode, provided it has a firmware version of at least 6.11 (decimal).

Wiring boards (e.g. Arduino Mega 2560 Rev3) are supported, utilizing STK500 V2.x protocol, but a simple DTR/RTS toggle to set the boards into programming mode. The programmer type is 'wiring'. Note that the -D option will likely be required in this case, because the bootloader will rewrite the program memory, but no true chip erase can be performed.

The Arduino (which is very similar to the STK500 1.x) is supported via its own programmer type specification 'arduino'. This programmer works for the Arduino Uno Rev3.

The BusPirate is a versatile tool that can also be used as an AVR programmer. A single BusPirate can be connected to up to 3 independent AVRs. See the section on *extended parameters* below for details.

The USBasp ISP and USBtinyISP adapters are also supported, provided AVRDUDE has been compiled with libusb support. They both feature simple firmware-only USB implementations, running on an ATmega8 (or ATmega88), or

ATtiny2313, respectively.

The Atmel DFU bootloader is supported in both, FLIP protocol version 1 (AT90USB* and ATmega*U* devices), as well as version 2 (Xmega devices). See below for some hints about FLIP version 1 protocol behaviour.

The MPLAB(R) PICkit 4 and MPLAB(R) SNAP are supported in ISP, PDI and UPDI mode. The Curiosity Nano board is supported in UPDI mode. It is dubbed ``PICkit on Board”, thus the name *pkobn_updi*.

SerialUPDI programmer implementation is based on Microchip’s *pymcuprog* (https://github.com/microchip-pic-avr-tools/pymcuprog) utility, but it also contains some performance improvements included in Spence Kohde’s *DxCore* Arduino core (https://github.com/SpenceKonde/DxCore). In a nutshell, this programmer consists of simple USB->UART adapter, diode and couple of resistors. It uses serial connection to provide UPDI interface. *SerialUPDI programmer* for more details and known issues.

The jtag2updi programmer is supported, and can program AVRs with a UPDI interface. Jtag2updi is just a firmware that can be uploaded to an AVR, which enables it to interface with avrdude using the jtagice mkii protocol via a serial link (https://github.com/ElTangas/jtag2updi).

The Micronucleus bootloader is supported for both protocol version V1 and V2. As the bootloader does not support reading from flash memory, use the *-V* option to prevent AVRDUDE from verifing the flash memory. See the section on *extended parameters* below for Micronucleus specific options.

1.1 History and Credits

AVRDUDE was written by Brian S. Dean under the name of AVRPROG to run on the FreeBSD Operating System. Brian renamed the software to be called AVRDUDE when interest grew in a Windows port of the software so that the name did not conflict with AVRPROG.EXE which is the name of Atmel’s Windows programming software.

The AVRDUDE source now resides in the public CVS repository on savannah.gnu.org (http://savannah.gnu.org/projects/avrdude/), where it continues to be enhanced and ported to other systems. In addition to FreeBSD, AVRDUDE now runs on Linux and Windows. The developers behind the porting effort primarily were Ted Roth, Eric Weddington, and Joerg Wunsch.

And in the spirit of many open source projects, this manual also draws on the work of others. The initial revision was composed of parts of the original Unix manual page written by Joerg Wunsch, the original web site documentation by Brian Dean, and from the comments describing the fields in the AVRDUDE configuration file by Brian Dean. The texi formatting was modeled after that of the Simulavr documentation by Ted Roth.

CHAPTER

TWO

COMMAND LINE OPTIONS

2.1 Option Descriptions

AVRDUDE is a command line tool, used as follows:

```
avrdude -p partno `options` ...
```

Command line options are used to control AVRDUDE's behaviour. The following options are recognized:

-p `partno` This is the only mandatory option and it tells AVRDUDE what type of part (MCU) that is connected to the programmer. The *partno* parameter is the part's id listed in the configuration file. Specify -p ? to list all parts in the configuration file. If a part is unknown to AVRDUDE, it means that there is no config file entry for that part, but it can be added to the configuration file if you have the Atmel datasheet so that you can enter the programming specifications. Currently, the following MCU types are understood:

@multitable @columnfractions .15 .3

`uc3a0512` @tab AT32UC3A0512

`c128` @tab AT90CAN128

`c32` @tab AT90CAN32

`c64` @tab AT90CAN64

`pwm2` @tab AT90PWM2

`pwm216` @tab AT90PWM216

`pwm2b` @tab AT90PWM2B

`pwm3` @tab AT90PWM3

`pwm316` @tab AT90PWM316

`pwm3b` @tab AT90PWM3B

`1200` @tab AT90S1200 (***)*

`2313` @tab AT90S2313

`2333` @tab AT90S2333

`2343` @tab AT90S2343 ()*

`4414` @tab AT90S4414

`4433` @tab AT90S4433

`4434` @tab AT90S4434

`8515` @tab AT90S8515

`8535` @tab AT90S8535

`usb1286` @tab AT90USB1286

`usb1287` @tab AT90USB1287

`usb162` @tab AT90USB162

`usb646` @tab AT90USB646

`usb647` @tab AT90USB647

`usb82` @tab AT90USB82

`m103` @tab ATmega103

`m128` @tab ATmega128

`m1280` @tab ATmega1280

`m1281` @tab ATmega1281

`m1284` @tab ATmega1284

`m1284p` @tab ATmega1284P

`m1284rfr2` @tab ATmega1284RFR2

`m128rfa1` @tab ATmega128RFA1

`m128rfr2` @tab ATmega128RFR2

`m16` @tab ATmega16

`m1608` @tab ATmega1608

`m1609` @tab ATmega1609

`m161` @tab ATmega161

`m162` @tab ATmega162

`m163` @tab ATmega163

`m164p` @tab ATmega164P

`m168` @tab ATmega168

`m168p` @tab ATmega168P

`m168pb` @tab ATmega168PB

`m169` @tab ATmega169

`m16u2` @tab ATmega16U2

`m2560` @tab ATmega2560 (*)

`m2561` @tab ATmega2561 (*)

`m2564rfr2` @tab ATmega2564RFR2

`m256rfr2` @tab ATmega256RFR2

`m32` @tab ATmega32

`m3208` @tab ATmega3208

`m3209` @tab ATmega3209

`m324a` @tab ATmega324A

`m324p` @tab ATmega324P

`m324pa` @tab ATmega324PA

`m324pb` @tab ATmega324PB

`m325` @tab ATmega325

`m3250` @tab ATmega3250

`m328` @tab ATmega328

`m328p` @tab ATmega328P

`m328pb` @tab ATmega328PB

`m329` @tab ATmega329

`m3290` @tab ATmega3290

`m3290p` @tab ATmega3290P

`m329p` @tab ATmega329P

`m32m1` @tab ATmega32M1

`m32u2` @tab ATmega32U2

`m32u4` @tab ATmega32U4

`m406` @tab ATMEGA406

`m48` @tab ATmega48

`m4808` @tab ATmega4808

`m4809` @tab ATmega4809

`m48p` @tab ATmega48P

`m48pb` @tab ATmega48PB

`m64` @tab ATmega64

`m640` @tab ATmega640

`m644` @tab ATmega644

`m644p` @tab ATmega644P

`m644rfr2` @tab ATmega644RFR2

`m645` @tab ATmega645

`m6450` @tab ATmega6450

`m649` @tab ATmega649

`m6490` @tab ATmega6490

`m64m1` @tab ATmega64M1

`m64rfr2` @tab ATmega64RFR2

`m8` @tab ATmega8

`m808` @tab ATmega808

`m809` @tab ATmega809

`m8515` @tab ATmega8515

`m8535` @tab ATmega8535

`m88` @tab ATmega88

`m88p` @tab ATmega88P

`m88pb` @tab ATmega88PB

`m8a` @tab ATmega8A

`m8u2` @tab ATmega8U2

`t10` @tab ATtiny10

*`t11` @tab ATtiny11 (**)**

`t12` @tab ATtiny12

`t13` @tab ATtiny13

`t15` @tab ATtiny15

`t1604` @tab ATtiny1604

`t1606` @tab ATtiny1606

`t1607` @tab ATtiny1607

`t1614` @tab ATtiny1614

`t1616` @tab ATtiny1616

`t1617` @tab ATtiny1617

`t1624` @tab ATtiny1624

`t1626` @tab ATtiny1626

`t1627` @tab ATtiny1627

`t1634` @tab ATtiny1634

`t20` @tab ATtiny20

`t202` @tab ATtiny202

`t204` @tab ATtiny204

`t212` @tab ATtiny212

`t214` @tab ATtiny214

`t2313` @tab ATtiny2313

`t24` @tab ATtiny24

`t25` @tab ATtiny25

`t26` @tab ATtiny26

`t261` @tab ATtiny261

`t28` @tab ATtiny28

`t3216` @tab ATtiny3216

`t3217` @tab ATtiny3217

`t4` @tab ATtiny4

`t40` @tab ATtiny40

`t402` @tab ATtiny402

`t404` @tab ATtiny404

`t406` @tab ATtiny406

`t412` @tab ATtiny412

`t414` @tab ATtiny414

`t416` @tab ATtiny416

`t417` @tab ATtiny417

`t424` @tab ATtiny424

`t426` @tab ATtiny426

`t427` @tab ATtiny427

`t4313` @tab ATtiny4313

`t43u` @tab ATtiny43u

`t44` @tab ATtiny44

`t441` @tab ATtiny441

`t45` @tab ATtiny45

`t461` @tab ATtiny461

`t5` @tab ATtiny5

`t804` @tab ATtiny804

`t806` @tab ATtiny806

`t807` @tab ATtiny807

`t814` @tab ATtiny814

`t816` @tab ATtiny816

`t817` @tab ATtiny817

`t824` @tab ATtiny824

`t826` @tab ATtiny826

`t827` @tab ATtiny827

`t84` @tab ATtiny84

`t841` @tab ATtiny841

`t85` @tab ATtiny85

`t861` @tab ATtiny861

`t88` @tab ATtiny88

`t9` @tab ATtiny9

`x128a1` @tab ATxmega128A1

`x128a1d` @tab ATxmega128A1revD

`x128a1u` @tab ATxmega128A1U

`x128a3` @tab ATxmega128A3

`x128a3u` @tab ATxmega128A3U

`x128a4` @tab ATxmega128A4

`x128a4u` @tab ATxmega128A4U

`x128b1` @tab ATxmega128B1

`x128b3` @tab ATxmega128B3

`x128c3` @tab ATxmega128C3

`x128d3` @tab ATxmega128D3

`x128d4` @tab ATxmega128D4

`x16a4` @tab ATxmega16A4

`x16a4u` @tab ATxmega16A4U

`x16c4` @tab ATxmega16C4

`x16d4` @tab ATxmega16D4

`x16e5` @tab ATxmega16E5

`x192a1` @tab ATxmega192A1

`x192a3` @tab ATxmega192A3

`x192a3u` @tab ATxmega192A3U

`x192c3` @tab ATxmega192C3

`x192d3` @tab ATxmega192D3

`x256a1` @tab ATxmega256A1

`x256a3` @tab ATxmega256A3

`x256a3b` @tab ATxmega256A3B

`x256a3bu` @tab ATxmega256A3BU

`x256a3u` @tab ATxmega256A3U

`x256c3` @tab ATxmega256C3

`x256d3` @tab ATxmega256D3

`x32a4` @tab ATxmega32A4

`x32a4u` @tab ATxmega32A4U

`x32c4` @tab ATxmega32C4

`x32d4` @tab ATxmega32D4

`x32e5` @tab ATxmega32E5

`x384c3` @tab ATxmega384C3

`x384d3` @tab ATxmega384D3

`x64a1` @tab ATxmega64A1

`x64a1u` @tab ATxmega64A1U

`x64a3` @tab ATxmega64A3

`x64a3u` @tab ATxmega64A3U

`x64a4` @tab ATxmega64A4

`x64a4u` @tab ATxmega64A4U

`x64b1` @tab ATxmega64B1

`x64b3` @tab ATxmega64B3

`x64c3` @tab ATxmega64C3

`x64d3` @tab ATxmega64D3

`x64d4` @tab ATxmega64D4

`x8e5` @tab ATxmega8E5

`avr128da28` @tab AVR128DA28

`avr128da32` @tab AVR128DA32

`avr128da48` @tab AVR128DA48

`avr128da64` @tab AVR128DA64

`avr128db28` @tab AVR128DB28

`avr128db32` @tab AVR128DB32

`avr128db48` @tab AVR128DB48

`avr128db64` @tab AVR128DB64

`avr32da28` @tab AVR32DA28

`avr32da32` @tab AVR32DA32

`avr32da48` @tab AVR32DA48

`avr32db28` @tab AVR32DB28

`avr32db32` @tab AVR32DB32

`avr32db48` @tab AVR32DB48

`avr64da28` @tab AVR64DA28

`avr64da32` @tab AVR64DA32

`avr64da48` @tab AVR64DA48

`avr64da64` @tab AVR64DA64

`avr64db28` @tab AVR64DB28

`avr64db32` @tab AVR64DB32

`avr64db48` @tab AVR64DB48

`avr64db64` @tab AVR64DB64

`ucr2` @tab deprecated,

`lgt8fx168p` @tab LGT8FX168P

`lgt8fx328p` @tab LGT8FX328P

`lgt8fx88p` @tab LGT8FX88P @end multitable

(*) The AT90S2323 and ATtiny22 use the same algorithm.

(**) Flash addressing above 128 KB is not supported by all programming hardware. Known to work are jtag2, stk500v2, and bit-bang programmers.

(***) The ATtiny11 can only be programmed in high-voltage serial mode.

(****) The ISP programming protocol of the AT90S1200 differs in subtle ways from that of other AVRs. Thus, not all programmers support this device. Known to work are all direct bitbang programmers, and all programmers talking the STK500v2 protocol.

-b `baudrate` Override the RS-232 connection baud rate specified in the respective programmer's entry of the configuration file.

-B `bitclock` Specify the bit clock period for the JTAG interface or the ISP clock (JTAG ICE only). The value is a floating-point number in microseconds. Alternatively, the value might be suffixed with "Hz", "kHz", or "MHz", in order to specify the bit clock frequency, rather than a period. The default value of the JTAG ICE results in about 1 microsecond bit clock period, suitable for target MCUs running at 4 MHz clock and above. Unlike certain parameters in the STK500, the JTAG ICE resets all its parameters to default values when the programming software signs off from the ICE, so for MCUs running at lower clock speeds, this parameter must be specified on the command-line. It can also be set in the configuration file by using the 'default_bitclock' keyword.

-c `programmer-id` Specify the programmer to be used. AVRDUDE knows about several common programmers. Use this option to specify which one to use. The *programmer-id* parameter is the programmer's id listed in the configuration file. Specify -c ? to list all programmers in the configuration file. If you have a programmer that is unknown to AVRDUDE, and the programmer is controlled via the PC parallel port, there's a good chance that it can be easily added to the configuration file without any code changes to AVRDUDE. Simply copy an existing entry and change the pin definitions to match that of the unknown programmer. Currently, the following programmer ids are understood and supported:

@multitable @columnfractions .2 .6

`2232HIO` @tab FT2232H based generic programmer

`4232h` @tab FT4232H based generic programmer

`arduino` @tab Arduino

`arduino-ft232r` @tab Arduino: FT232R connected to ISP

`atmelice` @tab Atmel-ICE (ARM/AVR) in JTAG mode

`atmelice_dw` @tab Atmel-ICE (ARM/AVR) in debugWIRE mode

`atmelice_isp` @tab Atmel-ICE (ARM/AVR) in ISP mode

`atmelice_pdi` @tab Atmel-ICE (ARM/AVR) in PDI mode

`atmelice_updi` @tab Atmel-ICE (ARM/AVR) in UPDI mode

`avr109` @tab Atmel AppNote AVR109 Boot Loader

`avr910` @tab Atmel Low Cost Serial Programmer

`avr911` @tab Atmel AppNote AVR911 AVROSP

`avrftdi` @tab FT2232D based generic programmer

`avrisp` @tab Atmel AVR ISP

`avrisp2` @tab Atmel AVR ISP mkII

`avrispmkII` @tab Atmel AVR ISP mkII

`avrispv2` @tab Atmel AVR ISP V2

`buspirate` @tab The Bus Pirate

`buspirate_bb` @tab The Bus Pirate (bitbang interface, supports TPI)

`butterfly` @tab Atmel Butterfly Development Board

`butterfly_mk` @tab Mikrokopter.de Butterfly

`bwmega` @tab BitWizard ftdi_atmega builtin programmer

`C232HM` @tab FT232H based module from FTDI and Glyn.com.au

`c2n232i` @tab serial port banging, reset=dtr sck=!rts mosi=!txd miso=!cts

`dasa` @tab serial port banging, reset=rts sck=dtr mosi=txd miso=cts

`dasa3` @tab serial port banging, reset=!dtr sck=rts mosi=txd miso=cts

`diecimila` @tab alias for arduino-ft232r

`dragon_dw` @tab Atmel AVR Dragon in debugWire mode

`dragon_hvsp` @tab Atmel AVR Dragon in HVSP mode

`dragon_isp` @tab Atmel AVR Dragon in ISP mode

`dragon_jtag` @tab Atmel AVR Dragon in JTAG mode

`dragon_pdi` @tab Atmel AVR Dragon in PDI mode

`dragon_pp` @tab Atmel AVR Dragon in PP mode

`ehajo-isp` @tab avr-isp-programmer from eHaJo,@* http://www.eHaJo.de

`flip1` @tab FLIP USB DFU protocol version 1 (doc7618)

`flip2` @tab FLIP USB DFU protocol version 2 (AVR4023)

`ft232r` @tab FT232R Synchronous BitBang

`ft245r` @tab FT245R Synchronous BitBang

`iseavrprog` @tab USBtiny-based USB programmer, https://github.com/IowaScaledEngineering/ckt-avrp

`jtag1` @tab Atmel JTAG ICE (mkI)

`jtag1slow` @tab Atmel JTAG ICE (mkI)

`jtag2` @tab Atmel JTAG ICE mkII

`jtag2avr32` @tab Atmel JTAG ICE mkII im AVR32 mode

`jtag2dw` @tab Atmel JTAG ICE mkII in debugWire mode

`jtag2fast` @tab Atmel JTAG ICE mkII

`jtag2isp` @tab Atmel JTAG ICE mkII in ISP mode

`jtag2pdi` @tab Atmel JTAG ICE mkII PDI mode

`jtag2slow` @tab Atmel JTAG ICE mkII

`jtag2updi` @tab JTAGv2 to UPDI bridge

`jtag3` @tab Atmel AVR JTAGICE3 in JTAG mode

`jtag3dw` @tab Atmel AVR JTAGICE3 in debugWIRE mode

`jtag3isp` @tab Atmel AVR JTAGICE3 in ISP mode

`jtag3pdi` @tab Atmel AVR JTAGICE3 in PDI mode

`jtag3updi` @tab Atmel AVR JTAGICE3 in UPDI mode

`jtagkey` @tab Amontec JTAGKey, JTAGKey-Tiny and JTAGKey2

`jtagmkI` @tab Atmel JTAG ICE (mkI)

`jtagmkII` @tab Atmel JTAG ICE mkII

`jtagmkII_avr32` @tab Atmel JTAG ICE mkII im AVR32 mode

`lm3s811` @tab Luminary Micro LM3S811 Eval Board (Rev. A)

`mib510` @tab Crossbow MIB510 programming board

`micronucleus` @tab Micronucleus Bootloader

`mkbutterfly` @tab Mikrokopter.de Butterfly

`nibobee` @tab NIBObee

`o-link` @tab O-Link, OpenJTAG from www.100ask.net

`openmoko` @tab Openmoko debug board (v3)

`pavr` @tab Jason Kyle's pAVR Serial Programmer

`pickit2` @tab MicroChip's PICkit2 Programmer

`pickit4_isp` @tab MPLAB(R) PICkit 4 in ISP mode

`pickit4_pdi` @tab MPLAB(R) PICkit 4 in PDI mode

`pickit4_updi` @tab MPLAB(R) PICkit 4 in UPDI mode

`pkobn_updi` @tab Curiosity nano (nEDBG) in UPDI mode

`ponyser` @tab design ponyprog serial, reset=!txd sck=rts mosi=dtr miso=cts

`powerdebugger` @tab Atmel PowerDebugger (ARM/AVR) in JTAG mode

`powerdebugger_dw` @tab Atmel PowerDebugger (ARM/AVR) in debugWire mode

`powerdebugger_isp` @tab Atmel PowerDebugger (ARM/AVR) in ISP mode

`powerdebugger_pdi` @tab Atmel PowerDebugger (ARM/AVR) in PDI mode

`powerdebugger_updi` @tab Atmel PowerDebugger (ARM/AVR) in UPDI mode

`serialupdi` @tab SerialUPDI

`siprog` @tab Lancos SI-Prog,@* http://www.lancos.com/siprogsch.html

`snap_isp` @tab MPLAB(R) SNAP in ISP mode

`snap_pdi` @tab MPLAB(R) SNAP in PDI mode

`snap_updi` @tab MPLAB(R) SNAP in UPDI mode

`stk500` @tab Atmel STK500

`stk500hvsp` @tab Atmel STK500 V2 in high-voltage serial programming mode

`stk500pp` @tab Atmel STK500 V2 in parallel programming mode

`stk500v1` @tab Atmel STK500 Version 1.x firmware

`stk500v2` @tab Atmel STK500 Version 2.x firmware

`stk600` @tab Atmel STK600

`stk600hvsp` @tab Atmel STK600 in high-voltage serial programming mode

`stk600pp` @tab Atmel STK600 in parallel programming mode

`tc2030` @tab Tag-Connect TC2030

`ttl232r` @tab FTDI TTL232R-5V with ICSP adapter

`tumpa` @tab TIAO USB Multi-Protocol Adapter

`UM232H` @tab FT232H based module from FTDI and Glyn.com.au

`uncompatino` @tab uncompatino with all pairs of pins shorted

`usbasp` @tab USBasp,@* http://www.fischl.de/usbasp/

`usbasp-clone` @tab Any usbasp clone with correct VID/PID

`usbtiny` @tab USBtiny simple USB programmer, https://learn.adafruit.com/usbtinyisp

`wiring` @tab Wiring

`xbee` @tab XBee Series 2 Over-The-Air (XBeeBoot)

`xplainedmini` @tab Atmel AVR XplainedMini in ISP mode

`xplainedmini_dw` @tab Atmel AVR XplainedMini in debugWIRE mode

`xplainedmini_updi` @tab Atmel AVR XplainedMini in UPDI mode

`xplainedpro` @tab Atmel AVR XplainedPro in JTAG mode

`xplainedpro_updi` @tab Atmel AVR XplainedPro in UPDI mode @end multitable

-C `config-file` Use the specified config file for configuration data. This file contains all programmer and part definitions that AVRDUDE knows about. If not specified, AVRDUDE looks for the configuration file in the following two locations:

** *<directory from which application loaded>/../etc/avrdude.conf*

**

<directory from which application loaded>/avrdude.conf

If not found there, the lookup procedure becomes platform dependent. On FreeBSD and Linux, AVRDUDE looks at */usr/local/etc/avrdude.conf*. See Appendix A for the method of searching on Windows.

If *config-file* is written as *+filename* then this file is read after the system wide and user configuration files. This can be used to add entries to the configuration without patching your system wide configuration file. It can be used several times, the files are read in same order as given on the command line.

-D Disable auto erase for flash. When the -U option with flash memory is specified, avrdude will perform a chip erase before starting any of the programming operations, since it generally is a mistake to program the flash without performing an erase first. This option disables that. Auto erase is not used for ATxmega devices as these devices can use page erase before writing each page so no explicit chip erase is required. Note however that any page not affected by the current operation will retain its previous contents.

-e Causes a chip erase to be executed. This will reset the contents of the flash ROM and EEPROM to the value `0xff', and clear all lock bits. Except for ATxmega devices which can use page erase, it is basically a prerequisite command before the flash ROM can be reprogrammed again. The only exception would be if the new contents would exclusively cause bits to be programmed from the value `1' to `0'. Note that in order to reprogram EERPOM cells, no explicit prior chip erase is required since the MCU provides an auto-erase cycle in that case before programming the cell.

-E `exitspec`[,...] By default, AVRDUDE leaves the parallel port in the same state at exit as it has been found at startup. This option modifies the state of the `/RESET' and `Vcc' lines the parallel port is left at, according to the exitspec arguments provided, as follows:

reset The `/RESET' signal will be left activated at program exit, that is it will be held low, in order to keep the MCU in reset state afterwards. Note in particular that the programming algorithm for the AT90S1200 device mandates that the `/RESET' signal is active before powering up the MCU, so in case an external power supply is used for this MCU type, a previous invocation of AVRDUDE with this option specified is one of the possible ways to guarantee this condition.

noreset The `/RESET' line will be deactivated at program exit, thus allowing the MCU target program to run while the programming hardware remains connected.

vcc This option will leave those parallel port pins active (i. e. high) that can be used to supply `Vcc' power to the MCU.

novcc This option will pull the `Vcc' pins of the parallel port down at program exit.

d_high This option will leave the 8 data pins on the parallel port active (i. e. high).

d_low

This option will leave the 8 data pins on the parallel port inactive (i. e. low).

Multiple *exitspec* arguments can be separated with commas.

-F Normally, AVRDUDE tries to verify that the device signature read from the part is reasonable before continuing. Since it can happen from time to time that a device has a broken (erased or overwritten) device signature but is otherwise operating normally, this options is provided to override the check. Also, for programmers like the Atmel STK500 and STK600 which can adjust parameters local to the programming tool (independent of an actual connection to a target controller), this option can be used together with *-t* to continue in terminal mode.

-i `delay` For bitbang-type programmers, delay for approximately *delay* microseconds between each bit state change. If the host system is very fast, or the target runs off a slow clock (like a 32 kHz crystal, or the 128 kHz internal RC oscillator), this can become necessary to satisfy the requirement that the ISP clock frequency must not be higher than 1/4 of the CPU clock frequency. This is implemented as a spin-loop delay to allow even for very short delays. On Unix-style operating systems, the spin loop is initially calibrated against a system timer, so the number of microseconds might be rather realistic, assuming a constant system load while AVRDUDE is running. On Win32 operating systems, a preconfigured number of cycles per microsecond is assumed that might be off a bit for very fast or very slow machines.

-l `logfile` Use *logfile* rather than *stderr* for diagnostics output. Note that initial diagnostic messages (during option parsing) are still written to *stderr* anyway.

-n No-write - disables actually writing data to the MCU (useful for debugging AVRDUDE).

-O Perform a RC oscillator run-time calibration according to Atmel application note AVR053. This is only supported on the STK500v2, AVRISP mkII, and JTAG ICE mkII hardware. Note that the result will be stored in the EEPROM cell at address 0.

-P `port` Use port to identify the device to which the programmer is attached. Normally, the default parallel port is used, but if the programmer type normally connects to the serial port, the default serial port will be used. See Appendix A, Platform Dependent Information, to find out the default port names for your platform. If you need to use a different parallel or serial port, use this option to specify the alternate port name.

On Win32 operating systems, the parallel ports are referred to as lpt1 through lpt3, referring to the addresses 0x378, 0x278, and 0x3BC, respectively. If the parallel port can be accessed through a different address, this address can be specified directly, using the common C language notation (i. e., hexadecimal values are prefixed by *0x*).

For the JTAG ICE mkII, if AVRDUDE has been built with libusb support, *port* may alternatively be specified as *usb`[:`serialno*]. In that case, the JTAG ICE mkII will be looked up on USB. If *serialno* is also specified, it will be matched against the serial number read from any JTAG ICE mkII found on USB. The match is done after stripping any existing colons from the given serial number, and right-to-left, so only the least significant bytes from the serial number need to be given. For a trick how to find out the serial numbers of all JTAG ICEs attached to USB, see *Example Command Line Invocations*.

As the AVRISP mkII device can only be talked to over USB, the very same method of specifying the port is required there.

For the USB programmer "AVR-Doper" running in HID mode, the port must be specified as *avrdoper*. Libhidapi support is required on Unix and Mac OS but not on Windows. For more information about AVR-Doper see http://www.obdev.at/avrusb/avrdoper.html.

For the USBtinyISP, which is a simplistic device not implementing serial numbers, multiple devices can be distinguished by their location in the USB hierarchy. See the respective Troubleshooting entry for examples.

For the XBee programmer the target MCU is to be programmed wirelessly over a ZigBee mesh using the XBeeBoot bootloader. The ZigBee 64-bit address for the target MCU's own XBee device must be supplied as a 16-character hexadecimal value as a port prefix, followed by the @code{} character, and the serial device to connect to a second directly contactable XBee device associated with the same mesh (with a default baud rate of 9600). This may look similar to: *0013a20000000001@/dev/tty.serial*.

For diagnostic purposes, if the target MCU with an XBeeBoot bootloader is connected directly to the serial port, the 64-bit address field can be omitted. In this mode the default baud rate will be 19200.

For programmers that attach to a serial port using some kind of higher level protocol (as opposed to bit-bang style programmers), *port* can be specified as *net*:*host*:*port*. In this case, instead of trying to open a local device, a TCP network connection to (TCP) *port* on *host* is established. Square brackets may be placed around *host* to improve readability for numeric IPv6 addresses (e.g. *net:[2001:db8::42]:1337*). The remote endpoint is assumed to be a terminal or console server that connects the network stream to a local serial port where the actual programmer has been attached to. The port is assumed to be properly configured, for example using a transparent 8-bit data connection without parity at 115200 Baud for a STK500.

Note: The ability to handle IPv6 hostnames and addresses is limited to Posix systems (by now).

-q Disable (or quell) output of the progress bar while reading or writing to the device. Specify it a second time for even quieter operation.

-u Disables the default behaviour of reading out the fuses three times before programming, then verifying at the end of programming that the fuses have not changed. If you want to change fuses you will need to specify this option, as avrdude will see the fuses have changed (even though you wanted to) and will change them back for your "safety". This option was designed to prevent cases of fuse bits magically changing (usually called *safemode*).

If one of the configuration files contains a line

default_safemode = no;

safemode is disabled by default. The *-u* option's effect is negated in that case, i. e. it *enables* safemode.

Safemode is always disabled for AVR32, Xmega and TPI devices.

-s Disable safemode prompting. When safemode discovers that one or more fuse bits have unintentionally changed, it will prompt for confirmation regarding whether or not it should attempt to recover the fuse bit(s). Specifying this flag disables the prompt and assumes that the fuse bit(s) should be recovered without asking for confirmation first.

-t Tells AVRDUDE to enter the interactive 'terminal' mode instead of up- or downloading files. See below for a detailed description of the terminal mode.

-U `memtype`:`op`:`filename`[:`format`] Perform a memory operation. Multiple *-U* options can be specified in order to operate on multiple memories on the same command-line invocation. The *memtype* field specifies the memory type to operate on. Use the *-v* option on the command line or the *part* command from terminal mode to display all the memory types supported by a particular device. Typically, a device's memory configuration at least contains the memory types *flash* and *eeprom*. All memory types currently known are:

calibration One or more bytes of RC oscillator calibration data.

eeprom The EEPROM of the device.

efuse The extended fuse byte.

flash The flash ROM of the device.

fuse The fuse byte in devices that have only a single fuse byte.

hfuse The high fuse byte.

lfuse The low fuse byte.

lock The lock byte.

signature The three device signature bytes (device ID).

fuse*N* The fuse bytes of ATxmega devices, *N* is an integer number for each fuse supported by the device.

application The application flash area of ATxmega devices.

apptable The application table flash area of ATxmega devices.

boot The boot flash area of ATxmega devices.

prodsig The production signature (calibration) area of ATxmega devices.

usersig

The user signature area of ATxmega devices.

The *op* field specifies what operation to perform:

r read the specified device memory and write to the specified file

w read the specified file and write it to the specified device memory

v

read the specified device memory and the specified file and perform a verify operation

The *filename* field indicates the name of the file to read or write. The *format* field is optional and contains the format of the file to read or write. Possible values are:

i Intel Hex

s Motorola S-record

r raw binary; little-endian byte order, in the case of the flash ROM data

e ELF (Executable and Linkable Format), the final output file from the linker; currently only accepted as an input file

m immediate mode; actual byte values specified on the command line, separated by commas or spaces in place of the *filename* field of the *-U* option. This is useful for programming fuse bytes without having to create a single-byte file or enter terminal mode. If the number specified begins with *0x*, it is treated as a hex value. If the number otherwise begins with a leading zero (*0*) it is treated as octal. Otherwise, the value is treated as decimal.

a auto detect; valid for input only, and only if the input is not provided at stdin.

d decimal; this and the following formats are only valid on output. They generate one line of output for the respective memory section, forming a comma-separated list of the values. This can be particularly useful for subsequent processing, like for fuse bit settings.

h hexadecimal; each value will get the string *0x* prepended.

o octal; each value will get a *0* prepended unless it is less than 8 in which case it gets no prefix.

b

binary; each value will get the string *0b* prepended.

The default is to use auto detection for input files, and raw binary format for output files.

Note that if *filename* contains a colon, the *format* field is no longer optional since the filename part following the colon would otherwise be misinterpreted as *format*.

When reading any kind of flash memory area (including the various sub-areas in Xmega devices), the resulting output file will be truncated to not contain trailing 0xFF bytes which indicate unprogrammed (erased) memory. Thus, if the entire memory is unprogrammed, this will result in an output file that has no contents at all.

As an abbreviation, the form *-U filename* is equivalent to specifying *-U flash:w:*`*filename*`:a*. This will only work if *filename* does not have a colon in it.

-v Enable verbose output. More *-v* options increase verbosity level.

-V Disable automatic verify check when uploading data.

-x `extended_param` Pass *extended_param* to the chosen programmer implementation as an extended parameter. The interpretation of the extended parameter depends on the programmer itself. See below for a list of programmers accepting extended parameters.

@page

2.2 Programmers accepting extended parameters

JTAG ICE mkII/3

AVR Dragon When using the JTAG ICE mkII/3 or AVR Dragon in JTAG mode, the following extended parameter is accepted:

@samp{jtagchain=UB,UA,BB,BA} Setup the JTAG scan chain for *UB* units before, *UA* units after, *BB* bits before, and *BA* bits after the target AVR, respectively. Each AVR unit within the chain shifts by 4 bits. Other JTAG units might require a different bit shift count.

AVR910 The AVR910 programmer type accepts the following extended parameter:

@samp{devcode=VALUE} Override the device code selection by using *VALUE* as the device code. The programmer is not queried for the list of supported device codes, and the specified *VALUE* is not verified but used directly within the *T* command sent to the programmer. *VALUE* can be specified using the conventional number notation of the C programming language.

@samp{no_blockmode} Disables the default checking for block transfer capability. Use @samp{no_blockmode} only if your @samp{AVR910} programmer creates errors during initial sequence.

BusPirate The BusPirate programmer type accepts the following extended parameters:

@samp{reset=cs,aux,aux2} The default setup assumes the BusPirate's CS output pin connected to the RESET pin on AVR side. It is however possible to have multiple AVRs connected to the same BP with MISO, MOSI and SCK lines common for all of them. In such a case one AVR should have its RESET connected to BusPirate's *CS* pin, second AVR's RESET connected to BusPirate's *AUX* pin and if your BusPirate has an *AUX2* pin (only available on BusPirate version v1a with firmware 3.0 or newer) use that to activate RESET on the third AVR.

It may be a good idea to decouple the BusPirate and the AVR's SPI buses from each other using a 3-state bus buffer. For example 74HC125 or 74HC244 are some good candidates with the latches driven by the appropriate reset pin (cs, aux or aux2). Otherwise the SPI traffic in one active circuit may interfere with programming the AVR in the other design.

@samp{spifreq=`0..7`} @multitable @columnfractions .05 .3

`0` @tab 30 kHz (default)

`1` @tab 125 kHz

`2` @tab 250 kHz

`3` @tab 1 MHz

`4` @tab 2 MHz

`5` @tab 2.6 MHz

`6` @tab 4 MHz

`7` @tab 8 MHz @end multitable

@samp{rawfreq=0..3} Sets the SPI speed and uses the Bus Pirate's binary 'raw-wire' mode instead of the default binary SPI mode:

@multitable @columnfractions .05 .3

`0` @tab 5 kHz

`1` @tab 50 kHz

`2` @tab 100 kHz (Firmware v4.2+ only)

`3` @tab 400 kHz (v4.2+) @end multitable

The only advantage of the 'raw-wire' mode is that different SPI frequencies are available. Paged writing is not implemented in this mode.

@samp{ascii} Attempt to use ASCII mode even when the firmware supports BinMode (binary mode). BinMode is supported in firmware 2.7 and newer, older FW's either don't have BinMode or their BinMode is buggy. ASCII mode is slower and makes the above @samp{reset=}, @samp{spifreq=} and @samp{rawfreq=} parameters unavailable. Be aware that ASCII mode is not guaranteed to work with newer firmware versions, and is retained only to maintain compatibility with older firmware versions.

@samp{nopagedwrite} Firmware versions 5.10 and newer support a binary mode SPI command that enables whole pages to be written to AVR flash memory at once, resulting in a significant write speed increase. If use of this mode is not desirable for some reason, this option disables it.

@samp{nopagedread} Newer firmware versions support in binary mode SPI command some AVR Extended Commands. Using the 'Bulk Memory Read from Flash' results in a significant read speed increase. If use of this mode is not desirable for some reason, this option disables it.

@samp{cpufreq=`125..4000`} This sets the *AUX* pin to output a frequency of *n* kHz. Connecting the *AUX* pin to the XTAL1 pin of your MCU, you can provide it a clock, for example when it needs an external clock because of wrong fuses settings. Make sure the CPU frequency is at least four times the SPI frequency.

@samp{serial_recv_timeout=`1…`} This sets the serial receive timeout to the given value. The timeout happens every time avrdude waits for the BusPirate prompt. Especially in ascii mode this happens very often, so setting a smaller value can speed up programming a lot. The default value is 100ms. Using 10ms might work in most cases.

Micronucleus bootloader When using the Micronucleus programmer type, the following optional extended parameter is accepted:

@samp{wait=`timeout`} If the device is not connected, wait for the device to be plugged in. The optional *timeout* specifies the connection time-out in seconds. If no time-out is specified, AVRDUDE will wait indefinitely until the device is plugged in.

Wiring When using the Wiring programmer type, the following optional extended parameter is accepted:

@samp{snooze=`0..32767`} After performing the port open phase, AVRDUDE will wait/snooze for *snooze* milliseconds before continuing to the protocol sync phase. No toggling of DTR/RTS is performed if *snooze* > 0.

PICkit2 Connection to the PICkit2 programmer: @multitable @columnfractions .05 .3

`(AVR)` @tab `(PICkit2)`

`RST` @tab `VPP/MCLR (1)`

`VDD` @tab `VDD Target (2) – possibly optional if AVR self powered`

`GND` @tab `GND (3)`

`MISO` @tab `PGD (4)`

`SCLK` @tab `PDC (5)`

`OSI` @tab `AUX (6)` @end multitable

Extended command line parameters:

@samp{clockrate=`rate`} Sets the SPI clocking rate in Hz (default is 100kHz). Alternately the -B or -i options can be used to set the period.

@samp{timeout=`usb-transaction-timeout`} Sets the timeout for USB reads and writes in milliseconds (default is 1500 ms).

USBasp Extended parameters:

@samp{section_config} Programmer will erase configuration section with option '-e' (chip erase), rather than entire chip. Only applicable to TPI devices (ATtiny 4/5/9/10/20/40).

xbee Extended parameters:

@samp{xbeeresetpin=`1..7`} Select the XBee pin *DIO<1..7>* that is connected to the MCU's '/RESET' line. The programmer needs to know which DIO pin to use to reset into the bootloader. The default (3) is the *DIO3* pin (XBee pin 17), but some commercial products use a different XBee pin.

The remaining two necessary XBee-to-MCU connections are not selectable - the XBee *DOUT* pin (pin 2) must be connected to the MCU's 'RXD' line, and the XBee *DIN* pin (pin 3) must be connected to the MCU's 'TXD' line.

@page

2.3 Example Command Line Invocations

Download the file *diag.hex* to the ATmega128 chip using the STK500 programmer connected to the default serial port:

```
@cartouche
% avrdude -p m128 -c stk500 -e -U flash:w:diag.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.03s

avrdude: Device signature = 0x1e9702
avrdude: erasing chip
avrdude: done.
avrdude: performing op: 1, flash, 0, diag.hex
avrdude: reading input file "diag.hex"
```

(continues on next page)

(continued from previous page)

```
avrdude: input file diag.hex auto detected as Intel Hex
avrdude: writing flash (19278 bytes):

Writing | ################################################## | 100% 7.60s

avrdude: 19456 bytes of flash written
avrdude: verifying flash memory against diag.hex:
avrdude: load data flash data from input file diag.hex:
avrdude: input file diag.hex auto detected as Intel Hex
avrdude: input file diag.hex contains 19278 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 6.83s

avrdude: verifying ...
avrdude: 19278 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done.  Thank you.

%
@end cartouche
```

@page

Upload the flash memory from the ATmega128 connected to the STK500 programmer and save it in raw binary format in the file named *c:/diag flash.bin*:

```
@cartouche
% avrdude -p m128 -c stk500 -U flash:r:"c:/diag flash.bin":r

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.03s

avrdude: Device signature = 0x1e9702
avrdude: reading flash memory:

Reading | ################################################## | 100% 46.10s

avrdude: writing output file "c:/diag flash.bin"

avrdude: safemode: Fuses OK

avrdude done.  Thank you.

%
@end cartouche
```

@page

Using the default programmer, download the file *diag.hex* to flash, *eeprom.hex* to EEPROM, and set the Extended, High, and Low fuse bytes to 0xff, 0x89, and 0x2e respectively:

```
@cartouche

% avrdude -p m128 -u -U flash:w:diag.hex \\
>                  -U eeprom:w:eeprom.hex \\
>                  -U efuse:w:0xff:m      \\
>                  -U hfuse:w:0x89:m      \\
>                  -U lfuse:w:0x2e:m

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.03s

avrdude: Device signature = 0x1e9702
avrdude: NOTE: FLASH memory has been specified, an erase cycle will be performed
         To disable this feature, specify the -D option.
avrdude: erasing chip
avrdude: reading input file "diag.hex"
avrdude: input file diag.hex auto detected as Intel Hex
avrdude: writing flash (19278 bytes):

Writing | ################################################## | 100% 7.60s

avrdude: 19456 bytes of flash written
avrdude: verifying flash memory against diag.hex:
avrdude: load data flash data from input file diag.hex:
avrdude: input file diag.hex auto detected as Intel Hex
avrdude: input file diag.hex contains 19278 bytes
avrdude: reading on-chip flash data:

Reading | ################################################## | 100% 6.84s

avrdude: verifying ...
avrdude: 19278 bytes of flash verified

[ ... other memory status output skipped for brevity ... ]

avrdude done.  Thank you.

%
@end cartouche
```

@page

Connect to the JTAG ICE mkII which serial number ends up in 1C37 via USB, and enter terminal mode:

```
@cartouche

% avrdude -c jtag2 -p m649 -P usb:1c:37 -t

avrdude: AVR device initialized and ready to accept instructions

Reading | ################################################## | 100% 0.03s

```

(continues on next page)

(continued from previous page)

```
avrdude: Device signature = 0x1e9603

[ ... terminal mode output skipped for brevity ... ]

avrdude done.  Thank you.

@end cartouche
```

List the serial numbers of all JTAG ICEs attached to USB. This is done by specifying an invalid serial number, and increasing the verbosity level.

```
@cartouche

% avrdude -c jtag2 -p m128 -P usb:xx -v
[...]
         Using Port            : usb:xxx
         Using Programmer      : jtag2
avrdude: usbdev_open(): Found JTAG ICE, serno: 00A000001C6B
avrdude: usbdev_open(): Found JTAG ICE, serno: 00A000001C3A
avrdude: usbdev_open(): Found JTAG ICE, serno: 00A000001C30
avrdude: usbdev_open(): did not find any (matching) USB device "usb:xxx"

@end cartouche
```

CHAPTER

THREE

TERMINAL MODE OPERATION

AVRDUDE has an interactive mode called *terminal mode* that is enabled by the *-t* option. This mode allows one to enter interactive commands to display and modify the various device memories, perform a chip erase, display the device signature bytes and part parameters, and to send raw programming commands. Commands and parameters may be abbreviated to their shortest unambiguous form. Terminal mode also supports a command history so that previously entered commands can be recalled and edited.

3.1 Terminal Mode Commands

The following commands are implemented:

dump `memtype` `addr` `nbytes` Read *nbytes* from the specified memory area, and display them in the usual hexadecimal and ASCII form.

dump Continue dumping the memory contents for another *nbytes* where the previous dump command left off.

write `memtype` `addr` `byte1`... `byteN` Manually program the respective memory cells, starting at address addr, using the values *byte1* through *byteN*. This feature is not implemented for bank-addressed memories such as the flash memory of ATMega devices.

erase Perform a chip erase.

send `b1` `b2` `b3` `b4` Send raw instruction codes to the AVR device. If you need access to a feature of an AVR part that is not directly supported by AVRDUDE, this command allows you to use it, even though AVRDUDE does not implement the command. When using direct SPI mode, up to 3 bytes can be omitted.

sig Display the device signature bytes.

spi Enter direct SPI mode. The *pgmled* pin acts as slave select. *Only supported on parallel bitbang programmers.*

part Display the current part settings and parameters. Includes chip specific information including all memory types supported by the device, read/write timing, etc.

pgm Return to programming mode (from direct SPI mode).

verbose [`level`] Change (when *level* is provided), or display the verbosity level. The initial verbosity level is controlled by the number of *-v* options given on the command line.

?

help Give a short on-line summary of the available commands.

quit Leave terminal mode and thus AVRDUDE.

In addition, the following commands are supported on the STK500 and STK600 programmer:

vtarg `voltage` Set the target's supply voltage to *voltage* Volts.

varef [`channel`] `voltage` Set the adjustable voltage source to *voltage* Volts. This voltage is normally used to drive the target's *Aref* input on the STK500 and STK600. The STK600 offers two reference voltages, which can be selected by the optional parameter *channel* (either 0 or 1).

fosc `freq`[`M`|`k`] Set the master oscillator to *freq* Hz. An optional trailing letter *M* multiplies by 1E6, a trailing letter *k* by 1E3.

fosc off Turn the master oscillator off.

sck `period` *STK500 and STK600 only:* Set the SCK clock period to *period* microseconds.

JTAG ICE only: Set the JTAG ICE bit clock period to *period* microseconds. Note that unlike STK500 settings, this setting will be reverted to its default value (approximately 1 microsecond) when the programming software signs off from the JTAG ICE. This parameter can also be used on the JTAG ICE mkII/3 to specify the ISP clock period when operating the ICE in ISP mode.

parms *STK500 and STK600 only:* Display the current voltage and master oscillator parameters.

JTAG ICE only: Display the current target supply voltage and JTAG bit clock rate/period.

3.2 Terminal Mode Examples

Display part parameters, modify eeprom cells, perform a chip erase:

```
@cartouche
% avrdude -p m128 -c stk500 -t

avrdude: AVR device initialized and ready to accept instructions
avrdude: Device signature = 0x1e9702
avrdude: current erase-rewrite cycle count is 52 (if being tracked)
avrdude> part
>>> part

AVR Part              : ATMEGA128
Chip Erase delay      : 9000 us
PAGEL                 : PD7
BS2                   : PA0
RESET disposition     : dedicated
RETRY pulse           : SCK
serial program mode   : yes
parallel program mode : yes
Memory Detail         :

                          Page                       Polled
  Memory Type Paged  Size   Size #Pages MinW  MaxW   ReadBack
  ----------- ------ ------ ---- ------ ----- ----- ---------
  eeprom      no       4096    8      0  9000  9000 0xff 0xff
  flash       yes    131072  256    512  4500  9000 0xff 0x00
  lfuse       no          1    0      0     0     0 0x00 0x00
  hfuse       no          1    0      0     0     0 0x00 0x00
  efuse       no          1    0      0     0     0 0x00 0x00
  lock        no          1    0      0     0     0 0x00 0x00
  calibration no          1    0      0     0     0 0x00 0x00
  signature   no          3    0      0     0     0 0x00 0x00
```

(continues on next page)

(continued from previous page)

```
avrdude> dump eeprom 0 16
>>> dump eeprom 0 16
0000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude> write eeprom 0 1 2 3 4
>>> write eeprom 0 1 2 3 4

avrdude> dump eeprom 0 16
>>> dump eeprom 0 16
0000  01 02 03 04 ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude> erase
>>> erase
avrdude: erasing chip
avrdude> dump eeprom 0 16
>>> dump eeprom 0 16
0000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude>
@end cartouche
```

Program the fuse bits of an ATmega128 (disable M103 compatibility, enable high speed external crystal, enable brown-out detection, slowly rising power). Note since we are working with fuse bits the -u (unsafe) option is specified, which allows you to modify the fuse bits. First display the factory defaults, then reprogram:

```
@cartouche
% avrdude -p m128 -u -c stk500 -t

avrdude: AVR device initialized and ready to accept instructions
avrdude: Device signature = 0x1e9702
avrdude: current erase-rewrite cycle count is 52 (if being tracked)
avrdude> d efuse
>>> d efuse
0000  fd                                                |.               |

avrdude> d hfuse
>>> d hfuse
0000  99                                                |.               |

avrdude> d lfuse
>>> d lfuse
0000  e1                                                |.               |

avrdude> w efuse 0 0xff
>>> w efuse 0 0xff

avrdude> w hfuse 0 0x89
>>> w hfuse 0 0x89

avrdude> w lfuse 0 0x2f
>>> w lfuse 0 0x2f
```

(continues on next page)

(continued from previous page)

```
avrdude>
@end cartouche
```

CHAPTER

FOUR

CONFIGURATION FILE

AVRDUDE reads a configuration file upon startup which describes all of the parts and programmers that it knows about. The advantage of this is that if you have a chip that is not currently supported by AVRDUDE, you can add it to the configuration file without waiting for a new release of AVRDUDE. Likewise, if you have a parallel port programmer that is not supported by AVRDUDE, chances are good that you can copy and existing programmer definition, and with only a few changes, make your programmer work with AVRDUDE.

AVRDUDE first looks for a system wide configuration file in a platform dependent location. On Unix, this is usually */usr/local/etc/avrdude.conf*, while on Windows it is usually in the same location as the executable file. The name of this file can be changed using the *-C* command line option. After the system wide configuration file is parsed, AVRDUDE looks for a per-user configuration file to augment or override the system wide defaults. On Unix, the per-user file is *.avrduderc* within the user's home directory. On Windows, this file is the *avrdude.rc* file located in the same directory as the executable.

4.1 AVRDUDE Defaults

***default_parallel* = "`*default-parallel-device*`";** Assign the default parallel port device. Can be overridden using the *-P* option.

***default_serial* = "`*default-serial-device*`";** Assign the default serial port device. Can be overridden using the *-P* option.

***default_programmer* = "`*default-programmer-id*`";** Assign the default programmer id. Can be overridden using the *-c* option.

***default_bitclock* = "`*default-bitclock*`";** Assign the default bitclock value. Can be overridden using the *-B* option.

4.2 Programmer Definitions

The format of the programmer definition is as follows:

```
programmer
    parent <id>                                 # <id> is a quoted string
    id       = <id1> [, <id2> [, <id3>] ...] ;  # <idN> are quoted strings
    desc     = <description> ;                  # quoted string
    type     = "par" | "stk500" | ... ;         # programmer type (see below for a list)
    baudrate = <num> ;                          # baudrate for serial ports
    vcc      = <num1> [, <num2> ... ] ;         # pin number(s)
    buff     = <num1> [, <num2> ... ] ;         # pin number(s)
    reset    = <num> ;                          # pin number
```

(continues on next page)

(continued from previous page)

```
    sck        = <num> ;                          # pin number
    mosi       = <num> ;                          # pin number
    miso       = <num> ;                          # pin number
    errled     = <num> ;                          # pin number
    rdyled     = <num> ;                          # pin number
    pgmled     = <num> ;                          # pin number
    vfyled     = <num> ;                          # pin number
    usbvid     = <hexnum>;                        # USB VID (Vendor ID)
    usbpid     = <hexnum> [, <hexnum> ...];       # USB PID (Product ID)
    usbdev     = <interface>;                     # USB interface or other device info
    usbvendor  = <vendorname>;                    # USB Vendor Name
    usbproduct = <productname>;                   # USB Product Name
    usbsn      = <serialno>;                      # USB Serial Number
  ;
```

If a parent is specified, all settings of it (except its ids) are used for the new programmer. These values can be changed by new setting them for the new programmer.

To invert a bit in the pin definitions, use = ~ <*num*>.

Not all programmer types can handle a list of USB PIDs.

Following programmer types are currently implemented:

@multitable @columnfractions .25 .6 * *arduino* @tab Arduino programmer * *avr910* @tab Serial programmers using protocol described in application note AVR910 * *avrftdi* @tab Interface to the MPSSE Engine of FTDI Chips using libftdi. * *buspirate* @tab Using the Bus Pirate's SPI interface for programming * *buspirate_bb* @tab Using the Bus Pirate's bitbang interface for programming * *butterfly* @tab Atmel Butterfly evaluation board; Atmel AppNotes AVR109, AVR911 * *butterfly_mk* @tab Mikrokopter.de Butterfly * *dragon_dw* @tab Atmel AVR Dragon in debugWire mode * *dragon_hvsp* @tab Atmel AVR Dragon in HVSP mode * *dragon_isp* @tab Atmel AVR Dragon in ISP mode * *dragon_jtag* @tab Atmel AVR Dragon in JTAG mode * *dragon_pdi* @tab Atmel AVR Dragon in PDI mode * *dragon_pp* @tab Atmel AVR Dragon in PP mode * *flip1* @tab FLIP USB DFU protocol version 1 (doc7618) * *flip2* @tab FLIP USB DFU protocol version 2 (AVR4023) * *ftdi_syncbb* @tab FT245R/FT232R Synchronous BitBangMode Programmer * *jtagmki* @tab Atmel JTAG ICE mkI * *jtagmkii* @tab Atmel JTAG ICE mkII * *jtagmkii_avr32* @tab Atmel JTAG ICE mkII in AVR32 mode * *jtagmkii_dw* @tab Atmel JTAG ICE mkII in debugWire mode * *jtagmkii_isp* @tab Atmel JTAG ICE mkII in ISP mode * *jtagmkii_pdi* @tab Atmel JTAG ICE mkII in PDI mode * *jtagice3* @tab Atmel JTAGICE3 * *jtagice3_pdi* @tab Atmel JTAGICE3 in PDI mode * *jtagice3_updi* @tab Atmel JTAGICE3 in UPDI mode * *jtagice3_dw* @tab Atmel JTAGICE3 in debugWire mode * *jtagice3_isp* @tab Atmel JTAGICE3 in ISP mode * *linuxgpio* @tab GPIO bitbanging using the Linux sysfs interface (not available) * *linuxspi* @tab SPI using Linux spidev driver (not available) * *micronucleus* @tab Micronucleus Bootloader * *par* @tab Parallel port bitbanging * *pickit2* @tab Microchip's PICkit2 Programmer * *serbb* @tab Serial port bitbanging * *serialupdi* @tab Driver for SerialUPDI programmers * *stk500* @tab Atmel STK500 Version 1.x firmware * *stk500generic* @tab Atmel STK500, autodetect firmware version * *stk500v2* @tab Atmel STK500 Version 2.x firmware * *stk500hvsp* @tab Atmel STK500 V2 in high-voltage serial programming mode * *stk500pp* @tab Atmel STK500 V2 in parallel programming mode * *stk600* @tab Atmel STK600 * *stk600hvsp* @tab Atmel STK600 in high-voltage serial programming mode * *stk600pp* @tab Atmel STK600 in parallel programming mode * *teensy* @tab Teensy Bootloader * *usbasp* @tab USBasp programmer, see http://www.fischl.de/usbasp/ * *usbtiny* @tab Driver for "usbtiny"-type programmers * *wiring* @tab http://wiring.org.co/, Basically STK500v2 protocol, with some glue to trigger the bootloader. * *xbee* @tab XBee Series 2 Over-The-Air (XBeeBoot) @end multitable

4.3 Part Definitions

```
part
    id                  = <id> ;                  # quoted string
    desc                = <description> ;         # quoted string
    family_id           = <description> ;         # quoted string
    has_jtag            = <yes/no> ;              # part has JTAG i/f
    has_debugwire       = <yes/no> ;              # part has debugWire i/f
    has_pdi             = <yes/no> ;              # part has PDI i/f
    has_updi            = <yes/no> ;              # part has UPDI i/f
    has_tpi             = <yes/no> ;              # part has TPI i/f
    devicecode          = <num> ;                 # numeric
    stk500_devcode      = <num> ;                 # numeric
    avr910_devcode      = <num> ;                 # numeric
    signature           = <num> <num> <num> ;     # signature bytes
    usbpid              = <num> ;                 # DFU USB PID
    reset               = dedicated | io;
    retry_pulse         = reset | sck;
    pgm_enable          = <instruction format> ;
    chip_erase          = <instruction format> ;
    chip_erase_delay = <num> ;                    # micro-seconds
    # STK500 parameters (parallel programming IO lines)
    pagel               = <num> ;                 # pin name in hex, i.e., 0xD7
    bs2                 = <num> ;                 # pin name in hex, i.e., 0xA0
    serial              = <yes/no> ;              # can use serial downloading
    parallel            = <yes/no/pseudo>;        # can use par. programming
    # STK500v2 parameters, to be taken from Atmel's XML files
    timeout             = <num> ;
    stabdelay           = <num> ;
    cmdexedelay         = <num> ;
    synchloops          = <num> ;
    bytedelay           = <num> ;
    pollvalue           = <num> ;
    pollindex           = <num> ;
    predelay            = <num> ;
    postdelay           = <num> ;
    pollmethod          = <num> ;
    mode                = <num> ;
    delay               = <num> ;
    blocksize           = <num> ;
    readsize            = <num> ;
    hvspcmdexedelay   = <num> ;
    # STK500v2 HV programming parameters, from XML
    pp_controlstack   = <num>, <num>, ...;        # PP only
    hvsp_controlstack = <num>, <num>, ...;        # HVSP only
    hventerstabdelay = <num>;
    progmodedelay       = <num>;                  # PP only
    latchcycles         = <num>;
    togglevtg           = <num>;
    poweroffdelay       = <num>;
    resetdelayms        = <num>;
    resetdelayus        = <num>;
```

(continues on next page)

(continued from previous page)

```
    hvleavestabdelay = <num>;
    resetdelay       = <num>;
    synchcycles      = <num>;                  # HVSP only
    chiperasepulsewidth = <num>;               # PP only
    chiperasepolltimeout = <num>;
    chiperasetime    = <num>;                  # HVSP only
    programfusepulsewidth = <num>;             # PP only
    programfusepolltimeout = <num>;
    programlockpulsewidth = <num>;             # PP only
    programlockpolltimeout = <num>;
    # JTAG ICE mkII parameters, also from XML files
    allowfullpagebitstream = <yes/no> ;
    enablepageprogramming = <yes/no> ;
    idr              = <num> ;                 # IO addr of IDR (OCD) reg.
    rampz            = <num> ;                 # IO addr of RAMPZ reg.
    spmcr            = <num> ;                 # mem addr of SPMC[S]R reg.
    eecr             = <num> ;                 # mem addr of EECR reg.
                                               # (only when != 0x3c)
    is_at90s1200     = <yes/no> ;              # AT90S1200 part
    is_avr32         = <yes/no> ;              # AVR32 part

    memory <memtype>
        paged            = <yes/no> ;          # yes / no
        size             = <num> ;             # bytes
        page_size        = <num> ;             # bytes
        num_pages        = <num> ;             # numeric
        min_write_delay = <num> ;              # micro-seconds
        max_write_delay = <num> ;              # micro-seconds
        readback_p1      = <num> ;             # byte value
        readback_p2      = <num> ;             # byte value
        pwroff_after_write = <yes/no> ;        # yes / no
        read             = <instruction format> ;
        write            = <instruction format> ;
        read_lo          = <instruction format> ;
        read_hi          = <instruction format> ;
        write_lo         = <instruction format> ;
        write_hi         = <instruction format> ;
        loadpage_lo      = <instruction format> ;
        loadpage_hi      = <instruction format> ;
        writepage        = <instruction format> ;
      ;
  ;
```

4.3.1 Parent Part

Parts can also inherit parameters from previously defined parts using the following syntax. In this case specified integer and string values override parameter values from the parent part. New memory definitions are added to the definitions inherited from the parent.

```
part parent <id>                                 # quoted string
    id                 = <id> ;                  # quoted string
    <any set of other parameters from the list above>
  ;
```

4.3.2 Instruction Format

Instruction formats are specified as a comma separated list of string values containing information (bit specifiers) about each of the 32 bits of the instruction. Bit specifiers may be one of the following formats:

1 The bit is always set on input as well as output

0 the bit is always clear on input as well as output

x the bit is ignored on input and output

a the bit is an address bit, the bit-number matches this bit specifier's position within the current instruction byte

a`N` the bit is the *N`th address bit, bit-number = N, i.e., `a12* is address bit 12 on input, *a0* is address bit 0.

i the bit is an input data bit

o the bit is an output data bit

Each instruction must be composed of 32 bit specifiers. The instruction specification closely follows the instruction data provided in Atmel's data sheets for their parts. For example, the EEPROM read and write instruction for an AT90S2313 AVR part could be encoded as:

```
read  = "1  0  1  0   0  0  0  0   x x x x  x x x x",
        "x a6 a5 a4  a3 a2 a1 a0   o o o o  o o o o";

write = "1  1  0  0   0  0  0  0   x x x x  x x x x",
        "x a6 a5 a4  a3 a2 a1 a0   i i i i  i i i i";
```

4.4 Other Notes

- The *devicecode* parameter is the device code used by the STK500 and is obtained from the software section (*avr061.zip*) of Atmel's AVR061 application note available from http://www.atmel.com/dyn/resources/prod_documents/doc2525.pdf.
- Not all memory types will implement all instructions.
- AVR Fuse bits and Lock bits are implemented as a type of memory.
- Example memory types are: *flash*, *eeprom*, *fuse*, *lfuse* (low fuse), *hfuse* (high fuse), *efuse* (extended fuse), *signature*, *calibration*, *lock*.
- The memory type specified on the AVRDUDE command line must match one of the memory types defined for the specified chip.

- The *pwroff_after_write* flag causes AVRDUDE to attempt to power the device off and back on after an unsuccessful write to the affected memory area if VCC programmer pins are defined. If VCC pins are not defined for the programmer, a message indicating that the device needs a power-cycle is printed out. This flag was added to work around a problem with the at90s4433/2333's; see the at90s4433 errata at:

  http://www.atmel.com/dyn/resources/prod_documents/doc1280.pdf

- The boot loader from application note AVR109 (and thus also the AVR Butterfly) does not support writing of fuse bits. Writing lock bits is supported, but is restricted to the boot lock bits (BLBxx). These are restrictions imposed by the underlying SPM instruction that is used to program the device from inside the boot loader. Note that programming the boot lock bits can result in a 'shoot-into-your-foot' scenario as the only way to unprogram these bits is a chip erase, which will also erase the boot loader code.

  The boot loader implements the 'chip erase' function by erasing the flash pages of the application section.

  Reading fuse and lock bits is fully supported.

  Note that due to the inability to write the fuse bits, the safemode functionality does not make sense for these boot loaders.

CHAPTER FIVE

PROGRAMMER SPECIFIC INFORMATION

5.1 Atmel STK600

The following devices are supported by the respective STK600 routing and socket card:

@multitable @columnfractions .25 .25 .5 @headitem Routing card @tab Socket card @tab Devices * _} @tab @code{STK600-ATTINY10_ @tab ATtiny4 ATtiny5 ATtiny9 ATtiny10 * _STK600-RC008T-2_ @tab _STK600-DIP_ @tab ATtiny11 ATtiny12 ATtiny13 ATtiny13A ATtiny25 ATtiny45 ATtiny85 * _STK600-RC008T-7_ @tab _STK600-DIP_ @tab ATtiny15 * _STK600-RC014T-42_ @tab _STK600-SOIC_ @tab ATtiny20 * _STK600-RC020T-1_ @tab _STK600-DIP_ @tab ATtiny2313 ATtiny2313A ATtiny4313 * _} @tab @code{STK600-TinyX3U_ @tab ATtiny43U * _STK600-RC014T-12_ @tab _STK600-DIP_ @tab ATtiny24 ATtiny44 ATtiny84 ATtiny24A ATtiny44A * _STK600-RC020T-8_ @tab _STK600-DIP_ @tab ATtiny26 ATtiny261 ATtiny261A ATtiny461 ATtiny861 ATtiny861A * _STK600-RC020T-43_ @tab _STK600-SOIC_ @tab ATtiny261 ATtiny261A ATtiny461 ATtiny461A ATtiny861 ATtiny861A * _STK600-RC020T-23_ @tab _STK600-SOIC_ @tab ATtiny87 ATtiny167 * _STK600-RC028T-3_ @tab _STK600-DIP_ @tab ATtiny28 * _STK600-RC028M-6_ @tab _STK600-DIP_ @tab ATtiny48 ATtiny88 ATmega8 ATmega8A ATmega48 ATmega88 ATmega168 ATmega48P ATmega48PA ATmega88P ATmega88PA ATmega168P ATmega168PA ATmega328P * _} @tab @code{QT600-ATTINY88-QT8_ @tab ATtiny88 * _STK600-RC040M-4_ @tab _STK600-DIP_ @tab ATmega8515 ATmega162 * _STK600-RC044M-30_ @tab _STK600-TQFP44_ @tab ATmega8515 ATmega162 * _STK600-RC040M-5_ @tab _STK600-DIP_ @tab ATmega8535 ATmega16 ATmega16A ATmega32 ATmega32A ATmega164P ATmega164PA ATmega324P ATmega324PA ATmega644 ATmega644P ATmega644PA ATmega1284P * _STK600-RC044M-31_ @tab _STK600-TQFP44_ @tab ATmega8535 ATmega16 ATmega16A ATmega32 ATmega32A ATmega164P ATmega164PA ATmega324P ATmega324PA ATmega644 ATmega644P ATmega644PA ATmega1284P * _} @tab @code{QT600-ATMEGA324-QM64_ @tab ATmega324PA * _STK600-RC032M-29_ @tab _STK600-TQFP32_ @tab ATmega8 ATmega8A ATmega48 ATmega88 ATmega168 ATmega48P ATmega48PA ATmega88P ATmega88PA ATmega168P ATmega168PA ATmega328P * _STK600-RC064M-9_ @tab _STK600-TQFP64_ @tab ATmega64 ATmega64A ATmega128 ATmega128A ATmega1281 ATmega2561 AT90CAN32 AT90CAN64 AT90CAN128 * _STK600-RC064M-10_ @tab _STK600-TQFP64_ @tab ATmega165 ATmega165P ATmega169 ATmega169P ATmega169PA ATmega325 ATmega325P ATmega329 ATmega329P ATmega645 ATmega649 ATmega649P * _STK600-RC100M-11_ @tab _STK600-TQFP100_ @tab ATmega640 ATmega1280 ATmega2560 * _} @tab @code{STK600-ATMEGA2560_ @tab ATmega2560 * _STK600-RC100M-18_ @tab _STK600-TQFP100_ @tab ATmega3250 ATmega3250P ATmega3290 ATmega3290P ATmega6450 ATmega6490 * _STK600-RC032U-20_ @tab _STK600-TQFP32_ @tab AT90USB82 AT90USB162 ATmega8U2 ATmega16U2 ATmega32U2 * _STK600-RC044U-25_ @tab _STK600-TQFP44_ @tab ATmega16U4 ATmega32U4 * _STK600-RC064U-17_ @tab _STK600-TQFP64_ @tab ATmega32U6 AT90USB646 AT90USB1286 AT90USB647 AT90USB1287 * _STK600-RCPWM-22_ @tab _STK600-TQFP32_ @tab ATmega32C1 ATmega64C1 ATmega16M1 ATmega32M1 ATmega64M1 * _STK600-RCPWM-19_ @tab _STK600-SOIC_ @tab AT90PWM2 AT90PWM3 AT90PWM2B AT90PWM3B AT90PWM216 AT90PWM316 * _STK600-RCPWM-26_ @tab _STK600-SOIC_ @tab AT90PWM81 * _STK600-RC044M-24_ @tab _STK600-TSSOP44_ @tab ATmega16HVB ATmega32HVB * _} @tab @code{STK600-HVE2_ @tab ATmega64HVE * _} @tab @code{STK600-ATMEGA128RFA1_ @tab ATmega128RFA1 * _STK600-RC100X-13_ @tab _STK600-TQFP100_ @tab ATxmega64A1 ATxmega128A1 ATxmega128A1_revD ATxmega128A1U * _} @tab @code{STK600-ATXMEGA1281A1_ @tab ATxmega128A1 * _} @tab @code{QT600-ATXMEGA128A1-QT16_ @tab ATxmega128A1 * _STK600-RC064X-14_ @tab _STK600-TQFP64_ @tab ATxmega64A3 ATxmega128A3 ATxmega256A3 ATxmega64D3 ATxmega128D3

ATxmega192D3 ATxmega256D3 * *STK600-RC064X-14* @tab *STK600-MLF64* @tab ATxmega256A3B * *STK600-RC044X-15* @tab *STK600-TQFP44* @tab ATxmega32A4 ATxmega16A4 ATxmega16D4 ATxmega32D4 * *} @tab @code{STK600-ATXMEGAT0* @tab ATxmega32T0 * *} @tab @code{STK600-uC3-144* @tab AT32UC3A0512 AT32UC3A0256 AT32UC3A0128 * *STK600-RCUC3A144-33* @tab *STK600-TQFP144* @tab AT32UC3A0512 AT32UC3A0256 AT32UC3A0128 * *STK600-RCuC3A100-28* @tab *STK600-TQFP100* @tab AT32UC3A1512 AT32UC3A1256 AT32UC3A1128 * *STK600-RCuC3B0-21* @tab *STK600-TQFP64-2* @tab AT32UC3B0256 AT32UC3B0512RevC AT32UC3B0512 AT32UC3B0128 AT32UC3B064 AT32UC3D1128 * *STK600-RCuC3B48-27* @tab *STK600-TQFP48* @tab AT32UC3B1256 AT32UC3B164 * *STK600-RCUC3A144-32* @tab *STK600-TQFP144* @tab AT32UC3A3512 AT32UC3A3256 AT32UC3A3128 AT32UC3A364 AT32UC3A3256S AT32UC3A3128S AT32UC3A364S * *STK600-RCUC3C0-36* @tab *STK600-TQFP144* @tab AT32UC3C0512 AT32UC3C0256 AT32UC3C0128 AT32UC3C064 * *STK600-RCUC3C1-38* @tab *STK600-TQFP100* @tab AT32UC3C1512 AT32UC3C1256 AT32UC3C1128 AT32UC3C164 * *STK600-RCUC3C2-40* @tab *STK600-TQFP64-2* @tab AT32UC3C2512 AT32UC3C2256 AT32UC3C2128 AT32UC3C264 * *STK600-RCUC3C0-37* @tab *STK600-TQFP144* @tab AT32UC3C0512 AT32UC3C0256 AT32UC3C0128 AT32UC3C064 * *STK600-RCUC3C1-39* @tab *STK600-TQFP100* @tab AT32UC3C1512 AT32UC3C1256 AT32UC3C1128 AT32UC3C164 * *STK600-RCUC3C2-41* @tab *STK600-TQFP64-2* @tab AT32UC3C2512 AT32UC3C2256 AT32UC3C2128 AT32UC3C264 * *STK600-RCUC3L0-34* @tab *STK600-TQFP48* @tab AT32UC3L064 AT32UC3L032 AT32UC3L016 * *} @tab @code{QT600-AT32UC3L-QM64* @tab AT32UC3L064 @end multitable

Ensure the correct socket and routing card are mounted *before* powering on the STK600. While the STK600 firmware ensures the socket and routing card mounted match each other (using a table stored internally in nonvolatile memory), it cannot handle the case where a wrong routing card is used, e. g. the routing card *STK600-RC040M-5* (which is meant for 40-pin DIP AVRs that have an ADC, with the power supply pins in the center of the package) was used but an ATmega8515 inserted (which uses the 'industry standard' pinout with Vcc and GND at opposite corners).

Note that for devices that use the routing card *STK600-RC008T-2*, in order to use ISP mode, the jumper for *AREF0* must be removed as it would otherwise block one of the ISP signals. High-voltage serial programming can be used even with that jumper installed.

The ISP system of the STK600 contains a detection against shortcuts and other wiring errors. AVRDUDE initiates a connection check before trying to enter ISP programming mode, and display the result if the target is not found ready to be ISP programmed.

High-voltage programming requires the target voltage to be set to at least 4.5 V in order to work. This can be done using *Terminal Mode*, see *Terminal Mode Operation*.

5.2 Atmel DFU bootloader using FLIP version 1

Bootloaders using the FLIP protocol version 1 experience some very specific behaviour.

These bootloaders have no option to access memory areas other than Flash and EEPROM.

When the bootloader is started, it enters a *security mode* where the only acceptable access is to query the device configuration parameters (which are used for the signature on AVR devices). The only way to leave this mode is a *chip erase*. As a chip erase is normally implied by the *-U* option when reprogramming the flash, this peculiarity might not be very obvious immediately.

Sometimes, a bootloader with security mode already disabled seems to no longer respond with sensible configuration data, but only 0xFF for all queries. As these queries are used to obtain the equivalent of a signature, AVRDUDE can only continue in that situation by forcing the signature check to be overridden with the *-F* option.

A *chip erase* might leave the EEPROM unerased, at least on some versions of the bootloader.

5.3 SerialUPDI programmer

SerialUPDI programmer can be used for programming UPDI-only devices using very simple serial connection. You can read more about the details here https://github.com/SpenceKonde/AVR-Guidance/blob/master/UPDI/jtag2updi.md

SerialUPDI programmer has been tested using FT232RL USB->UART interface with the following connection layout (copied from Spence Kohde's page linked above):

```
--------------------                                 To Target device
                DTR|                                  __________________
                Rx |--------------,------------------| UPDI---\\/\\/---------->
  Tx---/\\/\\/\\---Tx |-------|<|---'          .--------| Gnd    470 ohm
    resistor    Vcc|---------------------------------| Vcc
        1k      CTS|                     .`          |__________________
                Gnd|--------------------'
--------------------
```

There are several limitations in current SerialUPDI/AVRDUDE integration, listed below.

At the end of each run there are fuse values being presented to the user. For most of the UPDI-enabled devices these definitions (low fuse, high fuse, extended fuse) have no meaning whatsoever, as they have been simply replaced by array of fuses: fuse0..9. Therefore you can simply ignore this particular line of AVRDUDE output.

In connection to the above, *safemode* has no meaning in context of UPDI devices and should be ignored.

Currently available devices support only UPDI NVM programming model 0 and 2, but there is also experimental implementation of model 3 - not yet tested.

One of the core AVRDUDE features is verification of the connection by reading device signature prior to any operation, but this operation is not possible on UPDI locked devices. Therefore, to be able to connect to such a device, you have to provide *-F* to override this check.

Please note: using *-F* during write operation to locked device will force chip erase. Use carefully.

Another issue you might notice is slow performance of EEPROM writing using SerialUPDI for AVR Dx devices. This can be addressed by changing *avrdude.conf* section for this device - changing EEPROM page size to 0x20 (instead of default 1), like so:

```
#------------------------------------------------------------
# AVR128DB28
#------------------------------------------------------------

part parent    ".avrdx"
    id        = "avr128db28";
    desc      = "AVR128DB28";
    signature = 0x1E 0x97 0x0E;

    memory "flash"
        size      = 0x20000;
        offset    = 0x800000;
        page_size = 0x200;
        readsize  = 0x100;
    ;

    memory "eeprom"
        size      = 0x200;
```

(continues on next page)

(continued from previous page)

```
        offset    = 0x1400;
        page_size = 0x1;
        readsize  = 0x100;
    ;
;
```

USERROW memory has not been defined for new devices except for experimental addition for AVR128DB28. The point of USERROW is to provide ability to write configuration details to already locked device and currently SerialUPDI interface supports this feature, but it hasn't been tested on wide variety of chips. Treat this as something experimental at this point. Please note: on locked devices it's not possible to read back USERROW contents when written, so the automatic verification will most likely fail and to prevent error messages, use *-V*.

Please note that SerialUPDI interface is pretty new and some issues are to be expected. In case you run into them, please make sure to run the intended command with debug output enabled (*-v -v -v*) and provide this verbose output with your bug report. You can also try to perform the same action using *pymcuprog* (https://github.com/microchip-pic-avr-tools/pymcuprog) utility with *-v debug* and provide its output too. You will notice that both outputs are pretty similar, and this was implemented like that on purpose - it was supposed to make analysis of UPDI protocol quirks easier.

@appendix Platform Dependent Information

5.4 Unix

5.4.1 Unix Installation

To build and install from the source tarball on Unix like systems:

```
$ gunzip -c avrdude-6.99-20211218.tar.gz | tar xf -
$ cd avrdude-6.99-20211218
$ ./configure
$ make
$ su root -c 'make install'
```

The default location of the install is into */usr/local* so you will need to be sure that */usr/local/bin* is in your *PATH* environment variable.

If you do not have root access to your system, you can do the following instead:

```
$ gunzip -c avrdude-6.99-20211218.tar.gz | tar xf -
$ cd avrdude-6.99-20211218
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

FreeBSD Installation

AVRDUDE is installed via the FreeBSD Ports Tree as follows:

```
% su - root
# cd /usr/ports/devel/avrdude
# make install
```

If you wish to install from a pre-built package instead of the source, you can use the following instead:

```
% su - root
# pkg_add -r avrdude
```

Of course, you must be connected to the Internet for these methods to work, since that is where the source as well as the pre-built package is obtained.

Linux Installation

On rpm based Linux systems (such as RedHat, SUSE, Mandrake, etc.), you can build and install the rpm binaries directly from the tarball:

```
$ su - root
# rpmbuild -tb avrdude-6.99-20211218.tar.gz
# rpm -Uvh /usr/src/redhat/RPMS/i386/avrdude-6.99-20211218-1.i386.rpm
```

Note that the path to the resulting rpm package, differs from system to system. The above example is specific to RedHat.

5.4.2 Unix Configuration Files

When AVRDUDE is build using the default *–prefix* configure option, the default configuration file for a Unix system is located at */usr/local/etc/avrdude.conf*. This can be overridden by using the *-C* command line option. Additionally, the user’s home directory is searched for a file named *.avrduderc*, and if found, is used to augment the system default configuration file.

FreeBSD Configuration Files

When AVRDUDE is installed using the FreeBSD ports system, the system configuration file is always */usr/local/etc/avrdude.conf*.

Linux Configuration Files

When AVRDUDE is installed using from an rpm package, the system configuration file will be always be */etc/avrdude.conf*.

5.4.3 Unix Port Names

The parallel and serial port device file names are system specific. The following table lists the default names for a given system.

@multitable @columnfractions .30 .30 .30 * @strong{System} @tab @strong{Default Parallel Port} @tab @strong{Default Serial Port} * FreeBSD @tab */dev/ppi0* @tab */dev/cuad0* * Linux @tab */dev/parport0* @tab */dev/ttyS0* * Solaris @tab */dev/printers/0* @tab */dev/term/a* @end multitable

On FreeBSD systems, AVRDUDE uses the ppi(4) interface for accessing the parallel port and the sio(4) driver for serial port access.

On Linux systems, AVRDUDE uses the ppdev interface for accessing the parallel port and the tty driver for serial port access.

On Solaris systems, AVRDUDE uses the ecpp(7D) driver for accessing the parallel port and the asy(7D) driver for serial port access.

5.4.4 Unix Documentation

AVRDUDE installs a manual page as well as info, HTML and PDF documentation. The manual page is installed in */usr/local/man/man1* area, while the HTML and PDF documentation is installed in */usr/local/share/doc/avrdude* directory. The info manual is installed in */usr/local/info/avrdude.info*.

Note that these locations can be altered by various configure options such as *–prefix*.

5.5 Windows

5.5.1 Installation

A Windows executable of avrdude is included in WinAVR which can be found at http://sourceforge.net/projects/winavr. WinAVR is a suite of executable, open source software development tools for the AVR for the Windows platform.

There are two options to build avrdude from source under Windows. The first one is to use Cygwin (http://www.cygwin.com/).

To build and install from the source tarball for Windows (using Cygwin):

```
$ set PREFIX=<your install directory path>
$ export PREFIX
$ gunzip -c avrdude-6.99-20211218.tar.gz | tar xf -
$ cd avrdude-6.99-20211218
$ ./configure LDFLAGS="-static" --prefix=$PREFIX --datadir=$PREFIX
--sysconfdir=$PREFIX/bin --enable-versioned-doc=no
$ make
$ make install
```

Note that recent versions of Cygwin (starting with 1.7) removed the MinGW support from the compiler that is needed in order to build a native Win32 API binary that does not require to install the Cygwin library *cygwin1.dll* at run-time. Either try using an older compiler version that still supports MinGW builds, or use MinGW (http://www.mingw.org/) directly.

5.5.2 Configuration Files

Configuration file names

AVRDUDE on Windows looks for a system configuration file name of *avrdude.conf* and looks for a user override configuration file of *avrdude.rc*.

How AVRDUDE finds the configuration files.

AVRDUDE on Windows has a different way of searching for the system and user configuration files. Below is the search method for locating the configuration files:

- Only for the system configuration file: *<directory from which application loaded>/../etc/avrdude.conf*
- The directory from which the application loaded.
- The current directory.
- The Windows system directory. On Windows NT, the name of this directory is *SYSTEM32*.
- Windows NT: The 16-bit Windows system directory. The name of this directory is *SYSTEM*.
- The Windows directory.
- The directories that are listed in the PATH environment variable.

5.5.3 Port Names

Serial Ports

When you select a serial port (i.e. when using an STK500) use the Windows serial port device names such as: com1, com2, etc.

Parallel Ports

AVRDUDE will accept 3 Windows parallel port names: lpt1, lpt2, or lpt3. Each of these names corresponds to a fixed parallel port base address:

lpt1	0x378
lpt2	0x278
lpt3	0x3BC

On your desktop PC, lpt1 will be the most common choice. If you are using a laptop, you might have to use lpt3 instead of lpt1. Select the name of the port the corresponds to the base address of the parallel port that you want.

If the parallel port can be accessed through a different address, this address can be specified directly, using the common C language notation (i. e., hexadecimal values are prefixed by *0x*).

5.5.4 Documentation

AVRDUDE installs a manual page as well as info, HTML and PDF documentation. The manual page is installed in */usr/local/man/man1* area, while the HTML and PDF documentation is installed in */usr/local/share/doc/avrdude* directory. The info manual is installed in */usr/local/info/avrdude.info*.

Note that these locations can be altered by various configure options such as *–prefix* and *–datadir*.

@appendix Troubleshooting

In general, please report any bugs encountered via @* http://savannah.nongnu.org/bugs/?group=avrdude.

- Problem: I'm using a serial programmer under Windows and get the following error:

  avrdude: serial_open(): can't set attributes for device "com1",

  Solution: This problem seems to appear with certain versions of Cygwin. Specifying *"/dev/com1"* instead of *"com1"* should help.

- Problem: I'm using Linux and my AVR910 programmer is really slow.

  Solution (short): *setserial `port* low_latency`

  Solution (long): There are two problems here. First, the system may wait some time before it passes data from the serial port to the program. Under Linux the following command works around this (you may need root privileges for this).

  *setserial `port* low_latency`

  Secondly, the serial interface chip may delay the interrupt for some time. This behaviour can be changed by setting the FIFO-threshold to one. Under Linux this can only be done by changing the kernel source in *drivers/char/serial.c*. Search the file for *UART_FCR_TRIGGER_8* and replace it with *UART_FCR_TRIGGER_1*. Note that overall performance might suffer if there is high throughput on serial lines. Also note that you are modifying the kernel at your own risk.

- Problem: I'm not using Linux and my AVR910 programmer is really slow.

  Solutions: The reasons for this are the same as above. If you know how to work around this on your OS, please let us know.

- Problem: Updating the flash ROM from terminal mode does not work with the JTAG ICEs.

  Solution: None at this time. Currently, the JTAG ICE code cannot write to the flash ROM one byte at a time.

- Problem: Page-mode programming the EEPROM (using the -U option) does not erase EEPROM cells before writing, and thus cannot overwrite any previous value != 0xff.

  Solution: None. This is an inherent feature of the way JTAG EEPROM programming works, and is documented that way in the Atmel AVR datasheets. In order to successfully program the EEPROM that way, a prior chip erase (with the EESAVE fuse unprogrammed) is required. This also applies to the STK500 and STK600 in high-voltage programming mode.

- Problem: How do I turn off the *DWEN* fuse?

  Solution: If the *DWEN* (debugWire enable) fuse is activated, the */RESET* pin is not functional anymore, so normal ISP communication cannot be established. There are two options to deactivate that fuse again: high-voltage programming, or getting the JTAG ICE mkII talk debugWire, and prepare the target AVR to accept normal ISP communication again.

  The first option requires a programmer that is capable of high-voltage programming (either serial or parallel, depending on the AVR device), for example the STK500. In high-voltage programming mode, the */RESET* pin is activated initially using a 12 V pulse (thus the name *high voltage*), so the target AVR can subsequently be reprogrammed, and the *DWEN* fuse can be cleared. Typically, this operation cannot be performed while the AVR is located in the target circuit though.

The second option requires a JTAG ICE mkII that can talk the debugWire protocol. The ICE needs to be connected to the target using the JTAG-to-ISP adapter, so the JTAG ICE mkII can be used as a debugWire initiator as well as an ISP programmer. AVRDUDE will then be activated using the *jtag2isp* programmer type. The initial ISP communication attempt will fail, but AVRDUDE then tries to initiate a debugWire reset. When successful, this will leave the target AVR in a state where it can accept standard ISP communication. The ICE is then signed off (which will make it signing off from the USB as well), so AVRDUDE has to be called again afterwards. This time, standard ISP communication can work, so the *DWEN* fuse can be cleared.

The pin mapping for the JTAG-to-ISP adapter is:

@multitable @columnfractions .2 .2

- @strong{JTAG pin} @tab @strong{ISP pin}
- 1 @tab 3
- 2 @tab 6
- 3 @tab 1
- 4 @tab 2
- 6 @tab 5
- 9 @tab 4 @end multitable
- Problem: Multiple USBasp or USBtinyISP programmers connected simultaneously are not found.

  Solution: The USBtinyISP code supports distinguishing multiple programmers based on their bus:device connection tuple that describes their place in the USB hierarchy on a specific host. This tuple can be added to the *-P usb* option, similar to adding a serial number on other USB-based programmers.

  The actual naming convention for the bus and device names is operating-system dependent; AVRDUDE will print out what it found on the bus when running it with (at least) one *-v* option. By specifying a string that cannot match any existing device (for example, *-P usb:xxx*), the scan will list all possible candidate devices found on the bus.

  Examples:

  ```
  avrdude -c usbtiny -p atmega8 -P usb:003:025 (Linux)
  avrdude -c usbtiny -p atmega8 -P usb:/dev/usb:/dev/ugen1.3 (FreeBSD 8+)
  avrdude -c usbtiny -p atmega8 \\
    -P usb:bus-0:\\\\.\\libusb0-0001--0x1781-0x0c9f (Windows)
  ```

- Problem: I cannot do … when the target is in debugWire mode.

  Solution: debugWire mode imposes several limitations.

  The debugWire protocol is Atmel’s proprietary one-wire (plus ground) protocol to allow an in-circuit emulation of the smaller AVR devices, using the */RESET* line. DebugWire mode is initiated by activating the *DWEN* fuse, and then power-cycling the target. While this mode is mainly intended for debugging/emulation, it also offers limited programming capabilities. Effectively, the only memory areas that can be read or programmed in this mode are flash ROM and EEPROM. It is also possible to read out the signature. All other memory areas cannot be accessed. There is no *chip erase* functionality in debugWire mode; instead, while reprogramming the flash ROM, each flash ROM page is erased right before updating it. This is done transparently by the JTAG ICE mkII (or AVR Dragon). The only way back from debugWire mode is to initiate a special sequence of commands to the JTAG ICE mkII (or AVR Dragon), so the debugWire mode will be temporarily disabled, and the target can be accessed using normal ISP programming. This sequence is automatically initiated by using the JTAG ICE mkII or AVR Dragon in ISP mode, when they detect that ISP mode cannot be entered.

- Problem: I want to use my JTAG ICE mkII to program an Xmega device through PDI. The documentation tells me to use the *XMEGA PDI adapter for JTAGICE mkII* that is supposed to ship with the kit, yet I don’t have it.

Solution: Use the following pin mapping:

@multitable @columnfractions .2 .2 .2 .2

- @strong{JTAGICE} @tab @strong{Target} @tab @strong{Squid cab-} @tab @strong{PDI}
- @strong{mkII probe} @tab @strong{pins} @tab @strong{le colors} @tab @strong{header}
- 1 (TCK) @tab @tab Black @tab
- 2 (GND) @tab GND @tab White @tab 6
- 3 (TDO) @tab @tab Grey @tab
- 4 (VTref) @tab VTref @tab Purple @tab 2
- 5 (TMS) @tab @tab Blue @tab
- 6 (nSRST) @tab PDI_CLK @tab Green @tab 5
- 7 (N.C.) @tab @tab Yellow @tab
- 8 (nTRST) @tab @tab Orange @tab
- 9 (TDI) @tab PDI_DATA @tab Red @tab 1
- 10 (GND) @tab @tab Brown @tab @end multitable
- Problem: I want to use my AVR Dragon to program an Xmega device through PDI.

  Solution: Use the 6 pin ISP header on the Dragon and the following pin mapping:

  @multitable @columnfractions .2 .2
- @strong{Dragon} @tab @strong{Target}
- @strong{ISP Header} @tab @strong{pins}
- 1 (MISO) @tab PDI_DATA
- 2 (VCC) @tab VCC
- 3 (SCK) @tab
- 4 (MOSI) @tab
- 5 (RESET) @tab PDI_CLK / RST
- 6 (GND) @tab GND @end multitable
- Problem: I want to use my AVRISP mkII to program an ATtiny4/5/9/10 device through TPI. How to connect the pins?

  Solution: Use the following pin mapping:

  @multitable @columnfractions .2 .2 .2
- @strong{AVRISP} @tab @strong{Target} @tab @strong{ATtiny}
- @strong{connector} @tab @strong{pins} @tab @strong{pin #}
- 1 (MISO) @tab TPIDATA @tab 1
- 2 (VTref) @tab Vcc @tab 5
- 3 (SCK) @tab TPICLK @tab 3
- 4 (MOSI) @tab @tab
- 5 (RESET) @tab /RESET @tab 6

- 6 (GND) @tab GND @tab 2 @end multitable
- Problem: I want to program an ATtiny4/5/9/10 device using a serial/parallel bitbang programmer. How to connect the pins?

  Solution: Since TPI has only 1 pin for bi-directional data transfer, both *MISO* and *MOSI* pins should be connected to the *TPIDATA* pin on the ATtiny device. However, a 1K resistor should be placed between the *MOSI* and *TPIDATA*. The *MISO* pin connects to *TPIDATA* directly. The *SCK* pin is connected to *TPICLK*.

  In addition, the *Vcc*, */RESET* and *GND* pins should be connected to their respective ports on the ATtiny device.
- Problem: How can I use a FTDI FT232R USB-to-Serial device for bitbang programming?

  Solution: When connecting the FT232 directly to the pins of the target Atmel device, the polarity of the pins defined in the *programmer* definition should be inverted by prefixing a tilde. For example, the *dasa* programmer would look like this when connected via a FT232R device (notice the tildes in front of pins 7, 4, 3 and 8):

```
programmer
  id    = "dasa_ftdi";
  desc  = "serial port banging, reset=rts sck=dtr mosi=txd miso=cts";
  type  = serbb;
  reset = ~7;
  sck   = ~4;
  mosi  = ~3;
  miso  = ~8;
;
```

  Note that this uses the FT232 device as a normal serial port, not using the FTDI drivers in the special bitbang mode.
- Problem: My ATtiny4/5/9/10 reads out fine, but any attempt to program it (through TPI) fails. Instead, the memory retains the old contents.

  Solution: Mind the limited programming supply voltage range of these devices.

  In-circuit programming through TPI is only guaranteed by the datasheet at Vcc = 5 V.
- Problem: My ATxmega…A1/A2/A3 cannot be programmed through PDI with my AVR Dragon. Programming through a JTAG ICE mkII works though, as does programming through JTAG.

  Solution: None by this time (2010 Q1).

  It is said that the AVR Dragon can only program devices from the A4 Xmega sub-family.
- Problem: when programming with an AVRISPmkII or STK600, AVRDUDE hangs when programming files of a certain size (e.g. 246 bytes). Other (larger or smaller) sizes work though.

  Solution: This is a bug caused by an incorrect handling of zero-length packets (ZLPs) in some versions of the libusb 0.1 API wrapper that ships with libusb 1.x in certain Linux distributions. All Linux systems with kernel versions < 2.6.31 and libusb >= 1.0.0 < 1.0.3 are reported to be affected by this.

  See also: http://www.libusb.org/ticket/6
- Problem: after flashing a firmware that reduces the target's clock speed (e.g. through the *CLKPR* register), further ISP connection attempts fail.

  Solution: Even though ISP starts with pulling */RESET* low, the target continues to run at the internal clock speed as defined by the firmware running before. Therefore, the ISP clock speed must be reduced appropriately (to less than 1/4 of the internal clock speed) using the -B option before the ISP initialization sequence will succeed.

  As that slows down the entire subsequent ISP session, it might make sense to just issue a *chip erase* using the slow ISP clock (option *-e*), and then start a new session at higher speed. Option *-D* might be used there, to prevent another unneeded erase cycle.

INDEX