

AVRDUDE

A program for download/uploading AVR microcontroller flash and eeprom.
For AVRDUDE, Version 7.1, 3 January 2023.

by Brian S. Dean

Use <https://github.com/avrdudes/avrdude/issues> to report bugs and ask questions.
Copyright © Brian S. Dean, Jörg Wunsch

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Table of Contents

1	Introduction	1
1.1	History and Credits	4
2	Command Line Options	5
2.1	Option Descriptions	5
2.2	Programmers accepting extended parameters	23
2.3	Example Command Line Invocations	30
3	Terminal Mode Operation	34
3.1	Terminal Mode Commands	34
3.2	Terminal Mode Examples	37
4	Configuration File	41
4.1	AVRDUDE Defaults	41
4.2	Programmer Definitions	41
4.3	Part Definitions	44
4.3.1	Parent Part	46
4.3.2	Instruction Format	46
4.4	Other Notes	47
5	Programmer Specific Information	48
5.1	Atmel STK600	48
5.2	Atmel DFU bootloader using FLIP version 1	51
5.3	SerialUPDI programmer	51
Appendix A Platform Dependent Information ..		53
A.1	Unix	53
A.1.1	Unix Installation	53
A.1.1.1	FreeBSD Installation	53
A.1.1.2	Linux Installation	53
A.1.2	Unix Configuration Files	54
A.1.2.1	FreeBSD Configuration Files	54
A.1.2.2	Linux Configuration Files	54
A.1.3	Unix Port Names	54
A.1.4	Unix Documentation	54
A.2	Windows	54
A.2.1	Installation	54
A.2.2	Configuration Files	55
A.2.2.1	Configuration file names	55
A.2.2.2	How AVRDUDE finds the configuration files	55
A.2.3	Port Names	55

A.2.3.1	Serial Ports	55
A.2.3.2	Parallel Ports	55
A.2.4	Documentation	56
Appendix B	Troubleshooting.....	57
Concept Index	62

1 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 SDO, SDI, RESET and SCK pins of the AVR's SPI interface and use the linuxgpio programmer type. Older boards might use the labels MOSI for SDO and MISO for SDI. 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/blog/2013/01/06/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 chip select down when a transfer is not occurring 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 SDO, SDI, 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:

J8 pin	ISP pin	Name
21	1	SDI
-	2	Vcc - leave open
23	3	SCK
19	4	SDO
22	5	/RESET
25	6	GND

(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 SDO, and Bit 3 is SDI. 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, UPDI).

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.

Serial bootloaders that run a skeleton of the STK500 1.x protocol are supported via their own programmer type specification "arduino". This programmer works for the Arduino Uno Rev3 or any AVR that runs the Optiboot bootloader. The number of connection retry attempts can be specified as an extended parameter. See the section on *extended parameters* below for details.

Urprotocol is a leaner version of the STK500 1.x protocol that is designed to be backwards compatible with STK500 v1.x; it allows bootloaders to be much smaller, e.g., as implemented in the urboot project <https://github.com/stefanrueger/urboot>. The programmer type "urclock" caters for these urboot bootloaders. Owing to its backward compatibility, bootloaders that can be served by the arduino programmer can normally also be served by the urclock programmer. This may require specifying the size of (to AVRDUDE) *unknown* bootloaders in bytes using the `-x bootsize=<n>` option, which is necessary for the urclock programmer to enable it to protect the bootloader from being overwritten. If an unknown bootloader has EEPROM read/write capability then the option `-x eepromrw` informs `avrdude -c urclock` of that capability.

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 JTAG, TPI, 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 Konde’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. See Section 5.3 [SerialUPDI programmer], page 51, for more details and known issues.

The `jtag2updi` programmer is supported, and can program AVR’s 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 verifying the flash memory. See the section on *extended parameters* below for Micronucleus specific options.

The Teensy bootloader is supported for all AVR boards. As the bootloader does not support reading from flash memory, use the `-V` option to prevent `AVRDUDE` from verifying the flash memory. See the section on *extended parameters* below for Teensy 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.

For many years, the `AVRDUDE` source resided in public repositories on savannah.nongnu.org, where it continued 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.

In 2022, the project moved to Github (<https://github.com/avrdudes/avrdude/>).

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 text formatting was modeled after that of the `Simulavr` documentation by Ted Roth.

2 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 option tells AVRDUDE what part (MCU) is connected to the programmer. The *partno* parameter is the part's id listed in the configuration file. For currently supported MCU types use ? as partno, which will print a list of partno ids and official part names on the terminal. Both can be used with the -p option. 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. If -p ? is specified with a specific programmer, see -c below, then only those parts are output that the programmer expects to be able to handle, together with the programming interface(s) that can be used in that combination. In reality there can be deviations from this list, particularly if programming is directly via a bootloader. Currently, the following MCU types are understood:

uc3a0512	AT32UC3A0512
c128	AT90CAN128
c32	AT90CAN32
c64	AT90CAN64
pwm2	AT90PWM2
pwm216	AT90PWM216
pwm2b	AT90PWM2B
pwm3	AT90PWM3
pwm316	AT90PWM316
pwm3b	AT90PWM3B
1200	AT90S1200 (****)
2313	AT90S2313
2333	AT90S2333
2343	AT90S2343 (*)
4414	AT90S4414
4433	AT90S4433
4434	AT90S4434
8515	AT90S8515
8535	AT90S8535
usb1286	AT90USB1286
usb1287	AT90USB1287
usb162	AT90USB162
usb646	AT90USB646
usb647	AT90USB647

usb82	AT90USB82
m103	ATmega103
m128	ATmega128
m1280	ATmega1280
m1281	ATmega1281
m1284	ATmega1284
m1284p	ATmega1284P
m1284rfr2	ATmega1284RFR2
m128a	ATmega128A
m128rfa1	ATmega128RFA1
m128rfr2	ATmega128RFR2
m16	ATmega16
m1608	ATmega1608
m1609	ATmega1609
m161	ATmega161
m162	ATmega162
m163	ATmega163
m164a	ATmega164A
m164p	ATmega164P
m164pa	ATmega164PA
m165	ATmega165
m165a	ATmega165A
m165p	ATmega165P
m165pa	ATmega165PA
m168	ATmega168
m168a	ATmega168A
m168p	ATmega168P
m168pa	ATmega168PA
m168pb	ATmega168PB
m169	ATmega169
m169a	ATmega169A
m169p	ATmega169P
m169pa	ATmega169PA
m16a	ATmega16A
m16u2	ATmega16U2
m16u4	ATmega16U4
m2560	ATmega2560 (**)
m2561	ATmega2561 (**)
m2564rfr2	ATmega2564RFR2
m256rfr2	ATmega256RFR2
m32	ATmega32
m3208	ATmega3208
m3209	ATmega3209
m324a	ATmega324A
m324p	ATmega324P
m324pa	ATmega324PA
m324pb	ATmega324PB

m325	ATmega325
m3250	ATmega3250
m3250a	ATmega3250A
m3250p	ATmega3250P
m3250pa	ATmega3250PA
m325a	ATmega325A
m325p	ATmega325P
m325pa	ATmega325PA
m328	ATmega328
m328p	ATmega328P
m328pb	ATmega328PB
m329	ATmega329
m3290	ATmega3290
m3290a	ATmega3290A
m3290p	ATmega3290P
m3290pa	ATmega3290PA
m329a	ATmega329A
m329p	ATmega329P
m329pa	ATmega329PA
m32a	ATmega32A
m32m1	ATmega32M1
m32u2	ATmega32U2
m32u4	ATmega32U4
m406	ATmega406
m48	ATmega48
m4808	ATmega4808
m4809	ATmega4809
m48a	ATmega48A
m48p	ATmega48P
m48pa	ATmega48PA
m48pb	ATmega48PB
m64	ATmega64
m640	ATmega640
m644	ATmega644
m644a	ATmega644A
m644p	ATmega644P
m644pa	ATmega644PA
m644rfr2	ATmega644RFR2
m645	ATmega645
m6450	ATmega6450
m6450a	ATmega6450A
m6450p	ATmega6450P
m645a	ATmega645A
m645p	ATmega645P
m649	ATmega649
m6490	ATmega6490
m6490a	ATmega6490A

m6490p	ATmega6490P
m649a	ATmega649A
m649p	ATmega649P
m64a	ATmega64A
m64m1	ATmega64M1
m64rfr2	ATmega64RFR2
m8	ATmega8
m808	ATmega808
m809	ATmega809
m8515	ATmega8515
m8535	ATmega8535
m88	ATmega88
m88a	ATmega88A
m88p	ATmega88P
m88pa	ATmega88PA
m88pb	ATmega88PB
m8a	ATmega8A
m8u2	ATmega8U2
t10	ATtiny10
t102	ATtiny102
t104	ATtiny104
t11	ATtiny11 (***)
t12	ATtiny12
t13	ATtiny13
t13a	ATtiny13A
t15	ATtiny15
t1604	ATtiny1604
t1606	ATtiny1606
t1607	ATtiny1607
t1614	ATtiny1614
t1616	ATtiny1616
t1617	ATtiny1617
t1624	ATtiny1624
t1626	ATtiny1626
t1627	ATtiny1627
t1634	ATtiny1634
t1634r	ATtiny1634R
t167	ATtiny167
t20	ATtiny20
t202	ATtiny202
t204	ATtiny204
t212	ATtiny212
t214	ATtiny214
t2313	ATtiny2313
t2313a	ATtiny2313A
t24	ATtiny24
t24a	ATtiny24A

t25	ATtiny25
t26	ATtiny26
t261	ATtiny261
t261a	ATtiny261A
t28	ATtiny28
t3216	ATtiny3216
t3217	ATtiny3217
t3224	ATtiny3224
t3226	ATtiny3226
t3227	ATtiny3227
t4	ATtiny4
t40	ATtiny40
t402	ATtiny402
t404	ATtiny404
t406	ATtiny406
t412	ATtiny412
t414	ATtiny414
t416	ATtiny416
t417	ATtiny417
t424	ATtiny424
t426	ATtiny426
t427	ATtiny427
t4313	ATtiny4313
t43u	ATtiny43U
t44	ATtiny44
t441	ATtiny441
t44a	ATtiny44A
t45	ATtiny45
t461	ATtiny461
t461a	ATtiny461A
t48	ATtiny48
t5	ATtiny5
t804	ATtiny804
t806	ATtiny806
t807	ATtiny807
t814	ATtiny814
t816	ATtiny816
t817	ATtiny817
t824	ATtiny824
t826	ATtiny826
t827	ATtiny827
t828	ATtiny828
t828r	ATtiny828R
t84	ATtiny84
t841	ATtiny841
t84a	ATtiny84A
t85	ATtiny85

t861	ATtiny861
t861a	ATtiny861A
t87	ATtiny87
t88	ATtiny88
t9	ATtiny9
x128a1	ATxmega128A1
x128a1d	ATxmega128A1revD
x128a1u	ATxmega128A1U
x128a3	ATxmega128A3
x128a3u	ATxmega128A3U
x128a4	ATxmega128A4
x128a4u	ATxmega128A4U
x128b1	ATxmega128B1
x128b3	ATxmega128B3
x128c3	ATxmega128C3
x128d3	ATxmega128D3
x128d4	ATxmega128D4
x16a4	ATxmega16A4
x16a4u	ATxmega16A4U
x16c4	ATxmega16C4
x16d4	ATxmega16D4
x16e5	ATxmega16E5
x192a1	ATxmega192A1
x192a3	ATxmega192A3
x192a3u	ATxmega192A3U
x192c3	ATxmega192C3
x192d3	ATxmega192D3
x256a1	ATxmega256A1
x256a3	ATxmega256A3
x256a3b	ATxmega256A3B
x256a3bu	ATxmega256A3BU
x256a3u	ATxmega256A3U
x256c3	ATxmega256C3
x256d3	ATxmega256D3
x32a4	ATxmega32A4
x32a4u	ATxmega32A4U
x32c4	ATxmega32C4
x32d4	ATxmega32D4
x32e5	ATxmega32E5
x384c3	ATxmega384C3
x384d3	ATxmega384D3
x64a1	ATxmega64A1
x64a1u	ATxmega64A1U
x64a3	ATxmega64A3
x64a3u	ATxmega64A3U
x64a4	ATxmega64A4
x64a4u	ATxmega64A4U

x64b1	ATxmega64B1
x64b3	ATxmega64B3
x64c3	ATxmega64C3
x64d3	ATxmega64D3
x64d4	ATxmega64D4
x8e5	ATxmega8E5
avr128da28	AVR128DA28
avr128da32	AVR128DA32
avr128da48	AVR128DA48
avr128da64	AVR128DA64
avr128db28	AVR128DB28
avr128db32	AVR128DB32
avr128db48	AVR128DB48
avr128db64	AVR128DB64
avr16dd14	AVR16DD14
avr16dd20	AVR16DD20
avr16dd28	AVR16DD28
avr16dd32	AVR16DD32
avr16ea28	AVR16EA28
avr16ea32	AVR16EA32
avr16ea48	AVR16EA48
avr32da28	AVR32DA28
avr32da32	AVR32DA32
avr32da48	AVR32DA48
avr32db28	AVR32DB28
avr32db32	AVR32DB32
avr32db48	AVR32DB48
avr32dd14	AVR32DD14
avr32dd20	AVR32DD20
avr32dd28	AVR32DD28
avr32dd32	AVR32DD32
avr32ea28	AVR32EA28
avr32ea32	AVR32EA32
avr32ea48	AVR32EA48
avr64da28	AVR64DA28
avr64da32	AVR64DA32
avr64da48	AVR64DA48
avr64da64	AVR64DA64
avr64db28	AVR64DB28
avr64db32	AVR64DB32
avr64db48	AVR64DB48
avr64db64	AVR64DB64
avr64dd14	AVR64DD14
avr64dd20	AVR64DD20
avr64dd28	AVR64DD28
avr64dd32	AVR64DD32
avr64ea28	AVR64EA28

```

avr64ea32  AVR64EA32
avr64ea48  AVR64EA48
avr8ea28   AVR8EA28
avr8ea32   AVR8EA32
ucr2       deprecated,
lgt8f168p  LGT8F168P
lgt8f328p  LGT8F328P
lgt8f88p   LGT8F88P

```

(*) 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.

-p *wildcard/flags*

Run developer options for MCUs that are matched by *wildcard*. Whilst their main use is for developers some *flags* can be of utility for users, e.g., `avrdude -p m328p/S` outputs AVRDUDE's understanding of ATmega328P MCU properties; for more information run `avrdude -p x/h`.

-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, PDI, TPI, UPDI, or ISP interface. 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. Some programmers default their bit clock value to a 1 microsecond bit clock period, suitable for target MCUs running at 4 MHz clock and above. Slower MCUs need a correspondingly higher bit clock period. Some programmers reset their bit clock value to the default value when the programming software signs off, whilst others store the last used bit clock value. It is recommended to always specify the bit clock if read/write speed is important. You can use the 'default_bitclock' keyword in your `~/.config/avrdude/avrdude.rc` or `~/.avrduderc` configuration file to assign a default value to keep from having to specify this option on every invocation.

-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. If `-c ?` is specified with a specific part, see `-p` above, then only those programmers are output that expect to be able to handle this part, together with the programming interface(s) that can be used in that combination. In reality there can be deviations from this list, particularly if programming is directly via a bootloader. Currently, the following programmer ids are understood and supported:

2232hio	2232hio based on FT2232H with buffer and LEDs
4232h	FT4232H based generic programmer
adafruit_gemma	Adafruit Trinket Gemma bootloader disguised as USBtiny
arduino	Arduino for bootloader using STK500 v1 protocol
arduino-ft232r	Arduino: FT232R connected to ISP
arduino_gemma	Arduino Gemma bootloader disguised as USBtiny
arduinoisp	Arduino ISP Programmer
arduinoisporg	Arduino ISP Programmer
atmelice	Atmel-ICE (ARM/AVR) in JTAG mode
atmelice_dw	Atmel-ICE (ARM/AVR) in debugWIRE mode
atmelice_isp	Atmel-ICE (ARM/AVR) in ISP mode
atmelice_pdi	Atmel-ICE (ARM/AVR) in PDI mode
atmelice_tpi	Atmel-ICE (ARM/AVR) in TPI mode
atmelice_updi	Atmel-ICE (ARM/AVR) in UPDI mode
avr109	Atmel for bootloader using AppNote AVR109
avr910	Atmel Low Cost Serial Programmer
avr911	Atmel for bootloader using AppNote AVR911 AVROSP
avrftdi	FT2232D based generic programmer
avrisp	Atmel AVR ISP
avrisp2	Atmel AVR ISP mkII
avrispmkII	Atmel AVR ISP mkII
avrispv2	Atmel AVR ISP v2
buspirate	The Bus Pirate
buspirate_bb	The Bus Pirate (bitbang interface, supports TPI)
butterfly	Atmel for bootloader (Butterfly Development Board)
butterfly_mk	Mikrokoetter.de Butterfly for bootloader
bwmega	BitWizard ftdi_atmega builtin programmer
c232hm	C232HM cable from FTDI
c2n232i	serial port banging, reset=dtr sck=!rts sdo=!txd sdi=!cts
dasa	serial port banging, reset=rts sck=dtr sdo=txd sdi=cts

dasa3	serial port banging, reset=!dtr sck=rts sdo=txd sdi=cts
diecimila	alias for arduino-ft232r
digilent-hs2	Digilent JTAG HS2 (MPSSE)
dragon_dw	Atmel AVR Dragon in debugWire mode
dragon_hvsp	Atmel AVR Dragon in HVSP mode
dragon_isp	Atmel AVR Dragon in ISP mode
dragon_jtag	Atmel AVR Dragon in JTAG mode
dragon_pdi	Atmel AVR Dragon in PDI mode
dragon_pp	Atmel AVR Dragon in PP mode
ehajo-isp	avr-isp-programmer from eHaJo, http://www.eHaJo.de
flip1	FLIP for bootloader using USB DFU protocol version 1 (doc7618)
flip2	FLIP for bootloader using USB DFU protocol version 2 (AVR4023)
ft2232h	FT2232H based generic programmer
ft232h	FT232H based generic programmer
ft232r	FT232R based generic programmer
ft245r	FT245R based generic programmer
ft4232h	FT4232H based generic programmer
iseavrprog	USBTiny-based programmer, https://iascaled.com
jtag1	Atmel JTAG ICE (mkI)
jtag1slow	Atmel JTAG ICE (mkI)
jtag2	Atmel JTAG ICE mkII
jtag2avr32	Atmel JTAG ICE mkII in AVR32 mode
jtag2dw	Atmel JTAG ICE mkII in debugWire mode
jtag2fast	Atmel JTAG ICE mkII
jtag2isp	Atmel JTAG ICE mkII in ISP mode
jtag2pdi	Atmel JTAG ICE mkII in PDI mode
jtag2slow	Atmel JTAG ICE mkII
jtag2updi	JTAGv2 to UPDI bridge
jtag3	Atmel AVR JTAGICE3 in JTAG mode
jtag3dw	Atmel AVR JTAGICE3 in debugWIRE mode
jtag3isp	Atmel AVR JTAGICE3 in ISP mode
jtag3pdi	Atmel AVR JTAGICE3 in PDI mode
jtag3updi	Atmel AVR JTAGICE3 in UPDI mode
jtagkey	Amontec JTAGKey, JTAGKey-Tiny and JTAGKey2
jtagmkI	Atmel JTAG ICE (mkI)
jtagmkII	Atmel JTAG ICE mkII
jtagmkII_avr32	Atmel JTAG ICE mkII in AVR32 mode
ktlink	KT-LINK FT2232H interface with IO switching and voltage buffers.
lm3s811	Luminary Micro LM3S811 Eval Board (Rev. A)
mib510	Crossbow MIB510 programming board

<code>micronucleus</code>	Micronucleus for bootloader
<code>mkbutterfly</code>	Mikrokopter.de Butterfly for bootloader
<code>nibobee</code>	NIBObec
<code>o-link</code>	O-Link, OpenJTAG from www.100ask.net
<code>openmoko</code>	Openmoko debug board (v3)
<code>pavr</code>	Jason Kyle's pAVR Serial Programmer
<code>pickit2</code>	MicroChip's PICKit2 Programmer
<code>pickit4</code>	MPLAB(R) PICKit 4 in JTAG mode
<code>pickit4_isp</code>	MPLAB(R) PICKit 4 in ISP mode
<code>pickit4_pdi</code>	MPLAB(R) PICKit 4 in PDI mode
<code>pickit4_tpi</code>	MPLAB(R) PICKit 4 in TPI mode
<code>pickit4_updi</code>	MPLAB(R) PICKit 4 in UPDI mode
<code>pkobn_updi</code>	Curiosity nano (nEDBG) in UPDI mode
<code>ponyser</code>	design ponyprog serial, reset=!txd sck=rts sdo=dtr sdi=cts
<code>powerdebugger</code>	Atmel PowerDebugger (ARM/AVR) in JTAG mode
<code>powerdebugger_dw</code>	Atmel PowerDebugger (ARM/AVR) in debugWire mode
<code>powerdebugger_isp</code>	Atmel PowerDebugger (ARM/AVR) in ISP mode
<code>powerdebugger_pdi</code>	Atmel PowerDebugger (ARM/AVR) in PDI mode
<code>powerdebugger_tpi</code>	Atmel PowerDebugger (ARM/AVR) in TPI mode
<code>powerdebugger_updi</code>	Atmel PowerDebugger (ARM/AVR) in UPDI mode
<code>serialupdi</code>	SerialUPDI
<code>siprog</code>	Lancos SI-Prog < http://www.lancos.com/siprogsch.html >
<code>snap</code>	MPLAB(R) Snap in JTAG mode
<code>snap_isp</code>	MPLAB(R) SNAP in ISP mode
<code>snap_pdi</code>	MPLAB(R) SNAP in PDI mode
<code>snap_tpi</code>	MPLAB(R) SNAP in TPI mode
<code>snap_updi</code>	MPLAB(R) SNAP in UPDI mode
<code>stk500</code>	Atmel STK500
<code>stk500hvsp</code>	Atmel STK500 v2 in high-voltage serial program- ming mode
<code>stk500pp</code>	Atmel STK500 v2 in parallel programming mode
<code>stk500v1</code>	Atmel STK500 version 1.x firmware
<code>stk500v2</code>	Atmel STK500 version 2.x firmware
<code>stk600</code>	Atmel STK600
<code>stk600hvsp</code>	Atmel STK600 in high-voltage serial programming mode
<code>stk600pp</code>	Atmel STK600 in parallel programming mode
<code>tc2030</code>	Tag-Connect TC2030
<code>teensy</code>	Teensy for bootloader
<code>tigard</code>	Tigard interface board
<code>ttl232r</code>	FTDI TTL232R-5V with ICSP adapter
<code>tumpa</code>	TIAO USB Multi-Protocol Adapter

<code>um232h</code>	UM232H module from FTDI
<code>uncompatino</code>	uncompatino with all pairs of pins shorted
<code>urclock</code>	Urclock programmer for urboot bootloaders using urprotocol
<code>usbasp</code>	USBasp, http://www.fischl.de/usbasp/
<code>usbasp-clone</code>	Any usbasp clone with correct VID/PID
<code>usbtiny</code>	USBTiny simple USB programmer, https://learn.adafruit.com/usbtinyisp
<code>wiring</code>	Wiring for bootloader using STK500 v2 protocol
<code>xbee</code>	XBee for Series 2 Over-The-Air (XBeeBoot) bootloader using STK500 v1 protocol
<code>xplainedmini</code>	Atmel AVR XplainedMini in ISP mode
<code>xplainedmini_dw</code>	Atmel AVR XplainedMini in debugWIRE mode
<code>xplainedmini_tpi</code>	Atmel AVR XplainedMini in TPI mode
<code>xplainedmini_updi</code>	Atmel AVR XplainedMini in UPDI mode
<code>xplainedpro</code>	Atmel AVR XplainedPro in JTAG mode
<code>xplainedpro_pdi</code>	Atmel AVR XplainedPro in PDI mode
<code>xplainedpro_updi</code>	Atmel AVR XplainedPro in UPDI mode

-c wildcard/flags

Run developer options for programmers that are matched by *wildcard*. Whilst their main use is for developers some *flags* can be of utility for users, e.g., `avrdude -c usbtiny/S` shows AVRDUDE's understanding of usbtiny's properties; for more information run `avrdude -c x/h`.

-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:

1. <directory from which application loaded>/../etc/avrdude.conf
2. <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.

-A

Disable the automatic removal of trailing-0xFF sequences in file input that is to be programmed to flash and in AVR reads from flash memory. Normally, trailing 0xFFs can be discarded, as flash programming requires the memory be erased to 0xFF beforehand. -A should be used when the programmer hardware, or bootloader software for that matter, does not carry out chip erase and instead handles the memory erase on a page level. The popular Arduino bootloader exhibits this behaviour; for this reason -A is engaged by default when specifying `-c arduino`.

- 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. Setting **-D** implies **-A**.
- 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. reset is supported by the linuxspi and flip2 programmer options, as well as all parallel port based programmers. |
| noreset | The '/RESET' line will be deactivated at program exit, thus allowing the MCU target program to run while the programming hardware remains connected. noreset is supported by the linuxspi and flip2 programmer options, as well as all parallel port based programmers. |
| 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. Moreover, the option allows to continue despite failed initialization of connection between a programmer and a target.
- 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 Section 2.3 [Example Command Line Invocations], page 30.

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 See Appendix B [Troubleshooting], page 57, 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 @ 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: `0013a20000000001dev/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.
- `-s, -u` These options used to control the obsolete "safemode" feature which is no longer present. They are silently ignored for backwards compatibility.
- `-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

I Intel Hex with comments on download and tolerance of checksum errors on upload

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 <i>filename</i> field of the <code>-U</code> 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 <code>0x</code> , 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 <code>0x</code> prepended. Only valid on output.
o	octal; each value will get a <code>0</code> prepended unless it is less than 8 in which case it gets no prefix. Only valid on output.
b	binary; each value will get the string <code>0b</code> prepended. Only valid on output.

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 pro-

grammer itself. See below for a list of programmers accepting extended parameters.

2.2 Programmers accepting extended parameters

JTAG ICE mkII/3

Atmel-ICE

PICkit 4

MPLAB SNAP

Power Debugger

AVR Dragon

When using the JTAG ICE mkII, JTAGICE3, Atmel-ICE, PICkit 4, MPLAB SNAP, Power Debugger or AVR Dragon in JTAG mode, the following extended parameter is accepted:

`'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.

The PICkit 4 and the Power Debugger also supports high-voltage UPDI programming. This is used to enable a UPDI pin that has previously been set to RESET or GPIO mode. High-voltage UPDI can be utilized by using an extended parameter:

`'hvupdi'` Enable high-voltage UPDI initialization for targets that supports this.

AVR910

The AVR910 programmer type accepts the following extended parameter:

`'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.

`'no_blockmode'`

Disables the default checking for block transfer capability. Use `'no_blockmode'` only if your 'AVR910' programmer creates errors during initial sequence.

Arduino

The Arduino programmer type accepts the following extended parameter:

`'attempts=VALUE'`

Override the default number of connection retry attempt by using *VALUE*.

Urclock

The urclock programmer type accepts the following extended parameters:

- `'showall'`
Show all info for the connected part, then exit. The `-xshow...` options below can be used to assemble a bespoke response consisting of a subset (or only one item) of all available relevant information about the connected part and bootloader.
- `'showid'` Show a unique Urclock ID stored in either flash or EEPROM of the MCU, then exit.
- `'id=<E|F>.<addr>.<len>'`
Historically, the Urclock ID was a six-byte unique little-endian number stored in Urclock boards at EEPROM address 257. The location of this number can be set by the `-xid=<E|F>.<addr>.<len>` extended parameter. E stands for EEPROM and F stands for flash. A negative address `addr` counts from the end of EEPROM and flash, respectively. The length `len` of the Urclock ID can be between 1 and 8 bytes.
- `'showdate'`
Show the last-modified date of the input file for the flash application, then exit. If the input file was `stdin`, the date will be that of the programming. Date and filename are part of the metadata that the urclock programmer stores by default in high flash just under the bootloader; see also `-xnometadata`.
- `'showfilename'`
Show the input filename (or title) of the last flash writing session, then exit.
- `'title=<string>'`
When set, `<string>` will be used in lieu of the input filename. The maximum string length for the title/filename field is 254 bytes including terminating `nul`.
- `'showapp'`
Show the size of the programmed application, then exit.
- `'showstore'`
Show the size of the unused flash between the application and metadata, then exit.
- `'showmeta'`
Show the size of the metadata just below the bootloader, then exit.
- `'showboot'`
Show the size of the bootloader, then exit.
- `'showversion'`
Show bootloader version and capabilities, then exit.
- `'showvector'`
Show the vector number and name of the interrupt table vector used by the bootloader for starting the application, then exit. For

hardware-supported bootloaders this will be vector 0 (Reset), and for vector bootloaders this will be any other vector number of the interrupt vector table or the slot just behind the vector table with the name `VBL_ADDITIONAL_VECTOR`.

`'showpart'`

Show the part for which the bootloader was compiled, then exit.

`'bootsize=<size>'`

Manual override for bootloader size. Urboot bootloaders put the number of used bootloader pages into a table at the top of the bootloader section, ie, typically top of flash, so the urclock programmer can look up the bootloader size itself. In backward-compatibility mode, when programming via other bootloaders, this option can be used to tell the programmer the size, and therefore the location, of the bootloader.

`'vectornum=<arg>'`

Manual override for vector number. Urboot bootloaders put the vector number used by a vector bootloader into a table at the top of flash, so this option is normally not needed for urboot bootloaders. However, it is useful in backward-compatibility mode (or when the urboot bootloader does not offer flash read). Specifying a vector number in these circumstances implies a vector bootloader whilst the default assumption would be a hardware-supported bootloader.

`'eepromrw'`

Manual override for asserting EEPROM read/write capability. Not normally needed for urboot bootloaders, but useful for in backward-compatibility mode if the bootloader offers EEPROM read/write.

`'emulate_ce'`

If an urboot bootloader does not offer a chip erase command it will tell the urclock programmer so during handshake. In this case the urclock programmer emulates a chip erase, if warranted by user command line options, by filling the remainder of unused flash below the bootloader with `0xff`. If this option is specified, the urclock programmer will assume that the bootloader cannot erase the chip itself. The option is useful for backwards-compatible bootloaders that do not implement chip erase.

`'restore'`

Upload unchanged flash input files and trim below the bootloader if needed. This is most useful when one has a backup of the full flash and wants to play that back onto the device. No metadata are written in this case and no vector patching happens either if it is a vector bootloader. However, for vector bootloaders, even under the option `-xrestore` an input file will not be uploaded for which the reset vector does not point to the vector bootloader. This is to avoid writing an input file to the device that would render the

vector bootloader not functional as it would not be reached after reset.

- `'initstore'`
On writing to flash fill the store space between the flash application and the metadata section with 0xff.
- `'nofilename'`
On writing to flash do not store the application input filename (nor a title).
- `'nodate'` On writing to flash do not store the application input filename (nor a title) and no date either.
- `'nometadata'`
On writing to flash do not store any metadata. The full flash below the bootloader is available for the application. In particular, no data store frame is programmed.
- `'delay=<n>'`
Add a <n> ms delay after reset. This can be useful if a board takes a particularly long time to exit from external reset. <n> can be negative, in which case the default 120 ms delay after issuing reset will be shortened accordingly.
- `'strict'` Urclock has a faster, but slightly different strategy than `-c arduino` to synchronise with the bootloader; some `stk500v1` bootloaders cannot cope with this, and they need the `-xstrict` option.
- `'help'` Show this help menu and exit

BusPirate

The BusPirate programmer type accepts the following extended parameters:

- `'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 SDI, SDO 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.
- `'spifreq=0..7'`
0 30 kHz (default)

- 1 125 kHz
- 2 250 kHz
- 3 1 MHz
- 4 2 MHz
- 5 2.6 MHz
- 6 4 MHz
- 7 8 MHz

`'rawfreq=0..3'`

Sets the SPI speed and uses the Bus Pirate's binary "raw-wire" mode instead of the default binary SPI mode:

- 0 5 kHz
- 1 50 kHz
- 2 100 kHz
(Firmware v4.2+ only)
- 3 400 kHz (v4.2+)

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

`'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 `'reset='`, `'spifreq='` and `'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.

`'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.

`'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.

`'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.

```
'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:

```
'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.

Teensy bootloader

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

```
'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:

```
'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:

```
(AVR) (PICkit2)
RST  VPP/MCLR (1)
VDD  VDD Target (2)
      -- possibly
      optional if AVR
      self powered
GND  GND (3)
SDI  PGD (4)
SCLK PDC (5)
OSI  AUX (6)
```

Extended command line parameters:

`'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.

`'timeout=usb-transaction-timeout'`

Sets the timeout for USB reads and writes in milliseconds (default is 1500 ms).

`USBasp` Extended parameters:

`'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:

`'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.

`serialupdi`

Extended parameters:

`'rtsdtr=low|high'`

Forces `RTS/DTR` lines to assume low or high state during the whole programming session. Some programmers might use this signal to indicate UPDI programming state, but this is strictly hardware specific.

When not provided, driver/OS default value will be used.

`linuxspi` Extended parameter:

`'disable_no_cs'`

Ensures the programmer does not use the `SPI_NO_CS` bit for the SPI driver. This parameter is useful for kernels that do not support the `CS` line being managed outside the application.

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:

```
% 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"
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 done. Thank you.

%
```

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`:

```
% 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 done. Thank you.

%
```

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

```
% avrdude -p m128 -u -U flash:w:diag.hex \  
> -U eeprom:w:eeeprom.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.  
  
%
```

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

```
% avrdude -c jtag2 -p m649 -P usb:1c:37 -t
avrdude: AVR device initialized and ready to accept instructions

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

avrdude: Device signature = 0x1e9603

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

avrdude done. Thank you.
```

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.

```
% 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"
```

3 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 for all programmers:

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

`dump memtype addr ...`
Start reading from *addr*, all the way to the last memory address.

`dump memtype addr`
Read 256 bytes from the specified memory area, and display them.

`dump memtype ...`
Read all bytes from the specified memory, and display them.

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

`read` Can be used as an alias for dump.

`write memtype addr data[,] {data[,]}`
Manually program the respective memory cells, starting at address *addr*, using the data items provided. The terminal implements reading from and writing to flash and EEPROM type memories normally through a cache and paged access functions. All other memories are directly written to without use of a cache. Some older parts without paged access will also have flash and EEPROM directly accessed without cache.

Items *data* can have the following formats:

Type	Example	Size (bytes)
String	"Hello, world\n"	varying
Character	'A'	1
Decimal integer	12345	1, 2, 4, or 8
Octal integer	012345	1, 2, 4, or 8

Hexadecimal integer	0x12345	1, 2, 4, or 8
Float	3.1415926	4
Double	3.141592653589793D	8

data can be hexadecimal, octal or decimal integers, floating point numbers or C-style strings and characters. For integers, an optional case-insensitive suffix specifies the data size as in the table below:

LL	8 bytes / 64 bits
L	4 bytes / 32 bits
H or S	2 bytes / 16 bits
HH	1 byte / 8 bits

Suffix D indicates a 64-bit double, F a 32-bit float, whilst a floating point number without suffix defaults to 32-bit float. Hexadecimal floating point notation is supported. An ambiguous trailing suffix, e.g., 0x1.8D, is read as no-suffix float where D is part of the mantissa; use a zero exponent 0x1.8p0D to clarify.

An optional U suffix makes integers unsigned. Ordinary 0x hex integers are always treated as unsigned. +0x or -0x hex numbers are treated as signed unless they have a U suffix. Unsigned integers cannot be larger than $2^{64}-1$. If *n* is an unsigned integer then -*n* is also a valid unsigned integer as in C. Signed integers must fall into the $[-2^{63}, 2^{63}-1]$ range or a correspondingly smaller range when a suffix specifies a smaller type.

Ordinary 0x hex integers with *n* hex digits (counting leading zeros) use the smallest size of one, two, four and eight bytes that can accommodate any *n*-digit hex integer. If an integer suffix specifies a size explicitly the corresponding number of least significant bytes are written, and a warning shown if the number does not fit into the desired representation. Otherwise, unsigned integers occupy the smallest of one, two, four or eight bytes needed. Signed numbers are allowed to fit into the smallest signed or smallest unsigned representation: For example, 255 is stored as one byte as 255U would fit in one byte, though as a signed number it would not fit into a one-byte interval [-128, 127]. The number -1 is stored in one byte whilst -1U needs eight bytes as it is the same as 0xFFFFffffffU.

One trailing comma at the end of data items is ignored to facilitate copy and paste of lists.

`write memtype addr length data[,] {data[,]}` ...

The ellipses form ... of write is similar to above, but *length* byte of the memory are written. For that purpose, after writing the initial items, the last *data* item is replicated as many times as needed.

flush Synchronise with the device all pending cached writes to EEPROM or flash. With some programmer and part combinations, flash (and sometimes EEPROM, too) looks like a NOR memory, ie, one can only write 0 bits, not 1 bits.

When this is detected, either page erase is deployed (e.g., with parts that have PDI/UPDI interfaces), or if that is not available, both EEPROM and flash caches are fully read in, a chip erase command is issued and both EEPROM and flash are written back to the device. Hence, it can take minutes to ensure that a single previously cleared bit is set and, therefore, this command should be used sparingly.

- abort** Normally, caches are only ever actually written to the device when using **flush**, at the end of the terminal session after typing **quit**, or after EOF on input is encountered. The **abort** command resets the cache discarding all previous writes to the flash and EEPROM cache.
- erase** Perform a chip erase and discard all pending writes to EEPROM and flash.
- sig** Display the device signature bytes.
- part** Display the current part settings and parameters. Includes chip specific information including all memory types supported by the device, read/write timing, etc.
- 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.
- quell** [*level*]
Change (when *level* is provided), or display the quell level. 1 is used to suppress progress reports. 2 or higher yields progressively quieter operations. The initial quell level is controlled by the number of **-q** 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 some programmers:

- pgerase** *memory addr*
Erase one page of the memory specified.
- 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.
- spi** Enter direct SPI mode. The *pgmled* pin acts as chip select. *Only supported on parallel bitbang programmers, and partially by USBtiny.* Chip Select must be externally held low for direct SPI when using USBtinyISP, and send must be a multiple of four bytes.
- pgm** Return to programming mode (from direct SPI mode).
- 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 programming oscillator to *freq* Hz. An optional trailing letter *M* multiplies by 1E6, a trailing letter *k* by 1E3.

fosc off Turn the programming 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 programming 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:

```

% avrdude -p m128 -c stk500 -t

avrdude: AVR device initialized and ready to accept instructions
avrdude: Device signature = 0x1e9702
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
Memory Type Paged  Size  Size #Pages MinW  MaxW     Polled
-----
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

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>

```

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:

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

avrdude: AVR device initialized and ready to accept instructions
avrdude: Device signature = 0x1e9702
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

avrdude>
```

```
% avrdude -c pkobn_updi -p avr128db48 -t

      Vtarget          : 4.71 V
      PDI/UPDI clock Xmega/megaAVR : 100 kHz

avrdude: AVR device initialized and ready to accept instructions

Reading | ##### | 100% 0.01s

avrdude: Device signature = 0x1e970c (probably avr128db48)
avrdude> write eeprom 0 1234567890 'A' 'V' 'R' 2.718282 "Hello World!"
>>> write eeprom 0 1234567890 'A' 'V' 'R' 2.718282 "Hello World!"
Warning: no size suffix specified for "1234567890". Writing 4 byte(s)
Info: Writing 24 bytes starting from address 0x00

avrdude> dump eeprom 0 32
>>> dump eeprom 0 32

0000 d2 02 96 49 41 56 52 55 f8 2d 40 48 65 6c 6c 6f |...IAVRU.-@Hello|
0010 20 57 6f 72 6c 64 21 00 ff ff ff ff ff ff ff ff | World!.....|

avrdude> q
```

The following example demonstrates the second form of the `write` command where the last data value provided is used to fill up the indicated memory range.

```
avrdude> write eeprom 0x00 0x20 'a' 'b' 'c' 0x11 0xcafe 0x55 ...
>>> write eeprom 0x00 0x20 'a' 'b' 'c' 0x11 0xcafe 0x55 ...

avrdude> dump eeprom 0 0x30
>>> dump eeprom 0 0x30
0000  61 62 63 11 fe ca 55 55  55 55 55 55 55 55 55 55 |abc...UUUUUUUUUU|
0010  55 55 55 55 55 55 55 55  55 55 55 55 55 55 55 55 |UUUUUUUUUUUUUUUU|
0020  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff |.....|
```

4 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, chances are that you can copy an existing programmer definition and, with only a few changes, make your programmer work.

AVRDUDE first looks for a system wide configuration file in a platform dependent location. On Unix, this is usually `/usr/local/etc/avrdude.conf`, whilst on Windows it is usually in the same location as the executable file. The full name of this file can be specified using the `-C` command line option. After parsing the system wide configuration file, AVRDUDE looks for a per-user configuration file to augment or override the system wide defaults. On Unix, the per-user file is `${XDG_CONFIG_HOME}/avrdude/avrdude.rc`, whereas if `${XDG_CONFIG_HOME}` is either not set or empty, `${HOME}/.config/` is used instead. If that does not exist `.avrduderc` within the user's home directory is used. 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>                                # optional parent
  id      = <id1> [, <id2> ... ] ;           # <idN> are quoted strings
  desc    = <description> ;                   # quoted string
  type    = <type>;                           # programmer type, quoted string
                                              # supported types can be listed by "-c ?type"
  prog_modes = PM_<i/f> { | PM_<i/f> }        # interfaces, e.g., PM_SPM|PM_PDI
  connection_type = parallel | serial | usb | spi
  baudrate = <num> ;                           # baudrate for avr910-programmer
  vcc      = <pin1> [, <pin2> ... ] ;           # pin number(s)
  buff     = <pin1> [, <pin2> ... ] ;           # pin number(s)
  reset    = <pin> ;                             # pin number
  sck      = <pin> ;                             # pin number
  sdo      = <pin> ;                             # pin number
  sdi      = <pin> ;                             # pin number
  errled   = <pin> ;                             # pin number
  rdyled   = <pin> ;                             # pin number
```

```

pgmled   = <pin> ;                # pin number
vfyld   = <pin> ;                # 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
hvupdi_support = <num> [, <num>, ... ] ; # UPDI HV Variants Support
;

```

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.

Known programming modes are

- PM_SPM: Bootloaders, self-programming with SPM opcodes or NVM Controllers
- PM_TPI: Tiny Programming Interface (t4, t5, t9, t10, t20, t40, t102, t104)
- PM_ISP: SPI programming for In-System Programming (almost all classic parts)
- PM_PDI: Program and Debug Interface (xmega parts)
- PM_UPDI: Unified Program and Debug Interface
- PM_HVSP: High Voltage Serial Programming (some classic parts)
- PM_HVPP: High Voltage Parallel Programming (most non-HVSP classic parts)
- PM_debugWIRE: Simpler alternative to JTAG (a subset of HVPP/HVSP parts)
- PM_JTAG: Joint Test Action Group standard (some classic parts)
- PM_JTAGmkI: Subset of PM_JTAG, older parts, Atmel ICE mkI
- PM_XMEGAJTAG: JTAG, some XMEGA parts
- PM_AVR32JTAG: JTAG for 32-bit AVRs
- PM_aWire: AVR32 parts

To invert a bit in the pin definitions, use = ~ <num>. To invert a pin list (all pins get inverted) use ~ (<num1> [, <num2> ...]).

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

The following programmer types are currently implemented:

arduino	Arduino programmer
avr910	Serial programmers using protocol described in application note AVR910
avrftdi	Interface to the MPSSE Engine of FTDI Chips using libftdi.
buspirate	Using the Bus Pirate's SPI interface for programming
buspirate_bb	Using the Bus Pirate's bitbang interface for programming
butterfly	Atmel Butterfly evaluation board; Atmel AppNotes AVR109, AVR911
butterfly_mk	Mikrokopter.de Butterfly
dragon_dw	Atmel AVR Dragon in debugWire mode

dragon_hvsp	Atmel AVR Dragon in HVSP mode
dragon_isp	Atmel AVR Dragon in ISP mode
dragon_jtag	Atmel AVR Dragon in JTAG mode
dragon_pdi	Atmel AVR Dragon in PDI mode
dragon_pp	Atmel AVR Dragon in PP mode
flip1	FLIP USB DFU protocol version 1 (doc7618)
flip2	FLIP USB DFU protocol version 2 (AVR4023)
ftdi_syncbb	FT245R/FT232R Synchronous BitBangMode Programmer
jtagmki	Atmel JTAG ICE mkI
jtagmkii	Atmel JTAG ICE mkII
jtagmkii_avr32	Atmel JTAG ICE mkII in AVR32 mode
jtagmkii_dw	Atmel JTAG ICE mkII in debugWire mode
jtagmkii_isp	Atmel JTAG ICE mkII in ISP mode
jtagmkii_pdi	Atmel JTAG ICE mkII in PDI mode
jtagmkii_updi	Atmel JTAG ICE mkII in UPDI mode
jtagice3	Atmel JTAGICE3
jtagice3_pdi	Atmel JTAGICE3 in PDI mode
jtagice3_updi	Atmel JTAGICE3 in UPDI mode
jtagice3_dw	Atmel JTAGICE3 in debugWire mode
jtagice3_isp	Atmel JTAGICE3 in ISP mode
jtagice3_tpi	Atmel JTAGICE3 in TPI mode
linuxgpio	GPIO bitbanging using the Linux sysfs interface (not available)
linuxspi	SPI using Linux spidev driver (not available)
micronucleus	Micronucleus Bootloader
par	Parallel port bitbanging
pickit2	Microchip's PICkit2 Programmer
serbb	Serial port bitbanging
serialupdi	Driver for SerialUPDI programmers
stk500	Atmel STK500 Version 1.x firmware
stk500generic	Atmel STK500, autodetect firmware version
stk500v2	Atmel STK500 Version 2.x firmware
stk500hvsp	Atmel STK500 V2 in high-voltage serial programming mode
stk500pp	Atmel STK500 V2 in parallel programming mode
stk600	Atmel STK600
stk600hvsp	Atmel STK600 in high-voltage serial programming mode
stk600pp	Atmel STK600 in parallel programming mode
teensy	Teensy Bootloader
urclock	Urclock programmer for urboot bootloaders (arduino compatible)
usbasp	USBasp programmer, see http://www.fischl.de/usbasp/
usbtiny	Driver for "usbtiny"-type programmers

wiring <http://wiring.org.co/>, Basically STK500v2 protocol,
with some glue to trigger the bootloader.

xbee XBee Series 2 Over-The-Air (XBeeBoot)

4.3 Part Definitions

```

part
  desc          = <description> ;          # quoted string
  id            = <id> ;                    # quoted string
  family_id    = <id> ;                    # quoted string, e.g., "megaAVR" or "tinyAVR"
  prog_modes   = PM_<i/f> { | PM_<i/f>}    # interfaces, e.g., PM_SPM|PM_ISP|PM_HVPP|PM_debugWIRE
  mcuid        = <num>;                    # unique id in 0..2039 for 8-bit AVRs
  n_interrupts = <num>;                    # number of interrupts, used for vector bootloaders
  n_page_erase = <num>;                    # if set, number of pages erased during SPM erase
  n_boot_sections = <num>;                 # Number of boot sections
  boot_section_size = <num>;               # Size of (smallest) boot section, if any
  hvupdi_variant = <num> ;                 # numeric -1 (n/a) or 0..2
  devicecode   = <num> ;                   # deprecated, use stk500_devcode
  stk500_devcode = <num> ;                 # numeric
  avr910_devcode = <num> ;                 # numeric
  has_jtag      = <yes/no> ;               # part has JTAG i/f (deprecated, use prog_modes)
  has_debugwire = <yes/no> ;               # part has debugWire i/f (deprecated, use prog_modes)
  has_pdi       = <yes/no> ;               # part has PDI i/f (deprecated, use prog_modes)
  has_updi      = <yes/no> ;               # part has UPDI i/f (deprecated, use prog_modes)
  has_tpi       = <yes/no> ;               # part has TPI i/f (deprecated, use prog_modes)
  is_avr32      = <yes/no> ;               # AVR32 part (deprecated, use prog_modes)
  is_at90s1200  = <yes/no> ;               # AT90S1200 part
  signature     = <num> <num> <num> ;     # signature bytes
  usbpid        = <num> ;                  # DFU USB PID
  chip_erase_delay = <num> ;               # micro-seconds
  reset         = dedicated | io ;
  retry_pulse    = reset | sck ;
  chip_erase_delay = <num> ;               # chip erase delay (us)
  # 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 ATDF files
  timeout       = <num> ;
  stabdelay     = <num> ;
  cmdexedelay   = <num> ;
  synchloops    = <num> ;
  bytedelay     = <num> ;
  pollvalue     = <num> ;
  pollindex     = <num> ;
  predelay      = <num> ;
  postdelay     = <num> ;
  pollmethod    = <num> ;
  hvspcmdexedelay = <num> ;
  # STK500v2 HV programming parameters, from ATDFs
  pp_controlstack = <num>, <num>, ... ;   # PP only
  hvsp_controlstack = <num>, <num>, ... ; # HVSP only
  flash_instr    = <num>, <num>, <num> ;
  eeprom_instr   = <num>, <num>, ... ;
  hventerstabdelay = <num> ;
  progmodedelay = <num> ;                 # PP only
  latchcycles    = <num> ;

```



```

togglevtg      = <num> ;
poweroffdelay  = <num> ;
resetdelays    = <num> ;
resetdelayus   = <num> ;
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> ;
# debugWIRE and/or JTAG ICE mkII parameters, also from ATDF 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 != 0x3f
eind          = <num> ;                # mem addr of EIND reg
mcu_base      = <num> ;
nvm_base      = <num> ;
ocd_base      = <num> ;
ocdrev        = <num> ;
pgm_enable    = <instruction format> ;
chip_erase    = <instruction format> ;
# parameters for bootloaders
autobaud_sync = <num> ;                # autobaud detection byte, default 0x30

memory <memtype>
  paged       = <yes/no> ;              # yes/no (flash only, do not use for EEPROM)
  offset      = <num> ;                # memory offset
  size        = <num> ;                # bytes
  page_size   = <num> ;                # bytes
  num_pages   = <num> ;                # numeric
  n_word_writes = <num> ;              # TPI only: if set, number of words to write
  min_write_delay = <num> ;            # micro-seconds
  max_write_delay = <num> ;            # micro-seconds
  readback    = <num> <num> ;          # pair of byte values
  readback_p1 = <num> ;                # byte value (first component)
  readback_p2 = <num> ;                # byte value (second component)
  pwroff_after_write = <yes/no> ;     # yes/no
  mode        = <num> ;                # STK500 v2 file parameter from ATDF files
  delay       = <num> ;                # "
  blocksize   = <num> ;                # "
  readsize    = <num> ;                # "
  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> ;
;

```

;

If any of the above parameters are not specified, the default value of 0 is used for numerics (except for `mcuid`, `hvupdi_variant` and `ocdrev`, where the default value is -1, and for `autobaud_sync` which defaults to 0x30) or the empty string "" for string values. If a required parameter is left empty, AVRDUDE will complain. Almost all occurrences of numbers (with the exception of pin numbers and where they are separated by space, e.g., in signature and readback) can also be given as simple expressions involving arithmetic and bitwise operators.

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. If, however, a new memory definition refers to an existing one of the same name for that part then, from v7.1, the existing memory definition is extended, and components overwritten with new values. Assigning NULL removes an inherited SPI instruction format, memory definition, control stack, eeprom or flash instruction, e.g., as in `memory "efuse" = NULL`;

Example format for part inheritance:

```
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
- aN the bit is the Nth 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";
```

As the address bit numbers in the SPI opcodes are highly systematic, they don't really need to be specified. A compact version of the format specification neither uses bit-numbers for address lines nor spaces. If such a string is longer than 7 characters, then the characters 0, 1, x, a, i and o will be recognised as the corresponding bit, whilst any of the characters ., -, _ or / can act as arbitrary visual separators, which are ignored. Examples:

```
loadpage_lo = "0100.0000--000x.xxxx--xxaa.aaaa--iiii.iiii";

loadpage_lo = "0100.0000", "000x.xxxx", "xxaa.aaaa", "iiii.iiii";
```

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.

5 Programmer Specific Information

5.1 Atmel STK600

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

Routing card	Socket card	Devices
STK600-RC008T-2	STK600-ATTINY10 STK600-DIP	ATtiny4 ATtiny5 ATtiny9 ATtiny10 ATtiny11 ATtiny12 ATtiny13 ATtiny13A ATtiny25 ATtiny45 ATtiny85
STK600-RC008T-7	STK600-DIP	ATtiny15
STK600-RC014T-42	STK600-SOIC	ATtiny20
STK600-RC020T-1	STK600-DIP STK600-TinyX3U	ATtiny2313 ATtiny2313A ATtiny4313 ATtiny43U
STK600-RC014T-12	STK600-DIP	ATtiny24 ATtiny44 ATtiny84 ATtiny24A ATtiny44A
STK600-RC020T-8	STK600-DIP	ATtiny26 ATtiny261 ATtiny261A AT- tiny461 ATtiny861 ATtiny861A
STK600-RC020T-43	STK600-SOIC	ATtiny261 ATtiny261A ATtiny461 AT- tiny461A ATtiny861 ATtiny861A
STK600-RC020T-23	STK600-SOIC	ATtiny87 ATtiny167
STK600-RC028T-3	STK600-DIP	ATtiny28
STK600-RC028M-6	STK600-DIP	ATtiny48 ATtiny88 ATmega8 ATmega8A ATmega48 ATmega88 ATmega168 AT- mega48P ATmega48PA ATmega88P AT- mega88PA ATmega168P ATmega168PA ATmega328P
	QT600-ATTINY88- QT8	ATtiny88
STK600-RC040M-4	STK600-DIP	ATmega8515 ATmega162
STK600-RC044M-30	STK600-TQFP44	ATmega8515 ATmega162
STK600-RC040M-5	STK600-DIP	ATmega8535 ATmega16 ATmega16A AT- mega32 ATmega32A ATmega164P AT- mega164PA ATmega324P ATmega324PA ATmega644 ATmega644P ATmega644PA ATmega1284P
STK600-RC044M-31	STK600-TQFP44	ATmega8535 ATmega16 ATmega16A AT- mega32 ATmega32A ATmega164P AT- mega164PA ATmega324P ATmega324PA ATmega644 ATmega644P ATmega644PA ATmega1284P
	QT600-ATMEGA324- QM64	ATmega324PA

STK600-RC032M-29	STK600-TQFP32	ATmega8 ATmega8A ATmega48 ATmega88 ATmega168 ATmega48P ATmega48PA ATmega88P ATmega88PA ATmega168P ATmega168PA ATmega328P
STK600-RC064M-9	STK600-TQFP64	ATmega64 ATmega64A ATmega128 ATmega128A ATmega1281 ATmega2561 AT90CAN32 AT90CAN64 AT90CAN128
STK600-RC064M-10	STK600-TQFP64	ATmega165 ATmega165P ATmega169 AT- mega169P ATmega169PA ATmega325 AT- mega325P ATmega329 ATmega329P AT- mega645 ATmega649 ATmega649P
STK600-RC100M-11	STK600-TQFP100	ATmega640 ATmega1280 ATmega2560
	STK600-ATMEGA2560	ATmega2560
STK600-RC100M-18	STK600-TQFP100	ATmega3250 ATmega3250P ATmega3290 ATmega3290P ATmega6450 ATmega6490
STK600-RC032U-20	STK600-TQFP32	AT90USB82 AT90USB162 ATmega8U2 ATmega16U2 ATmega32U2
STK600-RC044U-25	STK600-TQFP44	ATmega16U4 ATmega32U4
STK600-RC064U-17	STK600-TQFP64	ATmega32U6 AT90USB646 AT90USB1286 AT90USB647 AT90USB1287
STK600-RCPWM-22	STK600-TQFP32	ATmega32C1 ATmega64C1 ATmega16M1 ATmega32M1 ATmega64M1
STK600-RCPWM-19	STK600-SOIC	AT90PWM2 AT90PWM3 AT90PWM2B AT90PWM3B AT90PWM216 AT90PWM316
STK600-RCPWM-26	STK600-SOIC	AT90PWM81
STK600-RC044M-24	STK600-TSSOP44	ATmega16HVB ATmega32HVB
	STK600-HVE2	ATmega64HVE
	STK600-ATMEGA128RFA1	ATmega128RFA1
STK600-RC100X-13	STK600-TQFP100	ATxmega64A1 ATxmega128A1 ATxmega128A1_revD ATxmega128A1U
	STK600-ATXMEGA1281A1	ATxmega128A1
	QT600-ATXMEGA128A1-ATxmega128A1	ATxmega128A1
	QT16	ATxmega128A1
STK600-RC064X-14	STK600-TQFP64	ATxmega64A3 ATxmega128A3 ATxmega256A3 ATxmega64D3 ATxmega128D3 ATxmega192D3 ATxmega256D3
STK600-RC064X-14	STK600-MLF64	ATxmega256A3B
STK600-RC044X-15	STK600-TQFP44	ATxmega32A4 ATxmega16A4 ATxmega16D4 ATxmega32D4
	STK600-ATXMEGAT0	ATxmega32T0
	STK600-uC3-144	AT32UC3A0512 AT32UC3A0256 AT32UC3A0128

STK600-RCUC3A144-33	STK600-TQFP144	AT32UC3A0512 AT32UC3A0128	AT32UC3A0256
STK600-RCuC3A100-28	STK600-TQFP100	AT32UC3A1512 AT32UC3A1128	AT32UC3A1256
STK600-RCuC3B0-21	STK600-TQFP64-2	AT32UC3B0256 AT32UC3B0512 AT32UC3B064	AT32UC3B0512RevC AT32UC3B0128 AT32UC3D1128
STK600-RCuC3B48-27	STK600-TQFP48	AT32UC3B1256	AT32UC3B164
STK600-RCUC3A144-32	STK600-TQFP144	AT32UC3A3512 AT32UC3A3128 AT32UC3A3256S AT32UC3A364S	AT32UC3A3256 AT32UC3A364 AT32UC3A3128S
STK600-RCUC3C0-36	STK600-TQFP144	AT32UC3C0512 AT32UC3C0128	AT32UC3C0256 AT32UC3C064
STK600-RCUC3C1-38	STK600-TQFP100	AT32UC3C1512 AT32UC3C1128	AT32UC3C1256 AT32UC3C164
STK600-RCUC3C2-40	STK600-TQFP64-2	AT32UC3C2512 AT32UC3C2128	AT32UC3C2256 AT32UC3C264
STK600-RCUC3C0-37	STK600-TQFP144	AT32UC3C0512 AT32UC3C0128	AT32UC3C0256 AT32UC3C064
STK600-RCUC3C1-39	STK600-TQFP100	AT32UC3C1512 AT32UC3C1128	AT32UC3C1256 AT32UC3C164
STK600-RCUC3C2-41	STK600-TQFP64-2	AT32UC3C2512 AT32UC3C2128	AT32UC3C2256 AT32UC3C264
STK600-RCUC3L0-34	STK600-TQFP48	AT32UC3L064 AT32UC3L016	AT32UC3L032
	QT600-AT32UC3L-QM64	AT32UC3L064	

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 AVR's 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 AREFO 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 Chapter 3 [Terminal Mode Operation], page 34.

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.

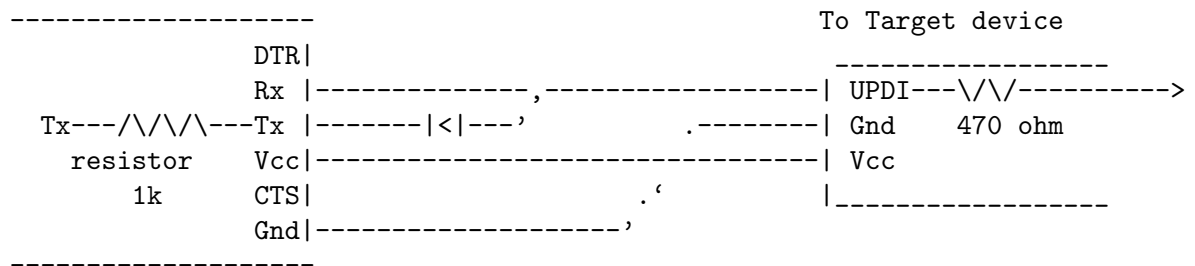
Sometimes, a bootloader with security mode already disabled seems to no longer respond with sensible configuration data, but only 0xFF for all queries. As these queries are used to obtain the equivalent of a signature, AVRDUDE can only continue in that situation by forcing the signature check to be overridden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



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.

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;
    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 A Platform Dependent Information

A.1 Unix

A.1.1 Unix Installation

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

```
$ gunzip -c avrdude-7.1.tar.gz | tar xf -
$ cd avrdude-7.1
$ ./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-7.1.tar.gz | tar xf -
$ cd avrdude-7.1
$ ./configure --prefix=$HOME/local
$ make
$ make install
```

A.1.1.1 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.

A.1.1.2 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-7.1.tar.gz
# rpm -Uvh /usr/src/redhat/RPMS/i386/avrdude-7.1-1.i386.rpm
```

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

A.1.2 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.

A.1.2.1 FreeBSD Configuration Files

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

A.1.2.2 Linux Configuration Files

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

A.1.3 Unix Port Names

The parallel and serial port device file names are system specific. MacOS has no default serial or parallel port names, but available ports can be found under `/dev/cu.*`. The following table lists the default names for a given system.

System	Default Parallel Port	Default Serial Port
FreeBSD	<code>/dev/ppi0</code>	<code>/dev/cuad0</code>
Linux	<code>/dev/parport0</code>	<code>/dev/ttyS0</code>
Solaris	<code>/dev/printers/0</code>	<code>/dev/term/a</code>

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.

A.1.4 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`.

A.2 Windows

A.2.1 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-7.1.tar.gz | tar xf -
$ cd avrdude-7.1
$ ./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.

A.2.2 Configuration Files

A.2.2.1 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`.

A.2.2.2 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:

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

A.2.3 Port Names

A.2.3.1 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.

A.2.3.2 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 that 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`).

A.2.4 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 B Troubleshooting

In general, please report any bugs encountered via <https://github.com/avrdudes/avrdude/issues>.

- 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:

JTAG pin	ISP pin
1	3
2	6
3	1
4	2
6	5
9	4

- 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:

JTAGICE mkII probe	Target pins	Squid cab- le colors	PDI header
1 (TCK)		Black	
2 (GND)	GND	White	6
3 (TDO)		Grey	
4 (VTref)	VTref	Purple	2
5 (TMS)		Blue	
6 (nSRST)	PDI_CLK	Green	5
7 (N.C.)		Yellow	
8 (nTRST)		Orange	
9 (TDI)	PDI_DATA	Red	1
10 (GND)		Brown	

- 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:

Dragon ISP Header	Target pins
1 (SDI)	PDI_DATA
2 (VCC)	VCC
3 (SCK)	
4 (SDO)	
5 (RESET)	PDI_CLK / RST
6 (GND)	GND

- 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:

AVRISP connector	Target pins	ATtiny pin #
1 (SDI)	TPIDATA	1
2 (VTref)	Vcc	5
3 (SCK)	TPICLK	3
4 (SDO)		
5 (RESET)	/RESET	6
6 (GND)	GND	2

- 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 *SDI* and *SDO* pins should be connected to the *TPIDATA* pin on the ATtiny device. However, a 1K resistor should be placed between the *SDO* and *TPIDATA*. The *SDI* 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 sdo=txd sdi=cts";
  type    = serbb;
  reset   = ~7;
  sck     = ~4;
  sdo     = ~3;
  sdi     = ~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 $V_{cc} = 5$ 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 $\geq 1.0.0 < 1.0.3$ 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. Or a programmer cannot initialize communication with a brand new chip.

Solution: Even though ISP starts with pulling `/RESET` low, the target continues to run at the internal clock speed either as defined by the firmware running before or as set by the factory. 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.

Concept Index

—

-x Arduino	23
-x AVR Dragon	23
-x AVR910	23
-x Buspirate	26
-x linuxspi	29
-x Micronucleus bootloader	28
-x PICkit2	28
-x serialupdi	29
-x Teensy bootloader	28
-x Urlock	23
-x USBasp	29
-x Wiring	28
-x xbee	29

A

avrdude.conf	40
--------------------	----

C

Configuration File	40
--------------------------	----

D

Device support	5
DFU bootloader	50

H

History	4
---------------	---

I

Introduction	1
--------------------	---

O

Options (command-line)	5
------------------------------	---

P

Programmer support	13
Programmings supported	1

S

SerialUPDI	51
STK600	48

T

Terminal Mode	34, 37
---------------------	--------