AVRDUDE

A program for downloading/uploading AVR microcontroller flash, EEPROM and more
for AVRDUDE, Version 7.3, 07 February 2024

by Hans Eirik Bull, Brian S. Dean, Stefan Rüger and Jörg Wunsch

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

Table of Contents

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 via the command line to read or write chip memories (eeprom, flash, fuses, lock bits) and read memories such as signature or calibration bytes; the same can be achieved 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 `https://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 occuring and thus it cannot be used as the reset pin. A readily available level translator should be used between the SPI bus/reset GPIO and the chip to avoid potentially damaging the computer's SPI controller in the event that the chip is running at 5V and the SPI runs at 3.3V. The GPIO chosen for reset can be configured in the avrdude configuration file using the `reset` entry under the linuxspi programmer, or directly in the port specification. An external pull-up resistor should be connected between the AVR's reset pin and Vcc. If Vcc is not the same as the SPI voltage, this should be done on the AVR side of the level translator to protect the hardware from damage.

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

Typically, pins 19, 21, and 23 are SPI 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 0 (the lsb of the byte in the config file) is SCK. Bit 1 is SDO, and Bit 2 is SDI. Bit 3 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, USBtinyISP and CH341A adapters are also supported, provided AVRDUDE has been compiled with libusb support. They former two feature simple firmware-only USB implementations, running on an ATmega8 (or ATmega88), or ATtiny2313, respectively. CH341A programmers connect directly to the AVR target. Their SPI bit clock is approximately 1.7 MHz and cannot be changed. As a consequence, the AVR target must have a CPU frequency of 6.8 MHz or more: factory-set AVR parts, which typically run on an internal oscillator between 1 MHz and 1.6 MHz, cannot be programmed using `-c ch341a`.

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 57, for more details and known issues.

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

The Micronucleus bootloader is supported for both protocol version V1 and V2. As the bootloader does not support reading from flash memory, use the `-V` option to prevent AVRDUDE from 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, MacOS and Windows. The developers behind the porting effort primarily were Ted Roth, Eric Weddington, and Jörg 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 Jörg Wunsch, the original web site documentation by Brian Dean, and from the comments describing the fields in the AVRDUDE configuration file by Brian Dean. The texi formatting was modeled after that of the Simulavr documentation by Ted Roth.

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. To see a list of currently supported MCUs use ? as partno, which will print the part ids and official part names. In connection with -v, this will also print a list of variant part names followed by an optional colon, the package code and some absolute maximum ratings. The part id, their official part name, any of the full variant part names or their initial part up to a dash can be used to specify a part 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:

(*) 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 `baudrate` entry of the configuration file or defined by the `default_baudrate` entry in your `~/.config/avrdude/avrdude.rc` or `~/.avrduderc` configuration file if no `baudrate` entry was provided for this programmer.

`-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.

Note that some official Microchip programmers store the bitclock setting and will continue to use it until a different value is provided. This applies to "2nd gen" programmers (AVRISPmkII, AVR Dragon, JTAG ICE mkII, STK600) and "3rd gen"programmers (JTAGICE3, Atmel ICE, Power Debugger). "4th gen" programmers (PICkit 4, MPLAB SNAP) will store the last user-specified bitclock until the programmer is disconnected from the computer.

`-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:

`-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 Fedora Linux, AVRDUDE looks at `/etc/avrdude/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.

`-N` Do not load the personal configuration file that is usually located at `~/.config/avrdude/avrdude.rc`, `~/.avrduderc` or in the same directory as the avrdude executable.

`-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 writing data to the MCU whilst processing -U (useful for debugging AVRDUDE). The terminal mode continues to write to the device.

`-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 connection through which the programmer is attached. This can be a parallel, serial, spi or linuxgpio connection. The programmer normally specifies the connection type; in absence of a `-P` specification, system-dependent default values `default_parallel`, `default_serial`, `default_spi`, or `default_linuxgpio` from the configuration file are used. If you need to use a different port, use this option to specify the alternate port name.

If avrdude has been configured with libserialport support, a serial port can be specified using a predefined serial adapter type in *avrdude.conf* or *.avrduderc*, e.g., `ch340` or `ft232r`. If more than one serial adapter of the same type is connected, they can be distinguished by appending a serial number, e.g., `ft232r:12345678`. Note that the USB to serial chip has to have a serial number for this to work. Avrdude can check for leading and trailing serial number matches as well. In the above example, `ft232r:1234` would also result in a match, and so would `ft232r:...5678`. If the USB to serial chip is not known to avrdude, it can be specified using the hexadecimal USB vendor ID, hexadecimal product ID and an optional serial number, following the serial number matching rules described above, e.g., `usb:0x2341:0x0043` or `usb:2341:0043:12345678`. To see a list of currently plugged-in serial ports use `-P ?s`. In order to see a list of all possible serial adapters known to avrdude use `-P ?sa`.

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 26.

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 `https://www.obdev.at/products/vusb/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.

For USBasp, multiple devices can be distinguished by either USB connection or serial number. See the respective Appendix B [Troubleshooting], page 64, 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).

`-r`
: Opens the serial port at 1200 baud and immediately closes it, waits 400 ms for each `-r` on the command line and then establishes communication with the programmer. This is commonly known as a "1200bps touch", and is used to trigger programming mode for certain boards like Arduino Leonardo, Arduino Micro/Pro Micro and the Arduino Nano Every. Longer waits, and therefore multiple `-r` options, are sometimes needed for slower, less powerful hosts.

`-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` *cmd*
: Run terminal line *cmd* when it is its turn in relation to other `-t` interactive terminals, `-T` terminal commands and `-U` memory operations. Except for the simplest of terminal commands the argument *cmd* will most likely need to be set in quotes, see your OS shell manual for details. See below for a detailed description of all terminal commands.

`-t`
: Tells AVRDUDE to run an interactive terminal when it is its turn in relation to other `-t` interactive terminals, `-T` terminal commands and `-U` memory operations.

`-U` *memory*:*op*:*filename*[:*format*]

Perform a memory operation when it is its turn in relation to other `-t` interactive terminals, `-T` terminal commands and `-U` memory operations. The *memory* field specifies the memory type to operate on. Use the `-T part` option on the command line or the `part` command in the interactive terminal to display all the memories supported by a particular device.

Typically, a device's memory configuration at least contains the memory types `flash`, `eeprom`, `signature` and `lock`, which is sometimes known as `lockbits`. The signature memory contains the three device signature bytes, which should be, but not always are, unique for the part. The `lock` memory of one or four bytes typically details whether or not external reading/writing of the flash memory, or parts of it, is allowed. Parts will also typically have fuse bytes, which are read/write memories for configuration of the device and calibration memories that typically contain read-only factory calibration values.

Classic devices may have the following memories in addition to `eeprom`, `flash`, `signature` and `lock`:

`calibration`
: One or more bytes of RC oscillator calibration data

`efuse`
: Extended fuse byte

`fuse`
: Fuse byte in devices that have only a single fuse byte

`hfuse`
: High fuse byte

`lfuse`
: Low fuse byte

`prodsig`
: Signature, calibration byte and serial number in a small read-only memory, which is only documented to be available for ATmega324PB, ATmega328PB, ATtiny102 and ATtiny104; programmers may or may not be able to read this memory

`sigrow`
: Memory alias for prodsig

`usersig`
: Three extra flash pages for firmware settings; this memory is not erased during a chip erase. Only some classic parts, ATmega(64|128|256|644|1284|2564)RFR2, have a usersig memory. Usersig is different to flash in the sense that it can neither be accessed with ISP serial programming nor written to by bootloaders. AVRDUDE offers JTAG programming of classic-part usersig memories. As with all flash-type memories the `-U` option can only write 0-bits but not 1-bits. Hence, usersig needs to be erased before a file can be uploaded to this memory region, e.g., using `-T "erase usersig" -U usersig:w:parameters.hex:i`

`io`
: Volatile register memory; it cannot be accessed by external programming methods only by bootloaders, which has limited use unless the bootloader jumps to the application directly, i.e., without a WDT reset

`sram`
: Volatile RAM memory; like `io` it cannot be accessed by external programming

ATxmega devices have the following memories in addition to `eeprom`, `flash`, `signature` and `lock`:

`application`
: Application flash area

`apptable`
: Application table flash area

`boot`
: Boot flash area

`fuse0`
: A.k.a. jtaguid: JTAG user ID for some devices

`fuse1`
: Watchdog configuration

`fuse6`
: Fault detection action configuration TC4/5 for ATxmega E series parts

`fuse`*N*
: Other fuse bytes of ATxmega devices, where *N* is 2, 4 or 5, for system configuration

`prodsig`
: The production signature row is a read-only memory section for factory programmed data such as the signature and calibration values for oscillators or analogue modules; it also contains a serial number that consists of the production lot number, wafer number and wafer coordinates for the part

`sigrow`
: Memory alias for prodsig

`usersig`
: Additional flash memory page that can be used for firmware settings; this memory is not erased during a chip erase

`io`
: Volatile register memory; AVRDUDE can read this memory but not write to it using external programming

`sram`
: Volatile RAM memory; cannot be usefully accessed by external programming

Modern 8-bit AVR devices have the following memories in addition to `eeprom`, `flash`, `signature` and `lock`:

`fuse0`
: A.k.a. wdtcfg: watchdog configuration

`fuse1`
: A.k.a. bodcfg: brownout detection configuration

`fuse2`
: A.k.a. osccfg: oscillator configuration

`fuse4`
: A.k.a. tcd0cfg (not all devices): timer counter type D configuration

`fuse5`
: A.k.a. syscfg0: system configuration 0

`fuse6`
: A.k.a. syscfg1: system configuration 1

`fuse7`
: A.k.a. append or codesize: either the end of the application code section or the code size in blocks of 256/512 bytes

`fuse8`
: A.k.a. bootend or bootsize: end of the boot section or the boot size in blocks of 256/512 bytes

`fusea`
: A.k.a. pdicfg: configures/locks updi access; it is the only fuse that consists of two bytes

`fuses` A "logical" memory of up to 16 bytes containing all fuseX of a part, which can be used to program all fuses at the same time

`osc16err` Two bytes typically describing the 16 MHz oscillator frequency error at 3 V and 5 V, respectively

`osc20err` Two bytes typically describing the 20 MHz oscillator frequency error at 3 V and 5 V, respectively

`osccal16` Two oscillator calibration bytes for 16 MHz

`osccal20` Two oscillator calibration bytes for 20 MHz

`prodsig` Read-only memory section for factory programmed data such as the signature, calibration values and serial number

`sigrow` Memory alias for prodsig

`sernum` Serial number with a unique ID for the part (10 or 16 bytes)

`tempsense`
Temperature sensor calibration values

`bootrow` Extra page of memory that is only accessible by the MCU in bootloader code; UDPI can read and write this memory only when the device is unlocked; bootrow is not erased during chip erase

`userrow` Extra page of EEPROM memory that can be used for firmware settings; this memory is not erased during a chip erase

`sib` Special system information block memory with information about AVR family, chip revision etc.

`io` Volatile register memory; AVRDUDE can program this memory but this is of limited utility because anything written to the io memory will be undefined or lost after reset; writing to individual registers in the terminal can still be used, e.g., to test I/O ports

`sram` Volatile RAM memory; can be read and written but contents will be lost after reset

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 are specified on the command line, separated by commas or spaces in place of the *filename* field of the `-U` option. This is useful for programming fuse bytes without having to create a single-byte file or enter terminal mode.

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

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

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

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

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

When used as input, the `m`, `d`, `h`, `o` and `b` formats will use the same code for reading lists of numbers separated by white space and/or commas. The read routine handles decimal, hexadecimal, octal or binary numbers on a number-by-number basis, and the list of numbers can therefore be of mixed type. In fact the syntax, is the same as for data used by the terminal write command, i.e., the file's input data can also be 2-byte short integers, 4-byte long integers or 8-byte long long integers, 4-byte floating point numbers, 8-byte double precision numbers, C-type strings with a terminating nul or C-like characters such as `'\t'`. Numbers are written as little endian to memory. When using `0x` hexadecimal or `0b` binary input leading zeros are used to determine the size of the integer, e.g., `0x002a` will occupy two bytes and write a `0x2a` to memory followed by `0x00`, while `0x01234` will occupy 4 bytes. See the description of the terminal write command for more details.

In absence of an explicit file format, the default is to use auto detection for input files, and raw binary format for output files. Note that if *filename* contains a colon as penultimate character the *format* field is no longer optional since the last character 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. This behaviour can be overridden with the `-A` option.

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 pair of colons in it that sandwich a single character as otherwise the first part might be interpreted as memory, and the single character as memory operation.

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

`-V` Disable automatic verify check when uploading data with `-U`.

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

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.

‘`hvupdi`’ *Power Debugger and Pickit 4 only*
High-voltage UPDI programming is used to enable a UPDI pin that has previously been set to RESET or GPIO mode. Use ‘`-xhvupdi`’ to enable high-voltage UPDI initialization for supported targets.

‘`vtarg=VALUE, vtarg`’
Power Debugger only
The voltage generator can be enabled by setting a target voltage. The current set-voltage can be read by ‘`-xvtarg`’ alone.

‘`help`’ Show help menu and exit.

`PICkit 4`

`MPLAB SNAP`

The PICkit 4 and MPLAB SNAP programmers accept the following extended parameters:

‘`mode=avr,pic`’
Switch programmer to AVR or PIC mode, then exit: the PICkit 4 and MPLAB SNAP programmer can only be utilised by Avrdude when in AVR mode. Use ‘`-xmode=avr`’ for switching to AVR mode, or ‘`-xmode=pic`’ for switching to PIC mode.

‘`help`’ Show help menu and exit.

`Xplained Mini`

The Xplained Mini/Nano programmer (ISP or UPDI, not TPI) type accepts the following extended parameters:

‘`suffer=VALUE`’, ‘`suffer`’
The SUFFER register allows the user to modify the behavior of the on-board mEDBG. The current state can be read by ‘`-xsuffer`’ alone.

`Bit 7 ARDUINO:`
Adds control of extra LEDs when set to 0

`Bit 6..3:` Reserved (must be set to 1)

`Bit 2 EOF:`
Agressive power-down, sleep after 5 seconds if no USB enumeration when set to 0

`Bit 1 LOWP:`
forc running the mEDBG at 1 MHz when bit set to 0

`Bit 0 FUSE:`
Fuses are safe-masked when bit sent to 1. Fuses are unprotected when set to 0

'`vtarg_switch=VALUE`', '`vtarg_switch`'
The on-board target voltage switch can be turned on or off by writing a 1 or a 0. The current state can be read by '`-xvtarg_switch`' alone. Note that the target power switch will always be on after a power cycle. Also note that the smaller Xplained Nano boards does not have a target power switch.

'`help`' Show help menu and exit.

`Curiosity Nano`
The Curiosity Nano board accepts the following extended parameter:

'`vtarg=VALUE, vtarg`'
The generated on-board target voltage can be changed by specifying a new voltage. The current set-voltage can be read by '`-xvtarg`' alone.

'`help`' Show help menu and exit.

`STK500`

`STK600`

The STK500 and STK600 boards accept the following extended parameters:

'`vtarg=VALUE, vtarg`'
The generated on-board target voltage can be changed by specifying a new voltage. The current set-voltage can be read by '`-xvtarg`' alone.

'`fosc=VALUE[MHz|M|kHz|k|Hz|H], fosc`'
Set the programmable oscillator frequency in MHz, kHz or Hz. The current frequency can be read by '`-xfosc`' alone.

'`varef=VALUE, varef`'
The generated on-board analog reference voltage can be changed by specifying a new reference voltage. The current reference voltage can be read by '`-xvaref`' alone.

'`varef[0,1]=VALUE, varef[0,1]`'
STK600 only
The generated on-board analog reference voltage for channel 0 or

channel 1 can be changed by specifying a new reference voltage. The current reference voltage can be read by '-xvaref0' or '-xvaref1' alone.

'attemps[=<1..99>]'
: *STK500V1 only*
Specify how many connection retry attemps to perform before exiting. Defaults to 10 if not specified.

'xtal=VALUE[MHz|M|kHz|k|Hz|H]'
: Defines the XTAL frequency of the programmer if it differs from 7.3728 MHz of the original STK500. Used by avrdude for the correct calculation of fosc and sck.

'help' Show help menu and exit.

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.

'help' Show help menu and exit.

Arduino

The Arduino programmer type accepts the following extended parameter:

'attemps=VALUE'
: Overide the default number of connection retry attempt by using *VALUE*.

'help' Show help menu and exit.

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, i.e., 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.

'nostore'
: On writing to flash do not store metadata except the metadata code byte `0xff` saying there are no metadata. In particular, no data store frame is programmed.

'nometadata'
: Do not support any metadata. The full flash besides the bootloader is available for the application. If the application is smaller than the available space then a metadata code byte `0xff` is stored nevertheless to indicate there are no further metadata available. In absence of `-xnometadata`, the default for the urclock programmer is to write as much metadata (filename, data and store information) as the size of the uploaded application and the other extended options allow. The subtle difference between `-xnometadata` and `-xnostore` is that the latter always explicitly stores in flash that no further metadata are available, so that a such prepared flash can always be queried with `avrdude -xshowall`. In contrast to this, it cannot be guaranteed that a `-xshowall` query on flash prepared with `-xnometadata` yields useful results.

'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 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.

'`help`' Show help menu and exit.

`Micronucleus bootloader`

The Micronucleus programmer type accepts the following extended parameter:

'`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.

'`help`' Show help menu and exit.

`Teensy bootloader`

The Teensy programmer type accepts the following extended parameter:

'`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.

'`help`' Show help menu and exit.

`Wiring`

The Wiring programmer type accepts the following extended parameters:

'`snooze=<n>`'
: 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.

'`delay=<n>`'
: Add a `<n>` milliseconds 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 100 ms delay after issuing reset will be shortened accordingly.

`help` Show help menu and exit.

`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)

The PICkit2 programmer type accepts the following extended 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).

`help` Show help menu and exit.

`USBasp`

The USBasp programmer type accepts the following extended parameter:

`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).

`help` Show help menu and exit.

`xbee`

The xbee programmer type accepts the following extended parameter:

`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.

`help` Show help menu and exit.

`jtag2updi`
`serialupdi`

The jtag2updi and serialupdi programmer types accept the following 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.

'`help`' Show help menu and exit.

`linuxspi`

The linuxspi programmer type accepts the following 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.

'`help`' Show help menu and exit.

2.3 Example Command Line Invocations

AVRDUDE error messages, warnings and progress reports are generally written to stderr which can, in bash, be turned off by `2>/dev/null` or by using increasingly more `-q` options to suppress them. Terminal output of commands or that of the `-U` command with an output file named `-` are written to stdout. In some examples empty lines are shown for clarity that are not printed by AVRDUDE or the shell.

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
avrdude: device signature = 0x1e9702 (probably m128)
avrdude: erasing chip
avrdude: reading input file diag.hex for flash
         with 19278 bytes in 74 section within [0, 0x4b4e]
avrdude: writing 19278 bytes flash ...
Writing | ################################################## | 100% 7.60 s
avrdude: 19278 bytes of flash written
avrdude: verifying flash memory against diag.hex
Reading | ################################################## | 100% 6.83 s
avrdude: 19278 bytes of flash verified

avrdude done.  Thank you.
```

Same but in **quell-progress-reporting (silent) mode `-qq`:**

```
$ avrdude -qq -p m128 -c stk500 -e -U flash:w:diag.hex
```

Using `&&` to confirm that the silent AVRDUDE command went OK:

```
$ avrdude -qq -p m128 -c stk500 -e -U flash:w:diag.hex && echo OK
OK
```

Save flash memory in raw binary format to 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
avrdude: device signature = 0x1e9702 (probably m128)
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, `eeprom.hex` to EEPROM, and **set the extended, high, and low fuse bytes** to 0xff, 0x89, and 0x2e respectively:

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

avrdude: AVR device initialized and ready to accept instructions
avrdude: device signature = 0x1e9702 (probably m128)
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 for flash
         with 19278 bytes in 74 section within [0, 0x4b4e]
avrdude: writing 19278 bytes flash ...
Writing | ################################################## | 100% 7.60 s
avrdude: 19278 bytes of flash written
avrdude: verifying flash memory against diag.hex
Reading | ################################################## | 100% 6.83 s
avrdude: 19278 bytes of flash verified

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

avrdude done.  Thank you.
```

Read the fuses and print their values in different formats (hexadecimal, binary and octal):

```
$ avrdude -cusbasp -patmega128 -qq -Ulfuse:r:-:h -Uhfuse:r:-:b -Uefuse:r:-:o

0xbf
0b11000110
0377
```

Connect to the JTAG ICE mkII with a **serial number ending 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
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"
```

Write data from stdin (standard input) to EEPROM; no error output means all went fine:

```
$ echo 'The quick brown fox' | avrdude -c usbasp -p attiny13 -qq -U eeprom:w:-:r
```

Execute multiple terminal mode commands separated by semicolons:

```
$ echo 'write eeprom 0 "Bonjour"; write ee 0x18 0x12345678; dump eeprom 0 0x20' | \
  avrdude -qqcdryrun -patmega328p -t

0000  42 6f 6e 6a 6f 75 72 00  ff ff ff ff ff ff ff ff  |Bonjour.........|
0010  ff ff ff ff ff ff ff ff  78 56 34 12 ff ff ff ff  |........xV4.....|
```

Read EEPROM and write content to stdout (standard output):

```
$ avrdude -qq -cusbasp -pattiny13 -Ueeprom:r:-:i

:20000000E2809954686520717569636B2062726F776E20666F78E280990AFFFFFFFFFFFFD3
:20002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
:00000001FF
```

Same but redirect stderr (standard error output) to `/dev/null` instead of using `-qq`:

```
$ avrdude -cusbasp -pattiny13 -Ueeprom:r:-:i 2>/dev/null

:20000000E2809954686520717569636B2062726F776E20666F78E280990AFFFFFFFFFFFFD3
:20002000FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE0
:00000001FF
```

Using the Avrdude output to print strings present in flash memory:

```
$ avrdude -pattiny13 -qq -U flash:r:-:r | strings

Main menu
Distance: %d cm
Exit
```

Factory fuse setting of a device:

```
$ avrdude -patmega328p/St | grep initval

.ptmm ATmega328P lfuse initval 0x62
.ptmm ATmega328P hfuse initval 0xd9
.ptmm ATmega328P efuse initval 0xff
.ptmm ATmega328P lock initval 0xff
```

List of all parts known to AVRDUDE:

```
$ avrdude -p*/d | grep = | cut -f2 -d"'"

ATtiny11
ATtiny12
ATtiny13
ATtiny13A
ATtiny15
AT89S51
[...]
AVR64EA48
LGT8F88P
LGT8F168P
LGT8F328P
```

List of all modern AVR parts (with UPDI interface) known to AVRDUDE:

```
$ avrdude -p*/d | grep PM_UPDI | cut -f2 -d"'"

ATtiny202
ATtiny204
ATtiny402
[...]
AVR64EA28
AVR64EA32
AVR64EA48
```

List of all curently plugged-in serial devices known to the libserialport library:

```
$ avrdude -P ?s
Possible candidate serial ports are:
  -P /dev/ttyUSB0 or -P ft232r:A600K203
  -P /dev/ttyUSB1 or -P ft232r:MCU8
  -P /dev/ttyUSB3, -P ch340 or -P ch340-115k
Note that above ports might not be connected to a target board or an AVR programmer.
Also note there may be other direct serial ports not listed above.
```

List of all serial adapters known to AVRDUDE, i.e., defined in avrdude.conf:

```
$ avrdude -P ?sa
Valid serial adapters are:
  ch340      = [usbvid 0x1a86, usbpid 0x7523]
  ch340-115k = [usbvid 0x1a86, usbpid 0x7523]
  ch341a     = [usbvid 0x1a86, usbpid 0x5512]
  ch9102     = [usbvid 0x1a86, usbpid 0x55d4]
  cp210x     = [usbvid 0x10c4, usbpid 0xea60 0xea70 0xea71]
  ft2232h    = [usbvid 0x0403, usbpid 0x6010]
  ft231x     = [usbvid 0x0403, usbpid 0x6015]
  ft234x     = [usbvid 0x0403, usbpid 0x6015]
  ft230x     = [usbvid 0x0403, usbpid 0x6015]
  ft232h     = [usbvid 0x0403, usbpid 0x6014]
  ft232r     = [usbvid 0x0403, usbpid 0x6001]
  ft4232h    = [usbvid 0x0403, usbpid 0x6011]
  pl2303     = [usbvid 0x067b, usbpid 0x2303]
```

AVRDUDE in a bash script creating terminal scripts that reset a part to factory settings:

```
$ cat make-init-scripts

#!/bin/bash
mkdir /tmp/factory
for part in $(avrdude -p*/d | grep = | cut -f2 -d"'"); do
  echo $part
  avrdude -p$part/St | grep initval | cut -f3,5 | grep -ve-1 \
  | sed "s/.*/write &/" >/tmp/factory/$part.ini
done
```

Run above script and use one of the created terminal scripts:

```
$ ./make-init-scripts

$ cat /tmp/factory/ATmega328P.ini
write lfuse 0x62
write hfuse 0xd9
write efuse 0xff
write lock 0xff

$ avrdude -qq -cusbasp -pATmega328P -t < /tmp/factory/ATmega328P.ini
```

Output a list of non-bootloader programmers that can be used for a part. Note that |& folds stderr into stdout in a bash shell:

```
$ avrdude -c? -pavr32ea32 |& grep -v bootloader

Valid programmers for part AVR32EA32 are:
  atmelice_updi      = Atmel-ICE (ARM/AVR) via UPDI
  dryrun             = Emulates programming without a programmer via UPDI
  jtag2updi          = JTAGv2 to UPDI bridge via UPDI
  jtag3updi          = Atmel AVR JTAGICE3 via UPDI
  pickit4_updi       = MPLAB(R) PICkit 4 via UPDI
  pkobn_updi         = Curiosity nano (nEDBG) via UPDI
  powerdebugger_updi = Atmel PowerDebugger (ARM/AVR) via UPDI
  serialupdi         = SerialUPDI via UPDI
  snap_updi          = MPLAB(R) SNAP via UPDI
  xplainedmini_updi  = Atmel AVR XplainedMini via UPDI
  xplainedpro_updi   = Atmel AVR XplainedPro via UPDI
```

Print filename of last stored sketch with its date stamp (only with urclock programmer):

```
$avrdude -qq -curclock -P/dev/ttyUSB0 -pattiny13 -xshowdate -xshowfilename

2023-05-19 11.13 blink.hex
```

Create a bash function `avrdude-elf` that takes an elf file as input, with support for optional Avrdude flags at the end, and **writes to all memories specified in the elf file.** In this example, the elf file did not contain any EEPROM data:

```
# Show all writable memories present for the ATtiny13
$ echo $(avrdude -pattiny13/ot | grep write | cut -f3 | uniq)

eeprom flash lfuse hfuse lock

# Function that writes to all memories present in the elf file
avrdude-elf() {
  avrdude -cusbasp -pattiny13 -U{eeprom,flash,{l,h}fuse,lock}:w:"$1":e "${@:2}"
}

# Run function where -B8 and -V is appended to the Avrdude command
$ avrdude-elf program.elf -B8 -V

avrdude: set SCK frequency to 93750 Hz
avrdude: AVR device initialized and ready to accept instructions
avrdude: device signature = 0x1e9007 (probably t13)
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 Blink.elf for eeprom
         with 0 bytes in 0 sections within [0, -1]
         using 0 pages and 0 pad bytes
avrdude: writing 0 bytes eeprom ...
Writing | ################################################## | 100% 0.00 s
avrdude: 0 bytes of eeprom written

avrdude: reading input file Blink.elf for flash
         with 78 bytes in 1 section within [0, 0x4d]
         using 3 pages and 18 pad bytes
avrdude: writing 78 bytes flash ...
Writing | ################################################## | 100% 0.09 s
avrdude: 78 bytes of flash written

avrdude: reading input file Blink.elf for lfuse
         with 1 byte in 1 section within [0, 0]
avrdude: writing 1 byte lfuse ...
avrdude: 1 byte of lfuse written

avrdude: reading input file Blink.elf for hfuse
         with 1 byte in 1 section within [0, 0]
avrdude: writing 1 byte hfuse ...
avrdude: 1 byte of hfuse written

avrdude: reading input file Blink.elf for lock
         with 1 byte in 1 section within [0, 0]
avrdude: writing 1 byte lock ...
avrdude: 1 byte of lock written

avrdude done.  Thank you.
```

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

In this mode, AVRDUDE only initializes communication with the MCU, and then awaits user commands on standard input. Commands and parameters may be abbreviated to the shortest unambiguous form. Terminal mode provides a command history using readline(3), so previously entered command lines can be recalled and edited.

The *addr* and *len* parameters of the dump, read, write, save and erase commands can be negative with the same syntax as substring computations in perl or python. The table below details their meaning with respect to an example memory of size `sz=0x800`.

`addr`	`len`	**Memory interval**	**Comment**
0/positive	positive	`[addr, addr+len-1]`	Note: `len = end-start + 1`
0/positive	negative	`[addr, sz+len]`	End is `|len|` bytes below memory size `sz`
negative	positive	`[sz+addr, ...]`	Start is `|addr|` bytes below memory size
negative	negative	`[sz+addr, sz+len]`	Combining above two cases
any	zero	empty set	No action
`0x700`	`12`	`[0x700, 0x70b]`	Conventional use
`1024`	`-257`	`[0x400, 0x6ff]`	Size of memory is `2048` or `0x800`
`-512`	`512`	`[0x600, 0x7ff]`	Last 512 bytes
`-256`	`-1`	`[0x700, 0x7ff]`	Last 256 bytes
`0`	`49`	`[0, 48]`	First 49 bytes
`0`	`-49`	`[0, 1999]`	All but the last `48 = |len+1|` bytes
`0`	`-1`	`[0, 0x7ff]`	All memory without knowing its size

The following commands are implemented for all programmers:

`dump` *memory addr len*
: Read from the specified memory interval (see above), and display in the usual hexadecimal and ASCII form.

`dump` *memory addr*
: Read from memory addr as many bytes as the most recent dump memory addr len command with this very memory had specified (default 256 bytes), and display them.

`dump` *memory*
: Continue dumping from the memory and location where the most recent dump command left off; if no previous dump command has addressed a memory an error message will be shown.

`dump` *memory addr* `...`
: Start reading from *addr*, all the way to the last memory address (deprecated: use `dump` *memory addr* `-1`).

`dump` *memory* `...`
: Read all bytes from the specified memory, and display them (deprecated: use `dump` *memory* `0 -1`).

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

`write` *memory 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, EEPROM, bootrow and usersig 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, depending on the programmer, might 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
File	`C:/My\ projects/blink.hex`	varying
File with format	`blink.hex:i`	varying
Character	`'A'`	1
Binary integer	0b101010	1, 2, 4, or 8
Octal integer	012345	1, 2, 4, or 8
Decimal integer	12345	1, 2, 4, or 8

Hexadecimal integer	0x12345	1, 2, 4, or 8
Decimal float	3.1415926	4
Hexadecimal float	0xA.8p2	4
Decimal double	3.141592653589793D	8
Hexadecimal double	0xA.8p2D	8

data can be binary, octal, decimal or hexadecimal integers, floating point numbers or C-style strings and characters. If nothing matches, `data` will be interpreted as a name of a file containing data, which will be read and inserted at this point. In order to force the interpretation of a data item as file, e.g., when the file name would be understood as a number otherwise, the file name can be given a `:`*f* format specifier. In absence of a format suffix, the terminal will try to auto-detect the file format.

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` hexadecimal and `0b` binary integers are always treated as unsigned. `+0x`, `-0x`, `+0b` and `-0b` numbers with an explicit sign 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` hexadecimal and `0b` binary 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 hexadecimal/binary 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 `0xFFFFffffFFFFffffU`.

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

`write` *memory addr data*
: The start address `addr` may be omitted if the size of the memory being written to is one byte.

`write` *memory addr len data[,] {data[,]} ...*
: The ellipsis ... form writes the data to the entire memory intervall addressed by *addr len* and, if necessary, pads the remaining space by repeating the last data item. The fill write command does not write beyond the specified memory area even if more data than needed were given.

`save` *memory {addr len} file[:format]*
: Save one or more memory segments to a file in a format specified by the :format letter. The default is `:r` for raw binary. Each memory segment is described by an address and length pair. In absence of any memory segments the entire memory is saved to the file. Only Motorola S-Record (`:s`) and Intel Hex (`:i` or `:I`) formats store address information with the saved data. Avrdude cannot currently save ELF file formats. All the other file formats lose the address information and concatenate the chosen memory segments into the output file. If the file name is - then avrdude writes to stdout.

`erase`
: Perform a chip erase and discard all pending writes to EEPROM and flash. Note that EEPROM will be preserved if the EESAVE fuse bit is set.

`erase` *memory*
: Erase the entire specified memory.

`erase` *memory addr len*
: Erase a section of the specified memory.

`flush`
: Synchronise with the device all pending writes to flash, EEPROM, bootrow and usersig. With some programmer and part combinations, flash (and sometimes EEPROM, too) looks like a NOR memory, i.e., a write can only clear bits, never set them. For NOR memories a page erase or, if not available, a chip erase needs to be issued before writing arbitrary data. Bootrow and usersig are generally unaffected by a chip erase. When a memory looks like a NOR memory, 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 routine should be called 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, EEPROM, bootrow and usersig cache.

`config [`*opts*`]`
: Show all configuration properties of the part; these are usually bitfields in fuses or lock bits bytes that can take on values, which typically have a mnemonic

name. Each part has their own set of configurable items. The option `-f` groups the configuration properties by the fuses and lock bits byte they are housed in, and shows the current value of these memories as well. Config `-a` outputs an initialisation script with all properties and all possible respective assignments. The currently assigned mnemonic values are the ones that are not commented out. The option `-v` increases the verbosity of the output of the config command.

`config [`*`opts`*`]` *`property`* `[`*`opts`*`]`
: Show the current value of the named configuration property. Wildcards or initial strings are permitted (but not both), in which case the current values of all matching properties are displayed.

`config [`*`opts`*`]` *`property=`* `[`*`opts`*`]`
: Show all possible values of the named configuration property (notice the trailing `=`). The one that is currently set is the only one not commented out. As before, wildcards or initial strings are permitted.

`config [`*`opts`*`]` *`property=value`* `[`*`opts`*`]`
: Modify the named configuration property to the given value. The corresponding fuse or lock bits will be changed immediately but the change will normally only take effect the next time the part is reset, at which point the fuses and lock bits are utilised. Value can either be a valid integer or one of the symbolic mnemonics, if known. Wildcards or initial strings are permitted for either the property or the assigned mnemonic value, but an assignment only happens if both the property and the name can be uniquely resolved.

  It is quite possible, as is with direct writing to the underlying fuses and lock bits, to brick a part, i.e., make it unresponsive to further programming with the chosen programmer: here be dragons.

`factory reset`
: Resets the connected part to factory state as far as possible (bootloaders, for example, cannot write fuses and may not have a means to erase EEPROM). This command may change the clock frequency F_CPU of the part after the next MCU reset when the changed fuse values come into effect. As such, this may require that future avrdude calls use a different bit clock rate up to F_CPU/4 for the programmer next time. Note that the command `factory` can be abbreviated but the required argument `reset` needs to be spelled out in full.

`regfile [`*`opts`*`]`
: `regfile` with no further argument displays the register file of a part, i.e., all register names and their contents in `io` memory, if possible: note that external programming cannot read the registers of classic parts (ISP or TPI interfaces).

  Option `-a` displays the register I/O addresses in addition; `-m` displays the register memory addresses used for `lds/sts` opcodes instead of the I/O addresses. Option `-s` also shows the size of the register in bytes whilst `-v` shows a slightly expanded register explanation alongside each register.

`regfile [`*`opts`*`]` *`reg`* `[`*`opts`*`]`
: `regfile` together with a register name *reg* shows all those registers that are matched by *reg*. Wildcards or partial strings are permitted but not both.

Register names have the form *module.name* or *module.instance.name*. If the provided *reg* is a full, existing register name, e.g., `porta.out` then that is the only register that is displayed even though that might be a partial name of another register, eg, `porta.outdir`. If the provided *reg* is the same as *instance.name* or *name* then partial matching is no longer utilised and all module registers with that exact *instance.name* or *name* are shown. Partial matching can be forced through use of wildcards, e.g., using `porta.out*`

`regfile [opts] reg=value [opts]`
: This sets a single register addressed by *reg* to the given *value*. Only external programming of modern parts (those with UPDI interface) can read from and write to register io memory, but as that memory is volatile, the contents will be lost after reset.

`include [opts] file`
: Include contents of the named file *file* as if it was typed. This is useful for batch scripts, e.g., recurring initialisation code for fuses. The include option `-e` prints the lines of the file as comments before processing them; on a non-zero verbosity level the line numbers are printed, too.

`signature`
: Display the device signature bytes.

`part`
: Display the current part information, including supported programming modes, memory and variants tables. Use *-m* to only print the memory table, and *-v* to only print the variants table.

`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.

`q`
: Can be used as an alias for `quit`.

`!line`
: Run the shell *line* in a subshell, e.g., `!ls *.hex`. Subshell commands take the rest of the line as their command. For security reasons, they must be enabled explictly by putting `allow_subshells = yes;` into your `${HOME}/.config/avrdude/avrdude.rc` or `${HOME}/.avrduderc` file.

`# comment`
: Place comments onto the terminal line (useful for scripts).

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*
: Set the SCK clock period to *period* microseconds. Note that some official Microchip programmers store the bitclock setting and will continue to use it until a diferent value is provided. See `-B bitclock` for more information.

`parms`
: Display programmer specific parameters.

3.2 Terminal Mode Examples

Enter terminal, display part parameters, modify EEPROM, perform a chip erase and quit:

```
$ avrdude -qq -c usbasp -p atmega328p -t

avrdude> part
ATmega328P with programming modes ISP, HVPP, debugWIRE, SPM

Memory        Size  Pg size
---------------------------
eeprom        1024        4
flash        32768      128
efuse            1        1
hfuse            1        1
lfuse            1        1
lock             1        1
signature        3        1
calibration      1        1
io             224        1
sram          2048        1

Variants         Package  F max   T range         V range
----------------------------------------------------------------
ATmega328P       N/A      20 MHz  [-40 C,   N/A]  [1.8 V, 5.5 V]
ATmega328P-15MZ  MLF32    20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-AN    TQFP32   20 MHz  [-40 C, 105 C]  [1.8 V, 5.5 V]
ATmega328P-ANR   TQFP32   20 MHz  [-40 C, 105 C]  [1.8 V, 5.5 V]
ATmega328P-AU    TQFP32   20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-AUR   TQFP32   20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-MMH   MLF28    20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-MMHR  MLF28    20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-MN    QFN32    20 MHz  [-40 C, 105 C]  [1.8 V, 5.5 V]
ATmega328P-MNR   MLF32    20 MHz  [-40 C, 105 C]  [1.8 V, 5.5 V]
ATmega328P-MU    MLF32    20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-MUR   MLF32    20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]
ATmega328P-PN    PDIP28   20 MHz  [-40 C, 105 C]  [1.8 V, 5.5 V]
ATmega328P-PU    PDIP28   20 MHz  [-40 C,  85 C]  [1.8 V, 5.5 V]

avrdude> 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 0xcafe "Avrdude"

avrdude> dump eeprom 0 16
0000  01 02 03 04 fe ca 41 76  72 64 75 64 65 00 ff ff  |......Avrdude...|

avrdude> flush

avrdude> erase
erasing chip ...

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

avrdude> quit
```

Program the fuse bits of an ATmega328P with a fuse calculator

- Enable full-swing high speed external crystal with long startup time
- Remove default clock division by 8
- Make reset jump to boot loader
- Set the size of the bootloader to 512 bytes (256 words)
- Enable brown-out detection at 2.7 V

First display the factory defaults, then consult an external fuse calculator, select the ATmega328P part, find above settings, note the ensuinig new values for the three fuses and reprogram:

```
$ avrdude -c usbasp -p atmega328p -t

avrdude: AVR device initialized and ready to accept instructions
avrdude: device signature = 0x1e950f (probably m328p)

avrdude> dump efuse
Reading | ################################################## | 100% 0.00 s
0000  ff                                                |.               |

avrdude> dump hfuse
Reading | ################################################## | 100% 0.00 s
0000  d9                                                |.               |

avrdude> dump lfuse
Reading | ################################################## | 100% 0.00 s
0000  62                                                |b               |

avrdude> #
avrdude> # Consult external fuse calculator
avrdude> #
avrdude> #

avrdude> write efuse 0xfd
Writing | ################################################## | 100% 0.01 s

avrdude> write hfuse 0xde
Writing | ################################################## | 100% 0.01 s

avrdude> write lfuse 0xf7
Writing | ################################################## | 100% 0.01 s

avrdude> quit

avrdude done.  Thank you.
```

Program the fuse bits of an ATmega328P with the config command

```
$ avrdude -qq -c usbasp -p atmega328p -t

avrdude> # Show all configurations
avrdude> config
config sut_cksel=intrcosc_8mhz_6ck_14ck_65ms # 34
config ckout=co_disabled # 1
config ckdiv8=by_8 # 0
config bootrst=application # 1
config bootsz=bs_2048w # 0
config eesave=ee_erased # 1
config wdton=wdt_programmable # 1
config spien=isp_enabled # 0
config dwen=dw_off # 1
config rstdisbl=external_reset # 1
config bodlevel=bod_disabled # 7
config lb=no_lock # 3
config blb0=no_lock_in_app # 3
config blb1=no_lock_in_boot # 3

avrdude> # Show possible values for full-swing external crystal
avrdude> config sut_cksel=extfs
avrdude warning: (config) ambiguous; known sut_cksel extfs symbols are:
 - sut_cksel=extfsxtal_258ck_14ck_4ms1 #  6
 - sut_cksel=extfsxtal_1kck_14ck_65ms  #  7
 - sut_cksel=extfsxtal_258ck_14ck_65ms # 22
 - sut_cksel=extfsxtal_16kck_14ck_0ms  # 23
 - sut_cksel=extfsxtal_1kck_14ck_0ms   # 38
 - sut_cksel=extfsxtal_16kck_14ck_4ms1 # 39
 - sut_cksel=extfsxtal_1kck_14ck_4ms1  # 54
 - sut_cksel=extfsxtal_16kck_14ck_65ms # 55

avrdude> # Set the one with appropriate startup times
avrdude> c su=55

avrdude> # Unprogram clock division by 8, make reset jump to boot loader
avrdude> c ckdiv8=1
avrdude> c bootrst=boot
avrdude> c bootsz=bs_256w

avrdude> # Query which bod levels exist; set brown-out at 2.7 V
avrdude> c bodlevel=
# conf bodlevel=bod_4v3      # 4
# conf bodlevel=bod_2v7      # 5
# conf bodlevel=bod_1v8      # 6
config bodlevel=bod_disabled # 7 (factory)
avrdude> c bod=*2v7

avrdude> quit
```

Show and change registers

```
$ avrdude -c xplainedpro_updi -p ATtiny817 \
  -T "reg ctrlc" -T "reg usart0.baud=0x1234" -T "reg -asv usart0"

avrdude_pr1580: AVR device initialized and ready to accept instructions
avrdude_pr1580: device signature = 0x1e9320 (probably t817)

avrdude_pr1580: processing -T reg ctrlc
0x00 portmux.ctrlc
0x00 adc0.ctrlc
0x03 usart0.ctrlc
0x00 tca0.ctrlc
0x00 tcd0.ctrlc

avrdude_pr1580: processing -T reg usart0.baud=0x1234

avrdude_pr1580: processing -T reg -asv usart0
I/O 0x800: (1)   0x00 usart0.rxdatal  # Receive data low byte
I/O 0x801: (1)   0x00 usart0.rxdatah  # Receive data high byte
I/O 0x802: (1)   0x00 usart0.txdatal  # Transmit data low byte
I/O 0x803: (1)   0x00 usart0.txdatah  # Transmit data high byte
I/O 0x804: (1)   0x20 usart0.status   # Status register
I/O 0x805: (1)   0x00 usart0.ctrla    # Control register A
I/O 0x806: (1)   0x00 usart0.ctrlb    # Control register B
I/O 0x807: (1)   0x03 usart0.ctrlc    # Control register C
I/O 0x808: (2) 0x1234 usart0.baud     # Baud rate register (16 bits)
I/O 0x80b: (1)   0x00 usart0.dbgctrl  # Debug control register
I/O 0x80c: (1)   0x00 usart0.evctrl   # Event control register
I/O 0x80d: (1)   0x00 usart0.txplctrl # IRCOM transmitter pulse length control register
I/O 0x80e: (1)   0x00 usart0.rxplctrl # IRCOM receiver pulse length control register

avrdude_pr1580 done.  Thank you.
```

Show register file of a classic part

```
$ avrdude -qq -p ATtiny11 -c dryrun -T "regfile -av"

I/O 0x08: ac.acsr     # Analog comparator control and status register
I/O 0x16: portb.pinb  # Port B input register
I/O 0x17: portb.ddrb  # Port B data direction register
I/O 0x18: portb.portb # Port B data register
I/O 0x21: wdt.wdtcr   # Watchdog timer control register
I/O 0x32: tc0.tcnt0   # Timer/counter 0
I/O 0x33: tc0.tccr0   # T/C 0 control register
I/O 0x34: cpu.mcusr   # MCU status register
I/O 0x35: cpu.mcucr   # MCU control register
I/O 0x38: tc0.tifr    # T/C interrupt flag register
I/O 0x39: tc0.timsk   # T/C interrupt mask register
I/O 0x3a: exint.gifr  # General interrupt flag register
I/O 0x3b: exint.gimsk # General interrupt mask register
I/O 0x3f: cpu.sreg    # Status register
```

The following example demonstrates **negative address and length bytes**, and the **fill form of the `write` command using an ellipis ...** where the last data item provided is used to fill up the indicated memory range.

```
$ avrdude -c usbasp -p atmega328p -t

avrdude: AVR device initialized and ready to accept instructions
avrdude: device signature = 0x1e950f (probably m328p)

avrdude> dump flash -64 -33
Reading | ################################################## | 100% 0.02 s
7fc0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
7fd0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude> dump flash -32 -1
Reading | ################################################## | 100% 0.00 s
7fe0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
7ff0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|

avrdude> write flash -64 1234567890 'A' 'V' 'R' 2.718282 0xaa 0xbb 0xcc "Hello World!"
Caching | ################################################## | 100% 0.00 s

avrdude> write flash -32 -1 0x01 0b00000010 0b11 0x04 0x05 ...
Caching | ################################################## | 100% 0.00 s

avrdude> read flash -64 -33
Reading | ################################################## | 100% 0.00 s
7fc0  d2 02 96 49 41 56 52 55  f8 2d 40 aa bb cc 48 65  |...IAVRU.-@...He|
7fd0  6c 6c 6f 20 57 6f 72 6c  64 21 00 ff ff ff ff ff  |llo World!......|

avrdude> read flash -32 -1
Reading | ################################################## | 100% 0.00 s
7fe0  01 02 03 04 05 05 05 05  05 05 05 05 05 05 05 05  |................|
7ff0  05 05 05 05 05 05 05 05  05 05 05 05 05 05 05 05  |................|

avrdude> flush
avrdude: synching cache to device ...
Writing | ################################################## | 100% 0.05 s

avrdude> quit

avrdude done.  Thank you.
```

Mixing terminal commands and `-U` memory operations: the example below burns a bootloader, uses a terminal line to write application data to flash, loads the application, configures the brownout detection level to 2.7 V and, finally, stores the full flash as new hex file. Note the use of different quotation marks in `bash` to pass the terminal command lines as single entity to AVRDUDE.

```
$ avrdude -qc dryrun -p m328p \
          -U urboot_m328p_1s_autobaud_uart0_pr_ee_ce.hex \
          -T 'write flash 0x7D00 0xc0cac01a 0xcafe "secret Coca Cola recipe"' \
          -U flash:w:cola-vending-machine.hex \
          -T "config -v bod=*2v7" \
          -U flash:r:app+data.hex:I

avrdude: AVR device initialized and ready to accept instructions
avrdude: device signature = 0x1e950f (probably m328p)
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: processing -U flash:w:urboot_m328p_1s_autobaud_uart0_pr_ee_ce.hex:i
avrdude: reading input file urboot_m328p_1s_autobaud_uart0_pr_ee_ce.hex for flash
         with 368 bytes in 2 sections within [0x7e80, 0x7fff]
         using 3 pages and 16 pad bytes
avrdude: writing 368 bytes flash ...
avrdude: 368 bytes of flash written
avrdude: verifying flash memory against urboot_m328p_1s_autobaud_uart0_pr_ee_ce.hex
avrdude: 368 bytes of flash verified

avrdude: processing -T write flash 0x7D00 0xc0cac01a 0xcafe "secret Coca Cola recipe"
avrdude: synching cache to device ... done

avrdude: processing -U flash:w:cola-vending-machine.hex:i
avrdude: reading input file cola-vending-machine.hex for flash
         with 736 bytes in 1 section within [0, 0x2df]
         using 6 pages and 32 pad bytes
avrdude: writing 736 bytes flash ...
avrdude: 736 bytes of flash written
avrdude: verifying flash memory against cola-vending-machine.hex
avrdude: 736 bytes of flash verified

avrdude: processing -T config -v bod=*2v7
config bodlevel=bod_2v7 # 5

avrdude: processing -U flash:r:app+data.hex:I
avrdude: reading flash memory ...
avrdude: writing output file app+data.hex

avrdude done.  Thank you.
```

4 Configuration Files

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`See Section A.1 [Unix], page 60, 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

`avrdude_conf_version = "`*`build-time-version`*`";`
: Automatically set during the build process.

`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_linuxgpio = "`*`default-linuxgpio-device`*`";`
: Assign the default gpiochip for linuxgpio's libgpiod mode, e.g. "gpiochip0". Ignored for linuxgpio's sysfs mode. 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_baudrate = "`*`default-baudrate`*`";`
: Assign the default baudrate value that will be used if the programmer doesn't provide its specific `baudrate` entry. Can be overridden using the `-b` option.

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

`allow_subshells = `*`no`*`;`
: Whether or not AVRDUDE's interactive terminal is allowed to use subshell `!` commands. This defaults to no for security reasons, e.g., in the rare case `avrdude -t` is set up with attached hardware to provide a web service, remote ssh or a login on a PC instead of a shell, say, for demo or training purposes. In almost all other cases this can be overridden in the per-user `avrddude.rc` or `.avrduderc` configuration file with *yes*.

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
                                              # list known types with -c ?type
    prog_modes = PM_<i/f> {| PM_<i/f>}        # interfaces, e.g., PM_SPM|PM_PDI (1)
    extra_features = HAS_<fea> {| HAS_<fea>}  # extra features, e.g., HAS_SUFFER (2)
    connection_type = parallel | serial | usb | spi
    baudrate = <num> ;                        # baudrate for avr910-programmer
    vcc      = <pin1> [, <pin2> ... ] ;       # pin number(s) (3)
    buff     = <pin1> [, <pin2> ... ] ;       # pin number(s)
    reset    = <pin> ;                        # pin number
    sck      = <pin> ;                        # pin number
    sdo      = <pin> ;                        # pin number
    sdi      = <pin> ;                        # pin number
    tck      = <pin> ;                        # pin number
    tdi      = <pin> ;                        # pin number
    tdo      = <pin> ;                        # pin number
    tms      = <pin> ;                        # pin number
    errled   = <pin> ;                        # pin number
    rdyled   = <pin> ;                        # pin number
    pgmled   = <pin> ;                        # pin number
    vfyled   = <pin> ;                        # pin number
    usbvid   = <hexnum> ;                     # USB vendor ID
    usbpid   = <hexnum> [, <hexnum> ...] ;    # USB product ID (4)
    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.

Notes

1. 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
2. The following extra programmer features are known
   - `HAS_SUFFER`: Only present on Xplained Mini/Nano programmers; the Super User Fantastic Feature Enable Register allows the user to modify the behavior of the mEDBG programmer/debugger chip, see the Xplained Mini/Nano documentation for more information
   - `HAS_VTARG_SWITCH`: Programer has a programmable target power switch
   - `HAS_VTARG_READ`: Programmer can read the target voltage
   - `HAS_VTARG_ADJ`: Programmer has an adjustable target power source that can be controlled with Avrdude
   - `HAS_FOSC_ADJ`: Programmer has a programable frequency generator that can clock an AVR directly through its XTAL1 pin
   - `HAS_VAREF_ADJ`: Programmer has an adjustable analog reference voltage that can be controlled with Avrdude
3. To invert the polarity of a pin, use a tilde `~<num>`; to invert the polarity of all pins in a list use `~(<num1> [, <num2> ... ])`
4. Not all programmer types can handle a list of USB PIDs

The following programmer types are currently implemented:

4.3 Serial Adapter Definitions

The format of a serial adapter definition is as follows:

```
serialadapter
    parent <id>                              # optional serialadapter or programmer parent
    id       = <id1> [, <id2> ... ] ;        # <idN> are quoted strings
    desc     = <description> ;               # quoted string
    baudrate = <num> ;                       # optional default baudrate, eg, in .avrduderc
    usbvid   = <hexnum> ;                    # USB vendor ID
    usbpid   = <hexnum> [, <hexnum> ...] ;   # list of USB product IDs
    usbsn    = <serialno> ;                  # USB Serial Number in per-user .avrduderc
;
```

Technically, a `serialadapter` is implemented as `programmer` that has only USB parameters defined. It can be used for a `-P <serialadapter>[:<serial number>]` port specification instead of the created serial port. Per-user serialadapter definitions in `~/.avrduderc` or `avrdude.rc` files can add a serial number to assign a particular board a specific id and default upload baud rate:

```
serialadapter parent "ft232r"
    id                     = "bike-shed-door";
    usbsn                  = "0123456789";
    baudrate               = 250000;
;
```

This is particularly useful for uploading to a bootloader as it allows specifying the port as `-P bike-shed-door` rather than having to figure out which serial port name the operating system has assigned to the plugged in bike-shed-door board at runtime. Note that each programmer that defines `usbpid` and sets `is_serialadapter = yes` can also be utilised as a serialadapter.

4.4 Part Definitions

```
part
    desc             = <description> ;        # quoted string, the long part name, eg, "ATmega328p"
    id               = <id> ;                 # quoted string, normally an abbreviated part name
    variants         = <str1> [, <str2> ...]; # quoted strings, each starts so "<alt-name>: ..."
    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> ;                # microseconds
    reset            = dedicated | io ;
    retry_pulse      = reset | sck ;
    # STK500 parameters (parallel programming IO lines)
    pagel            = <num> ;                # page load pin name in hex, e.g., 0xD7
    bs2              = <num> ;                # byte select 2 pin name in hex, e.g., 0xA0
    serial           = <yes/no> ;             # can use serial downloading
    parallel         = <yes/no/pseudo> ;      # can use parallel 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> ;
    resetdelayms     = <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
    eind             = <num> ;                # mem addr of EIND reg
    mcu_base         = <num> ;                # MCU control block in ATxmega devices
    nvm_base         = <num> ;                # NVM controller in ATxmega devices
    ocd_base         = <num> ;                # OCD module in AVR8X/UPDI devices
    syscfg_base      = <num> ;                # Chip revision ID in AVR8X/UPDI devices
    ocdrev           = <num> ;                # JTAGICE3 parameter from ATDF files
    pgm_enable       = <instruction format> ;
    chip_erase       = <instruction format> ;
    # parameters for bootloaders
    autobaud_sync    = <num> ;                # autobaud detection byte, default 0x30
    factory_fcpu     = <num> ;                # F_CPU in Hz on reset and factory-set fuses

    memory <memstr>
        paged           = <yes/no> ;          # yes/no (flash of classic parts only)
        offset          = <num> ;             # memory offset
        size            = <num> ;             # bytes
        page_size       = <num> ;             # bytes
        num_pages       = <num> ;             # numeric
        initval         = <num> ;             # factory setting of fuses and lockbits
        bitmask         = <num> ;             # bits used (only in fuses and lockbits)
        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`, `ocdrev`, `initval` and `bitmask`, all of which default to -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 arithemtic and bitwise operators.

4.4.1 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;`. The `variants` parameter is never inherited as it almost always would be a mistake to do so: `variants` defines a string list detailing variant names of the part followed by an optional colon, the package code and some absolute maximum ratings.

Example format for part inheritance:

```
part parent <id>                              # String identifying parent
    id              = <id> ;                  # Id string for new part
    <any set of other parameters from the list above>
;
```

4.4.2 Instruction Format

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

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

`0` The bit is always clear on input as well as output

`x` The bit is ignored on input and output

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

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

`i` The bit is an input data bit; as with `a` bits an input data bit can optionally be followed by a bit number, here between 0 and 7, if the bit needs to be moved to a different position in the SPI write command byte than it appears in memory.

`o` The bit is an output data bit; as with `i` bits an output data bit can optionally be followed by a bit number; this is useful in case the part's SPI read command places a particular bit into a different position than the write command put it, e.g., ATtiny22L or AT90S8535 lock bits.

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";
```

4.5 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 memories will implement all instructions.
- AVR Fuse bits and Lock bits are implemented as a type of memory.
- Example memories are: `flash`, `eeprom`, `fuse`, `lfuse` (low fuse), `hfuse` (high fuse), `efuse` (extended fuse), `signature`, `calibration`, `lock`.
- The memory specified on the AVRDUDE command line must match one of the memories 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:

  `https://www.microchip.com/content/dam/mchp/documents/OTH/ProductDocuments/DataSheets/doc1042.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-ATTINY10	ATtiny4 ATtiny5 ATtiny9 ATtiny10
STK600-RC008T-2	STK600-DIP	ATtiny11 ATtiny12 ATtiny13 ATtiny13A ATtiny25 ATtiny45 ATtiny85
STK600-RC008T-7	STK600-DIP	ATtiny15
STK600-RC014T-42	STK600-SOIC	ATtiny20
STK600-RC020T-1	STK600-DIP	ATtiny2313 ATtiny2313A ATtiny4313
	STK600-TinyX3U	ATtiny43U
STK600-RC014T-12	STK600-DIP	ATtiny24 ATtiny44 ATtiny84 ATtiny24A ATtiny44A
STK600-RC020T-8	STK600-DIP	ATtiny26 ATtiny261 ATtiny261A ATtiny461 ATtiny861 ATtiny861A
STK600-RC020T-43	STK600-SOIC	ATtiny261 ATtiny261A ATtiny461 ATtiny461A 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 ATmega48P ATmega48PA ATmega88P ATmega88PA 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 ATmega32 ATmega32A ATmega164P ATmega164PA ATmega324P ATmega324PA ATmega644 ATmega644P ATmega644PA ATmega1284P
STK600-RC044M-31	STK600-TQFP44	ATmega8535 ATmega16 ATmega16A ATmega32 ATmega32A ATmega164P ATmega164PA 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 ATmega169P ATmega169PA ATmega325 ATmega325P ATmega329 ATmega329P ATmega645 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-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 AT32UC3A0256 AT32UC3A0128
STK600-RCuC3A100-28	STK600-TQFP100	AT32UC3A1512 AT32UC3A1256 AT32UC3A1128

`STK600-RCuC3B0-21`	`STK600-TQFP64-2`	AT32UC3B0256 AT32UC3B0512RevC AT32UC3B0512 AT32UC3B0128 AT32UC3B064 AT32UC3D1128
`STK600-RCuC3B48-27`	`STK600-TQFP48`	AT32UC3B1256 AT32UC3B164
`STK600-RCUC3A144-32`	`STK600-TQFP144`	AT32UC3A3512 AT32UC3A3256 AT32UC3A3128 AT32UC3A364 AT32UC3A3256S AT32UC3A3128S AT32UC3A364S
`STK600-RCUC3C0-36`	`STK600-TQFP144`	AT32UC3C0512 AT32UC3C0256 AT32UC3C0128 AT32UC3C064
`STK600-RCUC3C1-38`	`STK600-TQFP100`	AT32UC3C1512 AT32UC3C1256 AT32UC3C1128 AT32UC3C164
`STK600-RCUC3C2-40`	`STK600-TQFP64-2`	AT32UC3C2512 AT32UC3C2256 AT32UC3C2128 AT32UC3C264
`STK600-RCUC3C0-37`	`STK600-TQFP144`	AT32UC3C0512 AT32UC3C0256 AT32UC3C0128 AT32UC3C064
`STK600-RCUC3C1-39`	`STK600-TQFP100`	AT32UC3C1512 AT32UC3C1256 AT32UC3C1128 AT32UC3C164
`STK600-RCUC3C2-41`	`STK600-TQFP64-2`	AT32UC3C2512 AT32UC3C2256 AT32UC3C2128 AT32UC3C264
`STK600-RCUC3L0-34`	`STK600-TQFP48`	AT32UC3L064 AT32UC3L032 AT32UC3L016
	`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 AVRs that have an ADC, with the power supply pins in the center of the package) was used but an ATmega8515 inserted (which uses the “industry standard” pinout with Vcc and GND at opposite corners).

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

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

High-voltage programming requires the target voltage to be set to at least 4.5 V in order to work. This can be done using *Terminal Mode*, see Chapter 3 [Terminal Mode Operation], page 33.

5.2 DFU Bootloader Using FLIP Version 1

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

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

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

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

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

5.3 SerialUPDI Programmer

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

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

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

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

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

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;
    ;
;
```

The point of USERROW is to provide ability to write configuration details to already locked device and currently SerialUPDI interface supports this feature. 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`.

In case you run into issues with the SerialUPDI interface, 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.

5.4 Programmer LED Management

Some hardware programmers have LEDs, and the firmware controls them fully without AVRDUDE having a way to influence their LED states. Other programmers have LEDs and expect the host downloader/uploader to handle them, for example bit-banging programmers, ftdi-based programmers or linuxgpio programmers. For those programmers AVRDUDE provides support of four LEDs (RDY, ERR, PGM and VFY) which can be set via corresponding subroutines in the code for the respective `-c` programmer.

The RDY LED is set once the programmer is initialised and switched off when AVRDUDE exits. During reading, writing or erasing the target the PGM LED flashes with around 2.5 Hz, whilst the VFY LED comes on during -U verification of the uploaded contents. Errors are indicated with the ERR LED.

Assuming AVRDUDE got to the point where LEDs are accessible and the RDY LED was switched on then, on exit, AVRDUDE will leave the LEDs in the following states:

PGM	VFY	ERR	Semantics
off	off	off	OK: all tasks done without errors
off	off	on	Some error not related to read, write or erase
on	off	on	Read, write or erase error
off	on	on	Verification error but no read, write or erase error
on	on	on	Verification error and read, write or erase error

Other combinations should not show after exit.

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-7.3.tar.gz | tar xf -
$ cd avrdude-7.3
$ ./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-7.3.tar.gz | tar xf -
$ cd avrdude-7.3
$ ./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-7.3.tar.gz
# rpm -Uvh /usr/src/redhat/RPMS/i386/avrdude-7.3-1.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	`/dev/ppi0`	`/dev/cuad0`
Linux	`/dev/parport0`	`/dev/ttyS0`
Solaris	`/dev/printers/0`	`/dev/term/a`

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-7.3.tar.gz | tar xf -
$ cd avrdude-7.3
$ ./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 Windows Configuration Files

A.2.2.1 Windows 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` in the same directory where avrdude.exe is located.

A.2.2.2 Windows Configuration File Location

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 Windows Port Names

A.2.3.1 Windows 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 Windows Parallel Ports

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

`lpt1` 0x378

`lpt2` 0x278

`lpt3` 0x3BC

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

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

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: Page-mode programming the EEPROM (using the -U option) does not erase EEPROM cells before writing, and thus cannot necessarily overwrite previous values that are different to 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)
  ```

  For USBasp, the same format for *-P usb* can be used to match usb bus/device. Alternatively, device serial number can be specified as follows (for serial number '1234').

  ```
  avrdude -c USBasp -p atmega8 -P usb:1234
  ```

- 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 cable 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 Vcc = 5 V.

- Problem: My ATxmega...A1/A2/A3 cannot be programmed through PDI with my AVR Dragon. Programming through a JTAG ICE mkII works though, as does programming through JTAG.

  Solution: None by this time (2010 Q1).

  It is said that the AVR Dragon can only program devices from the A4 Xmega sub-family.

- Problem: when programming with an AVRISPmkII or STK600, AVRDUDE hangs when programming files of a certain size (e.g. 246 bytes). Other (larger or smaller) sizes work though.

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

  See also: `http://www.libusb.org/ticket/6`

- Problem: after flashing a firmware that reduces the target's clock speed (e.g. through the `CLKPR` register), further ISP connection attempts fail. 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

P

S

T

U

W