```
================================================================================
AxiCat AVR ISP v1.1.0 User Manual

2016-11-06  Peter S'heeren, Axiris
--------------------------------------------------------------------------------


Overview
--------

Program axicatavrisp turns your AxiCat into an in-system programming tool for
use with AVR 8-bit microcontrollers.
```

Command Line

The program accepts numerous command line arguments. Some arguments accept one or more options.

An argument always begins with a hyphen, e.g. -device.

An option can be many things, like a path, a string, a label, a number. If you need to include spaces in an option, place quotation marks around the text. For example:

```
-flash "C:\My Documents\AVR Files\bootloader.hex"
```

A number can be written in decimal, hexadecimal, or binary. The first digit must be decimal (0..9). An indicator at the end determines the radix. If no radix is specified, the server assumes a decimal value. Character sequence 0x or 0X at the beginning indicates a hexadecimal value. The value may contain underscores to improve readability. Numbers are case-insensitive. Examples:

```
Decimal:        1  10d  1_655_432  255D

Hexadecimal:    0AFh  1234_eee_h  0AbbaH 0xFF 0x1234_5678 0XAA

Binary:         1100b  11_0011_0010_B
```

Specify argument -h for a concise overview. You can always specify -h even if other arguments are already there. This is useful at times when you're entering a list of arguments and need to browse the overview; enter -h, press enter, have a look, recall your last entry on the command line and continue editing.

```
Device
------

Device is a synonym for AVR 8-bit microcontroller.

The program supports numerous microcontrollers. Run with argument -info devices
to see a list of supported devices.

You can run the program with or without a physical device connected. If you
specify argument -axicat, the program expects an AxiCat with a device attached.
If you omit argument -axicat, the program will skip all actions that involve
access to AxiCat and device.

When you run without -axicat, you've to specify -device in order to tell the
program what type of device it's targetting.

The program accepts three combinations of arguments -device and -axicat:

(1) The program works with a specified type of device but doesn't access any
    hardware. For example:

    -device attiny45

(2) The program expects the presence of the specified AxiCat and detects the
    type of device. For example:

    -axicat /dev/ttyUSB0

(3) The program expects the presence of the specified AxiCat, detects the type
    of device, and checks if it matches the specified device. For example:

    -device attiny45 -axicat /dev/ttyUSB0
```

AxiCat

Related command line arguments:

-axicat PATH	Serial path of the AxiCat.
-gpio N	GPIO pin that acts as the reset line.
-spispeed N	Speed of the SPI bus.

When you specify the -axicat argument, the program expects the presence of an AxiCat and an attached device.

The AVR ISP protocol is based on the SPI bus. The device and the AxiCat must be interconnected as follows:

Device pin	AxiCat pin	
MOSI	MOSI	K2.19
MISO	MISO	K2.21
SCLK	SCLK	K2.23
-	/SS0	K2.24 (not connected)
/RESET	GPIOn	<any available GPIO pin>
GND	GND	<any available ground pin>

Always check the 3.3V/5V setting of your AxiCat (JP1) before powering up.

The AxiCat acts as the SPI master, the device is the SPI slave. There can be no other SPI slaves attached to the SPI bus.

There's no slave select line involved on the slave side. However, the AxiCat always drives one of the slave select line by design. The program uses SS0.

By default the program uses GPIO0 (K2.8) as the reset line. Specify argument -gpio to explicitly select a GPIO pin.

When the program has established communication with the device, it always reads the signature bytes, lock bits, fuse bits and calibration byte. When you've specified -v, the program prints information about the device. Note that the calibration byte isn't used.

If the device signature is unknown, the program exits. You can query a list of supported devices with argument -info devices.

With argument -device you can explicitly specify the type of device the program is to expect. If the program detects a different type, it exits. If -device is omitted, the program will proceed operations on any connected device as long as it's supported.

Configuration Data

Configuration data is a synonym for fuse bits and lock bits. The term refers to four 1-byte registers:

* Lock bits.
* Fuse low bits.
* Fuse high bits.
* Fuse extended bits.

Every supported device incorporates these registers. Each register is composed of a number of bit fields. The composition differs amongst the various families of AVR 8-bit microcontrollers.

The program maintains one instance of the configuration data in memory. As soon as the program has established communication with the device, it reads the configuration data bytes from the device into memory.

The program updates the configuration data in memory based on the -fuse and -lock arguments. If argument -write is specified, the program tracks which bits have effectively changed and only writes those configuration data bytes that will be modified in the device.

Understanding the lifetime of the configuration data in memory is important. When you request to dump the configuration data (see -dump argument), the program will dump the most recent state of the configuration data.

```
Print Information
-----------------

If you specify -info then the program prints the requested information and
exits. Other arguments are ignored.

Currently, only one option is defined:

  -info devices             Prints a list of all supported devices.
```

```
Data Buffers for Flash and EEPROM
---------------------------------

Argument -write instructs the program to write data to flash memory or EEPROM.
The data bytes to be written to the device are first stored in data buffers:

* Data buffer for flash memory.

* Data buffer for EEPROM.

The bytes can come from various sources:

* An Intel HEX file.

* A raw file.

* Bytes specified on the command line.

* The device itself.

Arguments -flash and -eeprom store bytes in their corresponding data buffer.
Options are:

  -flash ...            Select data buffer for flash memory:

     FILE                  Load IHEX file into data buffer.

     ihex FILE             Load IHEX file into data buffer.

     raw ...               Load file contents into data buffer:

        AD FILE               Starting at address AD.

        page P FILE           Starting at page P.

        AD page P FILE        Starting at page P plus byte offset AD.

     bytes ...             Store an array of bytes in data buffer:

        AD BYTES              Starting at address AD.

        page P BYTES          Starting at page P.

        AD page P BYTES       Starting at page P plus byte offset AD.

     dev ...               Read from device into data buffer:

        all                   Entire memory space.

        AD to AD2             AD2-AD+1 bytes starting at address AD.

        AD cnt N              N bytes starting at address AD.

        AD to AD2 page P      AD2-AD+1 bytes starting at page P plus byte
                              offset AD.

        AD cnt N page P       N bytes starting at page P plus byte offset AD.

        page P                Page P.

        page P to P2          Pages P..P2.

        page P cnt N          N pages starting at page P.
```

```
        AD page P             One page starting at page P plus byte offset AD.

        AD page P to P2       P2-P+1 pages starting at page P plus byte offset AD.

        AD page P cnt N       N pages starting at page P plus byte offset AD.

  -eeprom ...           Select data buffer for EEPROM:

     FILE                   Load IHEX file into data buffer.

     ihex FILE              Load IHEX file into data buffer.

     raw AD FILE            Load file contents into data buffer starting at
                            address AD.

     bytes AD BYTES         Store an array of bytes in data buffer starting at
                            address AD.

     dev ...                Read from device into data buffer:

        all                    Entire memory space.

        AD to AD2              AD2-AD+1 bytes starting at address AD.

        AD cnt N               N bytes starting at address AD.


Examples:

  -flash ihex program.hex

  -flash raw 0A400h image.bin

  -flash raw page 20 image.bin

  -flash bytes 5 page 10 65 66 66 65 0

  -flash dev all

  -flash dev page 4 to 7

  -eeprom settings.eep

  -eeprom bytes 0000h 120 35 6 80

  -eeprom dev 0000h cnt 40


You can specify these arguments multiple times. Each argument is added to a
list. There's a list for -flash arguments and a list for -eeprom arguments.

After the program has processed the command line, it iterates through each list
and adds the data from the specified source to the corresponding data buffer.
```

Flash memory Details

Flash memory, a.k.a. program memory, is the place where program code is stored. The microcontroller executes instructions stored in this memory. Flash memory also contains constant data, ASCII strings for example.

From the instruction viewpoint, program memory is read-only. It's not possible to directly write data to flash.

Note that a device may implement dedicated instructions for erasing and writing flash memory. These instructions indirectly modify flash memory by controlling specific hardware.

Access to the flash memory comes in different forms, depending on which part of the device accesses the memory space:

* Processor:

  The processor sees a read-only memory space of 8-bit words. Each address thus contains one byte.

* AVR ISP protocol:

  * Read-access:

    * The memory space consists of 16-bit words.
    * Each address contains one 16-bit word.
    * A 16-bit word is ordered LSB->MSB.
    * Each word can be read individually.

  * Write-access:

    The memory space is page-oriented:

    * A page is an array of 16-bit words.
    * Each 16-bit word is ordered LSB->MSB.
    * A page has a fixed length, a power of 2, e.g. 64 words/page.
    * The flash memory is made up of a range of pages, e.g. 128 pages.
    * When a page is written, bits can only be cleared, they are never set to one. As a consequence, writing to a previously written page may result in even more bits being cleared.
    * The only way to set bits to one is by erasing the device.

The program's data buffer for flash memory is page-oriented as well. The data buffer is organized as a page table. The size of the pages is the same as the page size of the device, although it stores bytes rather than 16-bit words.

The data buffer is initially empty, no pages are present in the page table. As data is added (see -flash argument), the program adds the necessary pages for the targetted address ranges and stores the bytes in these pages. If a page for an address range already exists, the data bytes will be overwritten.

A new page is always initialized to one-bits before the data bytes are stored. This is consistent with the state of erased flash memory where all bits are one.

When -write is specified, the program will only write the pages that are present in the page table.

Example: ATtiny25

The ATtiny25 microcontroller has 1024 words or 2048 bytes of flash memory. A page consists of 16 words. The various forms of memory access are:

* Processor read-access: addresses 0000h..07FFh, 2048 bytes

* AVR ISP read-access: addresses 0000h..03FFh, 1024 words

* AVR ISP write-access: 16 words/page, 64 pages

* Program's data buffer: 32 bytes/page, 0-64 pages (added dynamically)

Now let's see how the program builds up the page table of the data buffer for various example -flash arguments.

```
-flash bytes 0004h 41h 42h 43h 44h

+-[PT]-+
|      | Page table
+------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: FF FF FF FF 41 42 43 44 FF FF FF FF FF FF FF FF | ad. 0000h
      |+10h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | page 0
      +-------------------------------------------------------+
```

The program adds page 0 and stores the data bytes at 0004h..0007h.

```
-flash bytes 0090h 1 2 3 4 5 -flash bytes 0092h 6 7

+-[PT]-+
|      | Page table
+------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ad. 0080h
      |+10h: 01 02 06 07 05 FF FF FF FF FF FF FF FF FF FF FF | page 4
      +-------------------------------------------------------+
```

The first -flash argument provides data bytes for addresses 0090h..0094h. The program adds page 4 and stores the data bytes.

The second -flash argument carries data bytes for address range 0092h..0093h. Since the page is already present, the program overwrites the address range with the specified data bytes.

```
-flash bytes 025Eh 61h 62h 63h 64h

+-[PT]-+
|      | Page table
+------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ad. 0240h
   |  |+10h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF 61 62 | page 18
   |  +-------------------------------------------------------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: 63 64 FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ad. 0260h
      |+10h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | page 19
      +-------------------------------------------------------+
```

The specified data bytes cross a page boundary, hence the program adds two consecutive pages that span the specified address range (025Eh..0261h).

```
-flash bytes 025Eh 61h 62h 63h 64h -flash bytes 0004h 41h 42h 43h 44h

+-[PT]-+
|      | Page table
+------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: FF FF FF FF 41 42 43 44 FF FF FF FF FF FF FF FF | ad. 0000h
   |  |+10h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | page 0
   |  +-------------------------------------------------------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ad. 0240h
   |  |+10h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF 61 62 | page 18
   |  +-------------------------------------------------------+
   |
   |  +-[PAGE]-----------------------------------------------+
   +--|+00h: 63 64 FF FF FF FF FF FF FF FF FF FF FF FF FF FF | ad. 0260h
      |+10h: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF | page 19
      +-------------------------------------------------------+
```

The program first adds pages 18 and 19, then page 0.

Note that the pages are added in the order that the -flash arguments appear in the program's command line.

The examples use literal bytes for simplicity. The program manages the page table similarly when you specify other options with the -flash argument.

You can dump the data buffer with argument -dump buf flash. Let's take the last example:

```
$ axicatavrisp -v -device attiny25 -dump buf flash -flash bytes 025Eh 61h 62h
63h 64h -flash bytes 0004h 41h 42h 43h 44h
```

EEPROM Details

Access to the EEPROM comes in different forms, depending on which part of the device accesses the memory space. We'll only discuss the viewpoint of the AVR ISP protocol here.

* Read-access:
  * The memory space consists of 8-bit words a.k.a. bytes.
  * Each address contains one byte.
  * Each byte can be read individually.
* Write-access:
  * The memory space consists of 8-bit words a.k.a. bytes.
  * Each address contains one byte.
  * Each byte can be written individually.
  * Similar to flash, writing to EEPROM involves clearing bits. However, the device will always erase a byte location before a new value is written. So from the viewpoint of the AVR ISP protocol, EEPROM bytes can be written without any side effect i.e. similar to writing random-access memory.

The program's data buffer for EEPROM keeps track of each byte that's being stored. When the program writes to EEPROM, it'll only write those bytes that were marked as stored.

Note that the device does have a concept of page-oriented EEPROM. Each device's datasheet expresses EEPROM sizes using page size and page count. The program also uses these values for describing the supported devices. This is clearly visible in its output, for example:

```
$ axicatavrisp -v -device ATmega164A
```

The output will tell you that the EEPROM is made up of 128 pages each sized 64 words per page.

You can dump the data buffer with argument -dump buf eeprom:

```
$ axicatavrisp -v -device attiny25 -dump buf eeprom -eeprom bytes 0020h 120
```

When the program dumps large chunks of EEPROM memory, you may notice a break on 256-byte boundaries. This is nothing to worry about. Internally, the data buffer for EEPROM uses a page table, similar to the data buffer for flash memory. The page size is set to 256 bytes, hence the break. The page size has no influence on the program's EEPROM operations whatsoever. Here's an example of how to observe a break:

```
$ axicatavrisp -v -device atmega164a -dump buf eeprom -eeprom bytes 00FCh 1 2
  3 4 5 6 7 8
```

Lock Bits

Argument -lock accepts the following options:

Option	Range	Bits
byte	0..255	Lock register
lb	0..3	LB[2..1]
blb0	0..3	BLB0[2..1]
blb1	0..3	BLB1[2..1]

For options that span two or more bits (like CKSEL), the command line parser accepts values 0..255. The program checks the validity of these values once the device type is known.

You may set any combination of these bits. The program will only use the settings that are available in the device.

The byte option specifies a value for the 8-bit lock register. When this option is combined with other options, it's applied before the other options.

See the datasheet of the microcontroller for the exact meaning of the bits.

Examples:

```
-lock lb 1

-lock lb 2 blb0 0 blb1 3

-lock byte 0xff

-lock lb 00b byte 0FFh
```

1. The lock register is set to FFh.
2. LB is set to 00b, thus the lock register is set to FCh.

Multiple -lock arguments are accumulated. When an option occurs multiple times, the program retains the latest occurance.

```
Fuse Bits
---------

Argument -fuse accepts the following options:

  Option      Range   Bits
  ------      ------  ------------------------
  low         0..255  Fuse low register
  high        0..255  Fuse high register
  ext         0..255  Fuse extended register
  cksel       0..15   CKSEL[3..0]
  sut         0..7    SUT[2..0]
  ckout       0..1    CKOUT
  ckdiv8      0..1    CKDIV8
  bodlevel    0..7    BODLEVEL[2..0]
  bootrst     0..1    BOOTRST
  bootsz      0..3    BOOTSZ[1..0]
  eesave      0..1    EESAVE
  wdton       0..1    WDTON
  spien       0..1    SPIEN
  dwen        0..1    DWEN
  jtagen      0..1    JTAGEN
  rstdisbl    0..1    RSTDISBL
  ocden       0..1    OCDEN
  selfprgen   0..1    SELFPRGEN
  hwbe        0..1    HWBE
  oscsel      0..3    OSCSEL[1..0]
  cfd         0..1    CFD
  boden       0..1    BODEN
  ckopt       0..1    CKOPT
  m103c       0..1    M103C

For options that span two or more bits (like CKSEL), the command line parser
accepts values 0..255. The program checks the validity of these values once the
device type is known.

You may set any combination of these bits. The program will only use the
settings that are available in the device.

Option low, high and ext specify a value for the corresponding 8-bit register.
When such option is combined with other options that target bits in the same
register, it's applied before the other options.

See the datasheet of the microcontroller for the exact meaning of the bits.

Examples:

  -fuse ckdiv8 1 cksel 0000b bodlevel 4

  -fuse low 0xff high 0xd9 ext 0xfd

Multiple -fuse arguments are accumulated. When an option occurs multiple times,
the program retains the latest occurance.
```

Erase Device

Argument -erase commands the program to send command Erase Chip to the device.
The device performs the following actions:

1. Set all bits in EEPROM to one.

2. Set all bits in flash memory to one.

3. Set lock bits to one, thus to their unprogrammed state.

For more details, see the datasheet of the microcontroller.

```
Write to Device
---------------

Argument -write tells the program to write information to the device. The write
operation is quite elaborate, it encompasses the following steps:

1. Erase the device. You can disable this step with -noerase.

2. Write bytes from the data buffer for flash memory to the device. This step
   is skipped when no data is stored in the buffer.

3. Verify the data that was written to flash memory. Specify -noverify to skip
   this step.

4. Write bytes from the data buffer for EEPROM to the device. This step is
   skipped when no data is stored in the buffer.

5. Verify the data that was written to EEPROM. Specify -noverify to skip this
   step.

6. Write the lock bits to the device. This step is skipped if no changes are to
   be committed.

7. Write the fuse low bits to the device. This step is skipped if no changes
   are to be committed.

8. Write the fuse high bits to the device. This step is skipped if no changes
   are to be committed.

9. Write the fuse extended bits to the device. This step is skipped if no
   changes are to be committed.

Examples:

    -write -flash ihex main.hex

    -write -noverify -flash raw 0F000h image.bin
```

```
Verify Device
-------------

Argument -verify tells the program to verify the contents of the data buffers
with the data stored in the device.

Examples:

    -verify -flash ihex main.hex

    -verify -eeprom bytes 20h 65 66 67 68
```

```
Save Data
---------

The -save argument takes one or more of the following options:

  flash ihex FILE       Save data buffor for flash memory to IHEX file.

  flash FILE            Save data buffor for flash memory to IHEX file.

  eeprom ihex FILE      Save data buffor for EEPROM to IHEX file.

  eeprom FILE           Save data buffor for EEPROM to IHEX file.

Examples:

  -save flash ihex core.hex

  -save flash tv.hex eeprom tv.eep
```

```
Dump Information
----------------

Argument -dump instructs the program to print certain information to standard
output. The argument takes one or more of the following options:

  buf flash          Dump the data buffer for flash memory.

  buf eeprom         Dump the data buffer for EEPROM.

  dev flash          Read the entire flash memory and dump.

  dev eeprom         Read the entire EEPROM and dump.

  cfg                Dump the configuration data (fuse and lock bits).

Examples:

  -dump cfg dev eeprom

    Dump configuration data and EEPROM.
```

```
About SPI Speed
---------------

The program sets a default SPI speed of 750000 Hz. You can override this speed
with the -spispeed N option. When you do, the program will apply a speed that's
equal or less than the specified value N.

An AVR microcontroller is shipped with CKDIV8 set to zero and CKSEL set to
0010b meaning the device will operate at 1 MHz when powered on for the first
time. As a consequence, you'll have to apply an SPI speed that's low enough to
accomodate the 1 MHz clock of the device. This argument will work:

  -spispeed 100000

You probably want to disable CKDIV8 first. Specify the following arguments:

  -spispeed 100000 -fuse ckdiv8 1 -write -noerase
```

```
Examples
--------

The examples presented here are applicable in Linux. It's assumed that the
serial path of your AxiCat is /dev/ttyUSB0. If this is not the case, change to
the correct serial path, for example:

  $ axicatavrisp -v -axicat /dev/ttyUSB4

In case your Linux requires root privileges for accessing the serial path, run
the examples as root.

If you want to run the examples in Windows, make sure the serial path is
prefixed with \\.\ to access the Win32 device namespace. For example:

  > axicatavrisp.exe -v -axicat \\.\COM25

Examples:

  $ axicatavrisp -v -axicat /dev/ttyUSB0

    Display information about the connected device. One typically runs this
    command to check if the ISP link is wired properly.

  $ axicatavrisp -v -device attiny45

    Display information about the ATtiny45.

  $ axicatavrisp -v -device attiny45 -dump cfg

    Display the default values for lock and fuse bits of the ATtiny45.

  $ axicatavrisp -v -axicat /dev/ttyUSB0 -device attiny45

    Check if the connected device is an ATtiny45.

  $ axicatavrisp -v -axicat /dev/ttyUSB0 -erase

    Erase flash memory and EEPROM, set lock bits to unprogrammed state.

  $ axicatavrisp -v -axicat /dev/ttyUSB0 -device ATmega164A -write -flash
    boot.hex -flash main.hex

    The program will:
    1. Check if the connected device is an ATmega164A.
    2. Load two IHEX file into the data buffer for flash memory.
    3. Erase the device.
    4. Write the bytes in the data buffer to flash memory.
    5. Verify the written data in flash memory.

  $ axicatavrisp -v -axicat /dev/ttyUSB0 -device ATmega164A -verify -flash
    boot.hex -flash main.hex

    The program will:
    1. Check if the connected device is an ATmega164A.
    2. Load two IHEX file into the data buffer for flash memory.
    3. Verify the data in flash memory.
```

```
$ axicatavrisp -v -axicat /dev/ttyUSB0 -eeprom bytes 20h 0 10h 3Ah 1010b
  -write -noerase

  The program will:
  1. Store the four bytes at address 20h in the data buffer for EEPROM.
  2. Write the specified data to EEPROM.
  3. Verify the written data in EEPROM.

$ axicatavrisp -axicat /dev/ttyUSB0 -device atmega164a -fuse eesave 0 -write
  -noerase

  Enable EESAVE, meaning EEPROM won't be erased during a subsequent chip
  erase.

$ axicatavrisp -axicat /dev/ttyUSB0 -device atmega164a -fuse bodlevel 110b
  -write -noerase

  Set brown-out detection level to 110b in an ATmega164A.

$ axicatavrisp -v -device ATmega164A -flash boot.hex -flash main.hex -dump
  buf flash

  Dump the flash pages that would be written to an ATmega164A.

  The program will:
  1. Load two IHEX files into the data buffer for flash memory.
  2. Dump the data buffer for flash memory.

$ axicatavrisp -v -axicat /dev/ttyUSB0 -flash dev all -save flash fw.hex

  Read the entire flash memory and save to Intel HEX file.

$ axicatavrisp -v -axicat /dev/ttyUSB0 -eeprom dev 0020h to 003Fh -save
  eeprom chunk.eep

  Read a section of EEPROM and save to Intel HEX file.

$ axicatavrisp -info devices

  Print a list of all supported devices.
```

Document History

2016-09-03 Peter S'heeren, Axiris

  * First release.

2016-11-06 Peter S'heeren, Axiris

  * Second release.