

Fuse Settings and Boot Loaders for ATmega 328 based Arduinos (Uno, Nano and Pro Mini) or Stand Alone 328P Chip

By Mike Kitchen (based on an original article by Martyn Currey)
<http://www.martyncurrey.com/arduino-atmega-328p-fuse-settings/>

While the below is intended for Arduinos based on the AVR ATmega 328, it is also relevant to most other 8 bit ATmega chips.

The ATmega 328 has four fuse bytes that configure how it operates:

- low byte fuses
- high byte fuses
- extended fuses
- lock bits fuses

Normally when you are programming an Arduino, you aren't even aware of the fuses existence as the Arduino IDE takes care of setting them if required. The fuses set things like:

- Select the clock speed, clock type and clock source (eg 8 or 16MHz, XTAL or resonator, external or internal RC clock etc)
- Set the "brown out voltage" (ie the minimum voltage required for the chip to work before it is automatically reset).
- Set whether or not a boot loader is used and if so what size the boot loader is.
- Disable the reset pin.
- Disable serial programming with an ISP (inline serial programmer such as AVR ISP, USBasp etc).
- Stop the EEPROM data from being erased when uploading a new sketch or writing a new boot loader.

The lock bits can be used to restrict read/write access to the program memory.

- The boot section and the application section have their own lock bits and access for each area can be controlled separately.
- The lock bits could be used to stop code from being read from the ATmega 328, (if you don't want people copying your program).
- It is however recommended not to play with the lock bits as it is easy to "brick" your Arduino (make it unusable) and you'll need an HV programmer to recover.
- If using the Arduino environment the licencing agreement usually requires you to release your software as open source as you'll be using the Arduino run-time functions in your program.
- However it is fairly straight forward to understand and use the fuse settings especially if one of the "on-line" fuse calculators is used (more on this later).

One important point is that the ATmega 328 (and other AVR ATmega chips) use a 0 (zero) to set a fuse and a 1 (one) for not set/not programmed. This seems counter intuitive but dates from the time processors were “write once only” where every bit was a 1 (ie fuse not blown) and a 0 when blown (fuse blown open circuit).

Each fuse byte obviously has 8 bits and each bit is a separate setting or flag (not all bits are actually used though). When programming the fuses you can use binary notation or more commonly hexadecimal notation. For 8 bits (1 byte) we can use B11111111 or 0xFF both are the same value.

Low Byte Fuses

The low byte fuse deals with the clock source, how fast the chip will run, and how long it waits at start-up for the oscillator to start and stabilise.

Bit	Name	Description
7	CKDIV8	When set, divides the clock speed by 8.
6	CKOUT	When set, the clock pulse is output on PB0 (pin 14)
5	SUT1	Sets start up delay time
4	SUT0	
3	CKSEL3	Sets the Clock Source
2	CKSEL2	
1	CKSEL1	
0	CKSEL0	

The ATmega chips can be run at different speeds or frequencies and the frequency is determined by the clock source that is used. The clock source is set by using the CKSEL fuse bits.

CKSEL (Clock Sources / Clock Selection)

The clock signal can come from an internal RC oscillator, an external crystal/resonator, or an external signal (often the clock from another processor, to keep both in sync). Arduinos use an external 8MHz (3.3V) or 16MHz (5V) resonator (not a XTAL) but can be configured to use the internal RC oscillator.

The different options for CKSEL are:

Device Clocking Option	CKSEL3 CKSEL2 CKSEL1 CKSEL0
Low Power Crystal Oscillator	1111 - 1000
Full Swing Crystal Oscillator	0111 - 0110
Low Frequency Crystal Oscillator	0101 - 0100
Internal 128kHz RC Oscillator	0011
Calibrated Internal RC Oscillator	0010
External Clock	0000
Reserved / Not used	0001

Arduinos normally use a low power crystal oscillator.

The ATmega has 2 built in oscillators, a 128MHz RC oscillator and a calibrated RC oscillator.

The external clock option allows the chip to use an external square wave clock signal. This is used when you have a circuit with its own clock that you want to sync the ATmega with or if you want to use a separate clock chip. The external clock signal needs to be connected to the CLOCK-IN pin

Crystal Oscillator Options

Frequency Range (MHz)	CKSEL3 CKSEL2 CKSEL1
0.4 - 0.9	100 (This option should only be used with ceramic resonators)
0.9 - 3.0	101
3.0 - 8.0	110
8.0 - 16.0	111 (This is the value used on Arduinos)

CKSEL3=1 CKSEL2=1 CKSEL1=1 CKSEL0=1 selects a crystal/resonator anywhere from 8MHz to 16MHz and is the normal setting for Arduinos.

If you want to use a slower crystal, for example 6MHz, then you would use CKSEL3=1 CKSEL2=1 CKSEL1=0 CKSEL0=1 (the 3.0 to 8.0 range).

To use the internal RC oscillator at 8MHz, the settings are CKSEL3=0 CKSEL2=0 CKSEL1=1 CKSEL0=0

SUT1/SUT0 (Start-Up Time)

Crystals and ceramic resonators require sufficient voltage to operate correctly. When you start the ATmega chip it can take a brief period of time for the voltage to get to its maximum value. While the voltage is rising the clock source may not be working at the correct speed or frequency. To allow the clock to stabilise a start-up delay can be set.

CKSEL0 is used together with SUT1 and SUT0 to set the start-up delay time.

Oscillator Source / Power Conditions	Start-up Time from Power-down and Power-save	Additional Delay from Reset at 5V	CKSEL0	SUT1 SUT0
Ceramic resonator, fast rising power	258 CK	14CK + 4.1ms	0	00
Ceramic resonator, slowly rising power	258 CK	14CK + 65ms	0	01
Ceramic resonator, BOD enabled	1K CK	14CK	0	10
Ceramic resonator, fast rising power	1K CK	14CK + 4.1ms	0	11
Ceramic resonator, slowly rising power	1K CK	14CK + 65ms	1	00
Crystal Oscillator, BOD enabled	16K CK	14CK	1	01
Crystal Oscillator, fast rising power	16K CK	14CK + 4.1ms	1	10
Crystal Oscillator, slowly rising power	16K CK	14CK + 65ms	1	11

BOD is Brown Out Detection and discussed later.

The Arduino uses the maximum start-up delay of $14CK + 65ms$ ($CKSEL0=1$, $SUT1=1$, $SUT0=1$) which are also the default settings for a new ATmega 328/328P chip.

CHDIV8 (Clock Divide)

ATmega 328/328P chips have a built in RC oscillator which has an 8.0MHz frequency. New chips are shipped with this set as the clock source and the CKDIV8 fuse active, resulting in a 1.0MHz system clock. The start-up time is set to maximum and time-out period enabled. ($CKSEL = "0010"$, $SUT = "10"$, $CKDIV8 = "0"$). This setting is used so that all users can make their desired clock source setting using any available programming interface.

CKDIV8 should be used if the selected clock source has a higher frequency than the maximum frequency of the ATmega chip (at a given supply voltage).

The ATmega chips can be used at very low voltages, however, the lower the voltage the slower they need to work at. The CHDIV8 can be used to set a slow clock speed to match a low voltage.

CKOUT (Clock Out)

The clock signal can be routed to PB0. This is useful if you need the clock signal to drive other circuits. This works for all clock sources and the default setting for new chips is $CKOUT=1$ (not set).

High Byte Fuses

The high byte fuse has several different settings. The ones of normal interest to most hobbyist are the watchdog timer, preserving or erasing EEPROM when a program is uploaded and the boot loader attribute settings.

Bit	Name	Description
7	RSTDISBL	External reset disable
6	DWEN	debugWIRE enable
5	SPIEN	Enable Serial programming and Data Downloading
4	WDTON	Watchdog Timer Always On
3	EESAVE	Preserve eeprom memory through chip erase
2	BOOTSZ1	Sets the boot loader memory size
1	BOOTSZ0	
0	BOTRST	Select the reset vector

RSTDISBL (External reset disable)

The Arduino reset pin (PC6) will reset the chip, by holding it low (to ground). This is how the switch on the Arduino works. When you press the reset button it connects PC6 to ground and resets the chip.

PC6 can also be used as a regular IO pin but this means disabling the reset function which in turns means the chip can no longer be programmed with the normal USB interface or by an ISP (In-System Programmer)) as both require the reset (USB needs to run the boot loader see further below and ISP also needs the boot loader). This is one of the options it is better not to change.

RSTDISBL is implemented so that you can gain an extra general purpose I/O pin and for security reasons to stop the chip being reprogrammed or the firmware being downloaded (as reset is no longer available).

When RSTDISBL is programmed the start-up time (SUT1 SUT0) is increased to $14CK + 4.1ms$ to ensure the programming mode can be entered.

By default, the setting is RSTDISBL=1 (not set).

DWEN (debugWIRE enable)

ATmega chips have built in hardware for use with debugging tools (typically with JTAG interface) which are by default turned off. The DWEN fuse is used to turn them on.

DWEN uses the same pin as reset (PC6) and when DWEN is enabled (and the lock bits are not set) the reset pin becomes a communication pin and normal reset no longer works. For example, if you enable DWEN on an Arduino the reset button no longer works.

To use the on-chip debugging you need a compatible programmer such as the AVR Dragon. There is a good article to get you started at:

<http://www.hilltop-cottage.info/blogs/adam/debugging-arduino-using-debugwire-atmel-studio-and-an-avr-dragon/>

This is one of the fuses you should take care with. If you enable DWEN and also enable the lock bits you can no longer program the chip in the normal way.

If you are reading this guide to understand how to use the AVR fuses, then it's best to leave DWEN not set (default). After all, reset is very useful. If you understand how to use DWEN then you probably don't need to read this guide!

SPIEN (Enable Serial programming and Data Downloading)

The normal method of programming an Arduino is by the USB interface that connects the USB to a USB to RS232 serial interface converter (Uno's use an ATmega 8U, "official" Nano's use a FTDI chip and cheaper far eastern Nano's use a CH340 chip). This is connected to the ATmega328 on-board UART and when a program is uploaded it resets the Arduino that then runs the boot loader (a small program that runs at start-up to check if a new program needs to be loaded). The boot loader then gets the ATmega to write the incoming data to the flash storage. This is how the majority of hobbyists program an Arduino.

The other method of programming the ATmega chips is via an ISP (in-system programmer) via the SPI interface using the SCK (clock), MOSI (input) and MISO (output) pins. These pins together with power and reset are brought out on a dedicated 6 pin header for convenience. If you disable serial programming then you can no longer use the SPI to program the chip. You can also disable serial programming using the lock bits.

For normal use and development leave SPIEN enabled, however, if you have a device that is final and no further programming is required then you can use this option to stop people accessing the chip through serial communication.

The default setting is SPIEN enabled, SPIEN=0.

Just remember that the RSTDISBL, SPIEN and DWEN fuses have the potential to brick the ATmega chip, or at least make the chip very, very difficult to use again (a HV programmer is required). For general use, and especially if you are just starting out, you should not change these fuse settings. You should also double check that you are not accidentally changing them when you re-write the other fuse settings.

WDTON (Watchdog Timer Always On)

The watchdog is basically a timer that will cause the chip to reset if it does not receive an OK signal at regular specific times. This can be useful when you have a critical system that must not be allowed to permanently lock up.

It can also be configured to run an interrupt before resetting and is often referred to as the "last wish interrupt" as code can be run to reset attached equipment, write an error code to the UART or save information in the EEPROM. This is useful for development code as the interrupt can communicate the program address which was running when it locked up, plus other information such as variable values etc.

When the watchdog timer is enabled, should a sketch crash or freeze then the timer will time out and reset the chip causing a restart similar to pressing the reset button a running Arduino.

The Watchdog Timer can be activated in software so the fuse settings don't really need to be used. By default, the watchdog timer is off, WDTON=1.

There is a nice mini guide on the Watchdog timer by za_nic on the Arduino forum at:

<http://forum.arduino.cc/index.php/topic,63651.0.html>

Paul Martinsen has a guide on how to use the watchdog timer to detect lockups at <https://www.megunolink.com/articles/how-to-detect-lockups-using-the-arduino-watchdog/>

EESAVE (Preserve EEPROM memory)

When the ATmega chip is programmed the memory is erased before the new code is uploaded. Under normal circumstances the EEPROM memory is also erased as well as the program memory. The EESAVE fuse can be used to tell the chip not to erase the EEPROM. This is useful when you want to upgrade code but keep user settings that are stored in EEPROM.

The default value is EESAVE=1, not set and EEPROM memory is erased during the chip erase cycle when programming. If your sketch/program is being used to store parameters that you don't want erased then set EESAVE=0. Also if your Arduino is being used for development and is regularly reprogrammed or updated then you are decreasing the EEPROM life, it has a finite number of write/erase cycles.

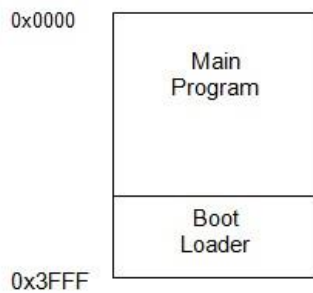
BOOTSZ1 & BOOTSZ0 (Boot loader Size)

The ATmega chips have the ability to use a boot loader, this is a small program that runs when the chip is first started or when it is reset. Boot loaders are normally used to initialize the device and check for external communication. The Arduino uses a boot loader to talk to a connected computer to see if there is a new program to be uploaded. This is the reason the Arduino resets just before you upload new code, the reset runs the boot loader and the boot loader checks to see if you have new code.

The boot loader is stored in program memory, the same memory used for the user application and since the boot loader can be different sizes you can tell the ATmega chip how much space to reserve. The older Arduino boot loader is 2 kilobytes (KB) and is still used by the Nano and Pro Mini, but the Uno is supplied with the newer Optiboot and is only 0.5KB (512 bytes). The Optiboot boot loader works perfectly on Nano's and Pro Mini's, it's a mystery why they are not supplied with the Optiboot boot loader as standard.

If using Optiboot and setting the boot loader size accordingly you gain an extra 1.5KB for your own program. An extra 1.5KB flash memory can be a lot of for sketches/programs with LCD or OLED displays.

Program Memory



The boot loader is stored at the top of the program memory

If you have a boot loader then **BOOTSZ** has to be used in conjunction with **BOOTRST**. **BOOTRST** tells the ATmega chip the memory address where the boot loader starts. If **BOOTRST** is configured correctly, then the user application can use the whole of the memory regardless of the **BOOTSZ** settings.

Boot Size Configuration for the 328/328P

BOOTSZ1	BOOTSZ0	Boot Size	Pages	Application Start	Application End	Boot Loader Start / Boot Reset Address
1	1	256 words	4	0x0000	0x3EFF	0x3F00
1	0	512 words	8	0x0000	0x3DFF	0x3E00
0	1	1024 words	16	0x0000	0x3BFF	0x3C00
0	0	2048 words	32	0x0000	0x37FF	0x3800

Note that the above sizes (taken from the ATmega 328 data sheet) are in words and 256 words is 512 bytes (a word is 2 bytes).

The default for new chips is **BOOTSZ1=0** **BOOTSZ0=0** which means they have 4KB reserved for a boot loader.

BOOTRST Select reset vector

If you have a boot loader then you need to inform the MCU and this is done by using the **BOOTRST** fuse setting. When the **BOOTRST** Fuse is set, when the device resets (or powers up) it will jump to the Boot Loader address. If not set, the chip will jump to the program start address at 0x0000.

The default value is **BOOTRST = 1** (not set).

If **BOOTRST** is not set (no boot loader) then the **BOOTSZ** fuses are ignored and the full 32K of program space can be used for the users program.

Extended Fuse Bits

The extended fuses are only used to set the brown-out detection level (BOD).

The ATmega chips can become unstable or unreliable when used with insufficient voltage. For example, the 328/328P can run safely at 16MHz if it has at least 4V. Anything below 4V means the chip is likely to misbehave. This would be a problem if the device were being used with sensors to take measurements or relied on accurate timings.

To ensure the chip has sufficient input voltage a minimum voltage level can be set. If the voltage falls below this level then the chip will reset itself. This is called the brown-out detector level. Basically, if the supplied voltage is below the BOD the chip keeps reset low.

Bit	Name	Description
7		Not used / reserved
6		Not used / reserved
5		Not used / reserved
4		Not used / reserved
3		Not used / reserved
2	BODLEVEL2	Sets the Brown-out detector level
1	BODLEVEL1	
0	BODLEVEL0	

BODLEVEL2, 1, 0	Typical Voltage
111	BOD disabled
110	1.8
101	2.7
100	4.3
011	Reserved / Not Used
010	
001	
000	

The default values for new chips are; BODLEVEL2=1 (not set), BODLEVEL1=1 (not set), BODLEVEL0=1 (not set) ie the BOD is disabled.

However on Arduino's the BOD is set to 2.7V. This is fine if it's a 3.3V Arduino, but the most common Arduino's are the 5V (5 volts) versions. AVR provided the 4.3V level for 328/328P running at 16MHz and 5V supply and 2.7V when running at 8MHz and 3.3V supply (ie BOD resets the Arduino 0.3V above where the chip may start to get unreliable). It is not clear why the Arduino developers chose to set the BOD at 2.7V for 16MHz, 5V versions.

Default Fuse Settings

New ATmega 328P Chip.

New 328/328P chips generally have the following fuse settings:

Low fuse = 0x62 (B01100010)

High fuse = 0xD9 (B11011001)

Extended fuse = 0xFF (B11111111)

Low Byte Fuse

Bit	Name	Description	Value		
7	CKDIV8	Divide clock by 8	0	Set	Divide clock by 8
6	CKOUT	Output clock on PB0	1	Not set	
5	SUT1	Sets start up delay time	1	Not set	14CK + 65m
4	SUT0		0	Set	
3	CKSEL3	Clock Source	0	Set	Internal clock @ 8MHz
2	CKSEL2		0	Set	
1	CKSEL1		1	Not set	
0	CKSEL0		0	Set	

High Byte Fuse

Bit	Name	Description	Value		
7	RSTDISBL	External reset disable	1	Not set	
6	DWEN	debugWIRE enable	1	Not set	
5	SPIEN	Enable Serial programming	0	Set	Allow serial programming
4	WDTON	Watchdog Timer Always On	1	Not set	
3	EESAVE	Preserve eeprom	1	Not set	Erase eeprom memory when the chip is programmed
2	BOOTSZ1	boot loader memory size	0	Set	Boot loader size
1	BOOTSZ0		0	Set	
0	BOOTRST	Boot loader reset vector	1	Not set	

Extended Fuse

Bit	Name	Description	Value		
7		Not used	1	Not set	
6		Not used	1	Not set	
5		Not used	1	Not set	
4		Not used	1	Not set	
3		Not used	1	Not set	
2	BODLEVEL2	Brown-out detector level	1	Not set	BOD level disabled
1	BODLEVEL1		1	Not set	
0	BODLEVEL0		1	Not set	

Default Fuse Settings for Arduino Duemilanove, Nano and Pro Mini with ATmega328/328P and original 2K Boot Loader

Low fuse = 0xFF (B11111111)

High fuse = 0xDA (B11011010)

Extended fuse = 0xFD (B11111101)

Low Fuse

Bit	Name	Description	Value		
7	CKDIV8	Divide clock by 8	1	Not set	
6	CKOUT	Output clock on PB0	1	Not set	
5	SUT1	Sets start up delay time	1	Not set	14CK + 65m
4	SUT0		1	Not set	
3	CKSEL3	Clock Source	1	Not set	Low Power Crystal Oscillator. 8.0 - 16.0MHz
2	CKSEL2		1	Not set	
1	CKSEL1		1	Not set	
0	CKSEL0		1	Not set	

High Fuse

Bit	Name	Description	Value		
7	RSTDISBL	External reset disable	1	Not set	
6	DWEN	debugWIRE enable	1	Not set	
5	SPIEN	Enable Serial programming	0	Set	Allow serial programming
4	WDTON	Watchdog Timer Always On	1	Not set	
3	EESAVE	Preserve eeprom	1	Not set	Erase eeprom memory when the chip is programmed
2	BOOTSZ1	boot loader memory size	0	Set	2KB boot loader size
1	BOOTSZ0		1	Not set	
0	BOOTRST	Boot loader reset vector	0	Set	The Arduino has a boot loader so needs the reset vector

Extended Fuse

Bit	Name	Description	Value		
7		Not used	1	Not set	
6		Not used	1	Not set	
5		Not used	1	Not set	
4		Not used	1	Not set	
3		Not used	1	Not set	
2	BODLEVEL2	Brown-out detector level	1	Not set	BOD level = 2.7V
1	BODLEVEL1		0	Set	
0	BODLEVEL0		1	Not set	

Default Fuse Settings for Arduino Uno with ATmega328P and Optiboot Boot Loader

The only difference is the space allocated to the boot loader. High fuse = 0xDE (B11011110)

Bit	Name	Description	Value		
7	RSTDISBL	External reset disable	1	Not set	
6	DWEN	debugWIRE enable	1	Not set	
5	SPIEN	Enable Serial programming	0	Set	Allow serial programming
4	WDTON	Watchdog Timer Always On	1	Not set	
3	EESAVE	Preserve eeprom	1	Not set	
2	BOOTSZ1	boot loader memory size	1	Not set	0.5KB boot loader size
1	BOOTSZ0		1	Not set	
0	BOOTRST	Boot loader reset vector	0	Set	The Arduino has a boot loader so needs the reset vector

For a comprehensive list of the default fuse settings for the various Arduinos have a look at Coding with Cody's [Arduino Default Fuse Settings](http://www.codingwithcody.com/2011/04/arduino-default-fuse-settings/) page.
<http://www.codingwithcody.com/2011/04/arduino-default-fuse-settings/>

Lock Bits

Lock bits can be used to restrict read/write access to the program memory. The boot section and the application section have their own lock bits and access for each area can be controlled separately.

You can brick the ATmega chip if you enable the lock bits and you should not change them.

Fuse Bits are not Programmed when a Program is Uploaded

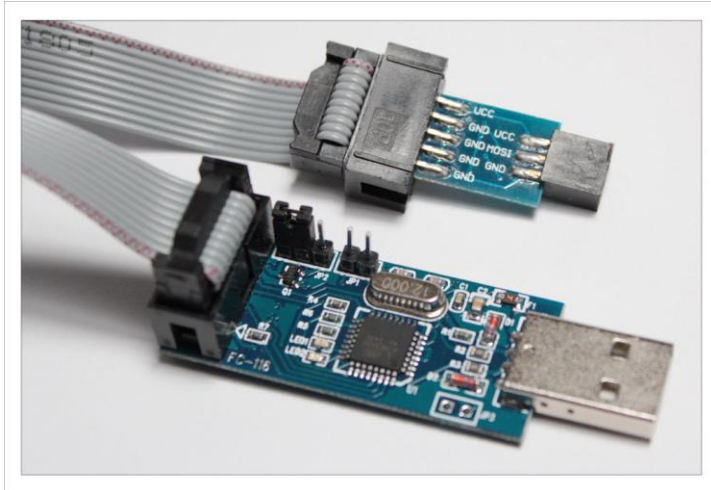
When a program is uploaded by the Arduino IDE, it does not change any of the fuse bits (except the lock bits that we are not interested in).

When you program using an ISP you overwrite everything in the chips flash memory, even the boot loader, but you don't alter the fuses. So when the ATmega is reset, it will start running at the boot loader reset vector, not at 0x0000. Normally the sketch/program will run OK as the upper memory where the boot loader would normally be will just be all NOP (no operations) and it will run through these rapidly and wrap around to 0x0000 and then run the program normally.

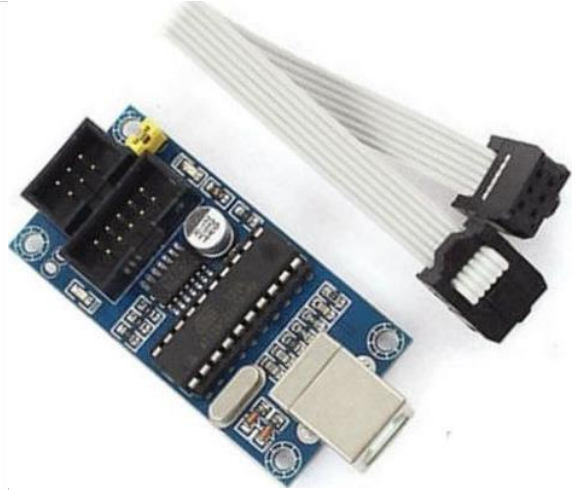
But it does mean that if you want to use the full 32K flash program memory for a user program you need the BOOTRST fuse not set. Actually you also need to tell the Arduino IDE the maximum upload size as well, but that is covered below in editing the boards.txt file.

How to Actually Write Fuse Bits

The fuse bits of an Arduino can be programmed by two methods, by the Arduino IDE burning a boot loader or using the AVRdude program run from a command prompt. Both methods require using an ISP (inline serial programmer) such as AVR ISP, USBtinyISP, USBasp or another Arduino programmed as an ISP. We will only cover the burning a boot loader method in this guide.



USBasp programmer and cable



USBtinyISP

When the Arduino IDE burns a boot loader fuses are set as part of the process. The fuse information together with other information is stored in the boards.txt file and this file has a section for each board type supported.

By editing the boards.txt file (usually in the Arduino installation folder – see later) you can use any fuse values you desire. Rather than altering existing board type it is best to add new board types so the originals are still available if needed in the future. The preferred method according to the Arduino community is to add variants, but this is slightly more complicated method than just editing the boards.txt file, which we will detail below.

Where is the boards.txt File Located?

The boards.txt file will usually be located in a sub-directory of where the Arduino IDE is installed. The location of this depends on a number of factors, such as the operating system (Windows, Mac or Linux) and which version of the operating system (Win 10, 8, 7, Vista or XP for example) and if the Arduino “files and libraries” have been updated by the “Boards Manager”.

On my Windows 10 Pro 64 bit system it is located at: C:\Program Files (x86)\Arduino\hardware\arduino\avr

And on my Windows 10 laptop (that has been updated by the “Boards Manager”):
C:\Users\{username}\AppData\Local\Arduino15\packages\arduino\hardware\avr\{version}\boards.txt

If the “Boards Manager” has been used to update boards and libraries, you may well have a number of boards.txt files, but only one will be active.

The fool-proof method to locate the active boards.txt file is to do the following in the Arduino IDE:

- Tools > Board > Arduino/Genuino Uno
- File > Examples > EEPROM > eeprom_clear
- Sketch > Show sketch folder - this will put you in
...\libraries\EEPROM\eeprom_clear folder so you only need to go back up a few levels to get to the boards.txt file

Editing the boards.txt File

- It goes without saying that before starting, make a backup of the boards.txt file (to something like boards.txt.old) so if something goes wrong, you can restore the original.
- Ensure that the Arduino IDE is closed before editing boards.txt, then restart it after editing.
- If using Windows you can use either WordPad or NotePad, just remember to save as a .txt (text) file. If using Linux or a Mac, use your favourite text editor.
- With Windows 10 (and maybe Windows 8?) and Linux you will need administrator rights to open and edit the boards.txt file.
- With Windows 10, this means opening WordPad or NotePad with admin rights, not just being logged in as a user with admin rights. To do this, go to the windows/start button, scroll down to Windows Accessories, WordPad (or NotePad) and right hand click the icon. Chose “More” at the top and select “Run as Administrator”. Now you’ll need to navigate to where you have found the active boards.txt file.

What Information is in the boards.txt file

If you open the boards.txt file you will find a series of entries, one for each of supported Arduino boards. The information for the Uno is on the page below:

- The # symbol is a “comment” so anything after a # is ignored
- uno.name= gives the description in the Arduino IDE tools/board type
- uno.vid and uno.pid are the board manufacturer identifier and are not strictly needed
- uno.upload tells the IDE/compiler what tool to use to upload the program (avrdude), the protocol, the maximum program size, the data block size, and the upload speed (this needs to match the boot loader)
- uno.bootloader has the upload program (avrdude), the fuse information, what boot loader to use and its location
- uno.build tells the compiler which mcu (microcontroller) to compile for, its clock speed and what Arduino board it is (for I/O pin names etc)

#####

uno.name=Arduino/Genuino Uno

uno.vid.0=0x2341
uno.pid.0=0x0043
uno.vid.1=0x2341
uno.pid.1=0x0001
uno.vid.2=0x2A03
uno.pid.2=0x0043
uno.vid.3=0x2341
uno.pid.3=0x0243

uno.upload.tool=avrdude
uno.upload.protocol=arduino
uno.upload.maximum_size=32256
uno.upload.maximum_data_size=2048
uno.upload.speed=115200

uno.bootloader.tool=avrdude
uno.bootloader.low_fuses=0xFF
uno.bootloader.high_fuses=0xDE
uno.bootloader.extended_fuses=0xFD
uno.bootloader.unlock_bits=0x3F
uno.bootloader.lock_bits=0x0F
uno.bootloader.file=optiboot/optiboot_atmega328.hex

uno.build.mcu=atmega328p
uno.build.f_cpu=16000000L
uno.build.board=AVR_UNO
uno.build.core=arduino
uno.build.variant=standard

#####

Example of a Nano with the Optiboot Bootloader and No Boot Loader

As an example adding additional variations of an Arduino to the boards.txt file, below are two examples: one for an Arduino Nano to use the Optiboot boot loader (that only takes 512 bytes instead of the original 2K boot loader) and an entry for a Nano to have no boot loader so the full 32K flash is available for user programs.

```
#####  
onano.name=Arduino Nano with opti
```

```
onano.upload.tool=avrdude  
onano.upload.protocol=arduino  
onano.upload.maximum_size=32256  
onano.upload.maximum_data_size=2048  
onano.upload.speed=115200
```

```
onano.bootloader.tool=avrdude  
onano.bootloader.low_fuses=0xFF  
onano.bootloader.high_fuses=0xD6  
onano.bootloader.extended_fuses=0xFC  
onano.bootloader.unlock_bits=0x3F  
onano.bootloader.lock_bits=0x0F  
onano.bootloader.file=optiboot/optiboot_atmega328.hex
```

```
onano.build.mcu=atmega328p  
onano.build.f_cpu=16000000L  
onano.build.board=AVR_NANO  
onano.build.core=arduino  
onano.build.variant=eightanaloginputs
```

```
#####  
nnano.name=Arduino Nano with no bootloader
```

```
nnano.upload.tool=avrdude  
nnano.upload.protocol=arduino  
nnano.upload.maximum_size=32768  
nnano.upload.maximum_data_size=2048  
nnano.upload.speed=115200
```

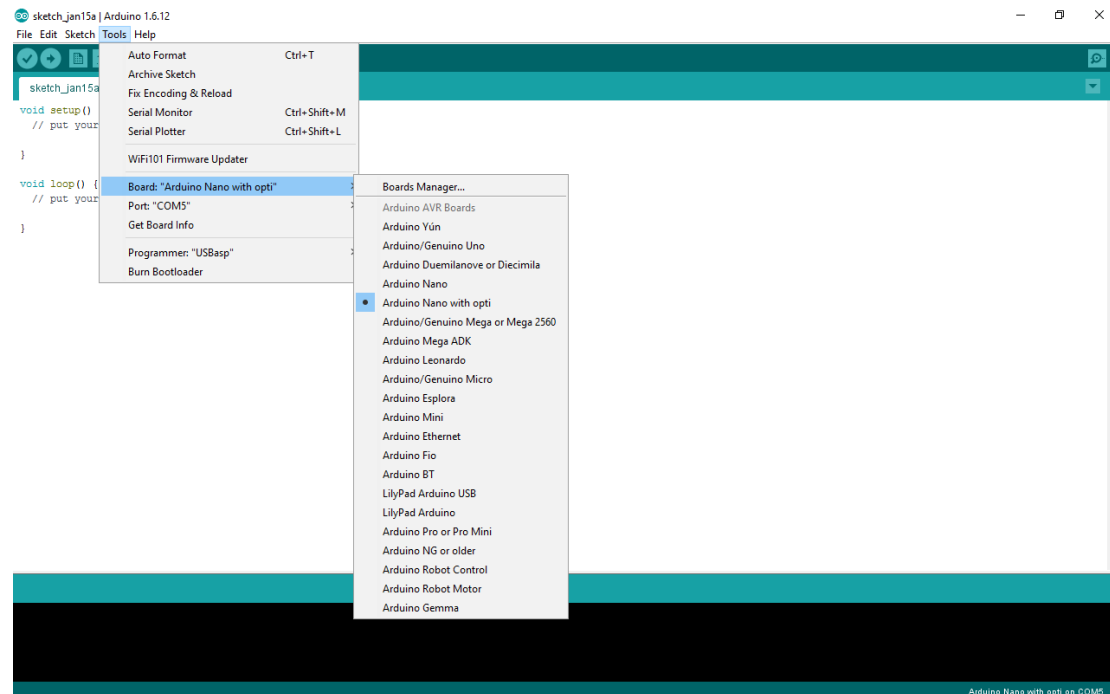
```
nnano.bootloader.tool=avrdude  
nnano.bootloader.low_fuses=0xFF  
nnano.bootloader.high_fuses=0xD7  
nnano.bootloader.extended_fuses=0xFC  
nnano.bootloader.unlock_bits=0x3F  
nnano.bootloader.lock_bits=0x0F  
nnano.bootloader.file=optiboot/optiboot_atmega328.hex
```

```
nnano.build.mcu=atmega328p  
nnano.build.f_cpu=16000000L  
nnano.build.board=AVR_NANO  
nnano.build.core=arduino  
nnano.build.variant=eightanaloginputs
```

```
#####
```

If the above is cut and pasted into the boards.txt file and saved (I usually place the entry just below the existing entry for the Nano, then the all the Nano options appear together), then the Arduino IDE is started, under Tools -> Boards, two new options should be shown:

“Arduino Nano with opti” and “Arduino Nano with no bootloader”



The settings are similar to the one for the Uno except the fuse settings are different and the last line has `nnano.build.variant=eightanaloginputs` so that all eight analogue inputs are available (the Nano uses a SMD ATmega328 that has 8 analogue inputs, the Uno uses a ATmega328P with only six analogue inputs).

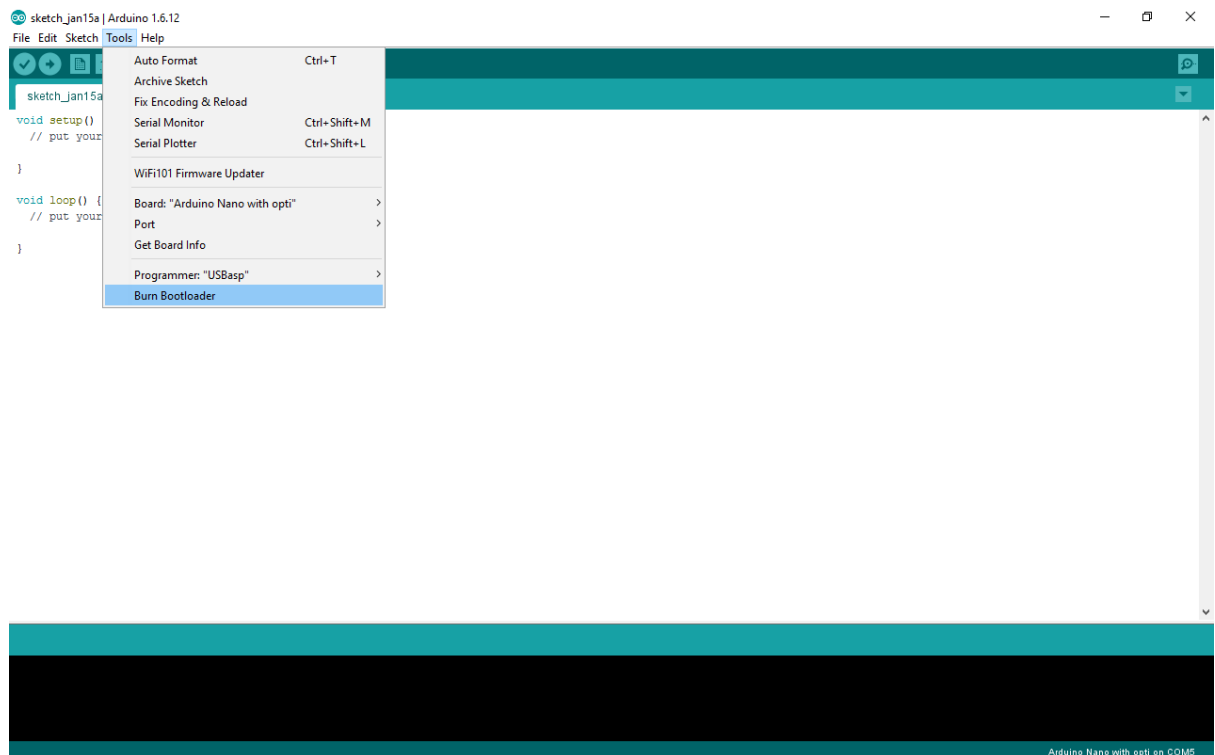
The fuse settings for the Nano with opti are modified to the Uno:

```
onano.bootloader.low_fuses=0xFF (Xtal with Max Start Time)
onano.bootloader.high_fuses=0xD6 (See below)
onano.bootloader.extended_fuses=0xFC (Brown Out Level 4.3V)
```

The High Fuses (0xD6 = 11010110) are set to preserve the EEPROM (ie not overwrite it when uploading a new program) and to have a 512 byte (256 word) boot loader.

Feel free to alter the fuses to suit your own needs eg some people will prefer the Brown Out Level to be set at 2.7V as with a standard 5V Arduino. Just replace 0xFC with 0xFD for the extended fuses.

Before the Arduino Nano with opti can be used, the boot loader must be burnt first. Select the “Board” (in this instance Arduino Nano with Opti). To burn the bootloader, connect the Arduino Nano to the PC via the ISP programmer, open the Arduino IDE, under Tools, select the correct Programmer and then select “Burn Bootloader”. After a short delay it should say completed. Your Nano now has the Opti bootloader installed. You have to select the Board Type as “Nano with opti” in the future when programming the Nano via USB. You will also notice when you compile a program it now shows “Maximum is 32,256 bytes” on the status line and the program will upload faster as it uses 115,200 baud, instead of 19,200 baud.



In a similar way to use an “Arduino Nano with no bootloader”, first we burn the boot loader! This may not seem intuitive at first, but don’t forget we are burning the boot loader to burn the configuration fuses. From the Tools, Board select “Arduino Nano with no bootloader” and follow the instructions above for the Arduino Nano with opti. You will have to use an ISP programmer for this board setting, but you will have the advantage of the full 32,768 byte of program space.

Programming a Stand-Alone ATmega328P Chip

Here is an entry for the boards.txt for use with bread board Arduinos without a boot loader. This has the low byte fuse at 0xFF, the high byte fuse at 0xDF, the extended fuse at 0xFD. Even though it shows a Opti boot loader the extended bits have BOOTRST reset (ie no boot loader) so the BOOTSZ0/1 will be ignored. The values for the lock bits were copied from the Arduino Uno entry.

```
atmegasal6.name=ATmega328P Stand Alone
atmegasal6.upload.protocol=stk500
atmegasal6.upload.maximum_size=32768
atmegasal6.upload.speed=115200
atmegasal6.upload.using=arduino:arduinoisp
atmegasal6.bootloader.low_fuses=0xff
atmegasal6.bootloader.high_fuses=0xdf
atmegasal6.bootloader.extended_fuses=0xfd
atmegasal6.bootloader.path=optiboot
atmegasal6.bootloader.file=optiboot_atmega328.hex
atmegasal6.bootloader.unlock_bits=0x3F
atmegasal6.bootloader.lock_bits=0x0F
atmegasal6.build.mcu=atmega328p
atmegasal6.build.f_cpu=1600000L
atmegasal6.build.core=arduino
atmegasal6.build.variant=arduino:standard
```

Useful links

Nick Gammon's site has a lot of really good information about building your own Arduino and uploading boot loaders:

[How to make an Arduino-compatible minimal board](#) (similar to this guide).

[Solving problems with uploading programs to your Arduino](#)

[ATmega boot loader programmer.](#)

A useful article about standalone 328P chips by Martyn Currey [Arduino on a breadboard](#) explains how to set up a stand-alone ATmega chip and [Using an Arduino Nano to program a ATmega328P chip](#) explains how to use an Arduino as an ISP programmer.

The ATmega 328P data sheet can be downloaded from the Amtel website at <http://www.atmel.com/Images/doc8161.pdf>

<http://eleccelerator.com> and <http://www.engbedded.com> both have a fuse calculator where you select what options you want and the page gives you the fuse settings to use. I prefer to manually calculate the fuse settings and then use the calculators as a check as I think it may be easy to make mistakes if only using the calculators, but this is only my personal opinion.

Nick Gammon also has a useful sketch that detects ATmega chip types. The sketch can be found at <http://www.gammon.com.au/forum/?id=11633>. The sketch displays the fuse settings, the lock bits and gives information about the boot loader if one is present. I found this sketch very useful when I first started programming chips directly.