\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Motivation \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Actually My final target is to gather enough knowledge so that I can do OTA for any chip. OTA means over the air firmware upgrade.

Target Story line OTA: mcu will work just like it does . it will receive commands from server via say any communication . then suddenly a programmer will want to upgrade the firmware of the mcu so it will update the ftp server with the latest firmware file [TODO: hex or bin] then it will command the mcu that you need to update yourself. The command will also have a timestamp at which mcu will update its firmware. So as mcu got this command it will switch to ftp mode via any communication that it is having with the server then it will receive the incoming bytes store them in an external flash then it will update a flag of its internal eeprom indicating it needs to upgrade its flash with the firmware residing at the external flash. Then mcu will wait until the timestamp comes. When the timestamp come mcu will soft reset itself [TODO: but how? May be it depends on different mcu architectures] after reset the custom bootloader will at first read the flag from the eeprom so it will now erase the application section code of the flash read bytes from the external flash via SPI or I2c like interface and write the application section of the flash. [AVR mcu FLASH has 2 sections at the top it has Application section and at the bottom portion it has the bootloader section (bootloader section can be extended using fuse bits but it will consume the application section memory blocks if extended) and it has 2 modes LPM (load program from memory) & SPM(self-programming mode) and you can disable SPM LPM capability of either of the section using fuse bits s] when BLS section will write bytes to the application section accidently it may write itself also that is where this safety feature can work . so this is how the BLS section code will write the new firmware to the application section . after that it will jump to the application code section [e.g. with statements like: asm ( "jmp 0x0000" ); <http://www.engineersgarage.com/embedded/avr-microcontroller-projects/How-To-Write-a-Simple-Bootloader-For-AVR-In-C-language> ]

So to attain the final target I have to first make a simple bootloader:

1st simple bootloader story Line: first I will make a bootloader which will read from the serial terminal and write on the flash and at the end it will jump.[for avr : I will write the bootloder using avr studio 7 then the compiled hex will be converted to .bin using python script <https://pythonhosted.org/IntelHex/part1-1.html#motivation> . then the bin file will be loaded to the BLS section using Atmel ICE like programmer because usbasp wont be able to do that , also fuse bit should be set so that BLS section allocation is sufficient to have the bin and also the boot reset vector must be enabled so that when mcu restarts it will start reading from the BLS section ]

After making the first simple bootloader my second target is to make a 2nd bootloader

2nd bootloader storyline: this will read eeprom flag and then read from an external flash using SPI or I2c and write the bytes to the application section of the flash and then jump to it

After the 1st and 2nd bootloader design I can develop the target story line OTA

I want to make the target story line for the atmega328P fisrt then want to make the exact solution for a chinese mcu [which probably don’t have BLS and Application section wise divide]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Bootloader study \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

So at first I am learning about bootloader . A perfect bootloader should be like this :

Bootloader Logic flow : [from embedded.com]

START

Reset

Host communication start? 

Timeout?

Wait indefinitely?

Wait for new app from host?

Application valid in flash?

Bootloader valid in flash?

no

Halt

No

yes

Go to application

Receive new application from host, install to flash by overwriting existing application

Host communication start?

no

yes

yes

yes

yes

no

yes

no

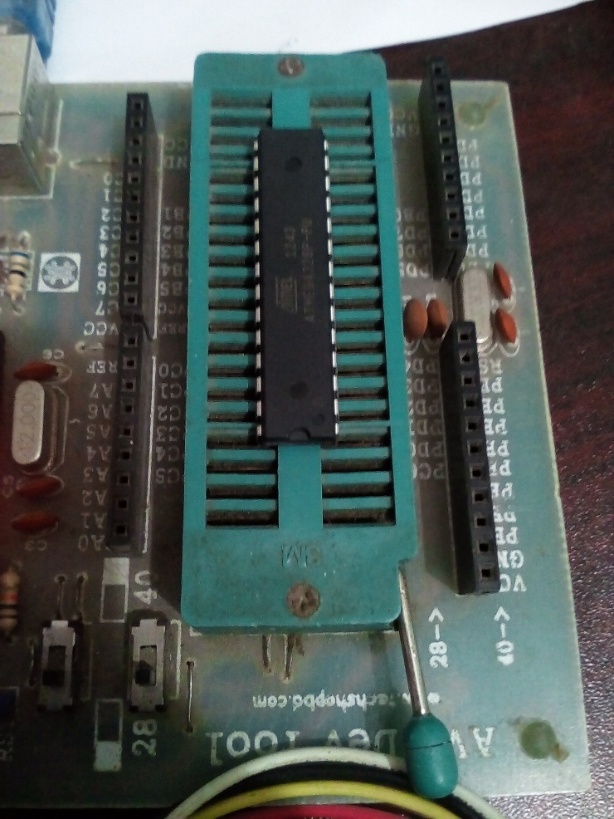
yes

in this study I am very interested in knowing how arduino bootloader works

So lets dig inside : I think by making a arduino uno from raw m328 will demystify about its bootloader

So Lets make a arduino uno using usbasp, arduino ide & extreme burner software. [for fuse settings explanation this site is useful <http://eleccelerator.com/fusecalc/fusecalc.php?chip=atmega328p&LOW=C1&HIGH=99&LOCKBIT=FF> ]

1. Take a raw atmega328p
2. Place atmega328 in usbasp like this



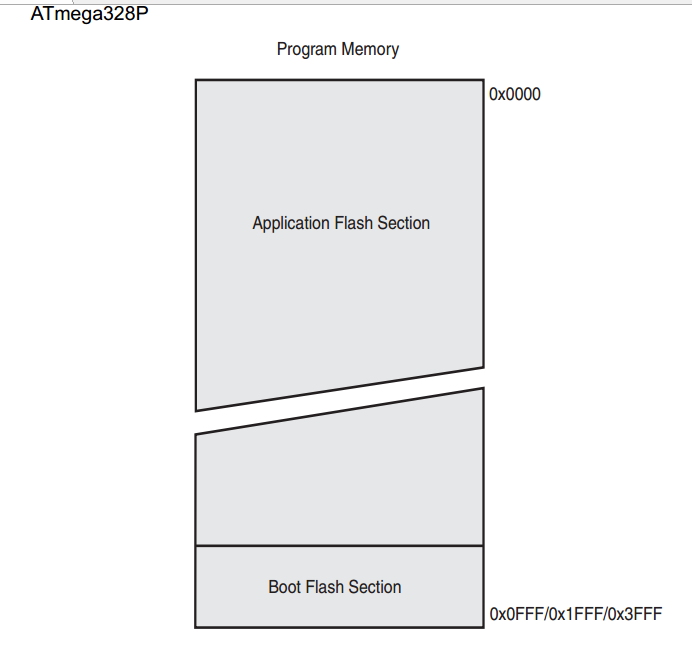
1. Read the raw factory shipped m328 fuse settings by extreme burner

LF=0x62 HF=0xD9 EF=0xFF Lock Fuse=0xFF

With the default fuse settings it means :

* Application protection mode 1: no lock on LPM and SPM in application section
* Bootloader protection mode 1: no lock on LPM and SPM in bootloader section
* Mode 1 : no memory lock features enabled
* Boot reset vector disabled means mcu will start from 0x0000

Oh I have not told you how flash of m328 looks inside ! it looks like this :



So you can see Application section starts from the top and BLS section at bottom and its size is configurable by fuse bits.

1. Now reading flash using extreme burner: everything is FFFF
2. Now Actually I am interested to see how is the flash of an arduino uno so I took a uno , uploaded a simple blinky then read its fuse bit which is

LF=0xFF HF=0xDA EF=0xFD Lock Fuse = 0xCF

It means:

* Application protection mode 1: no lock on LPM and SPM in application section
* Bootloader protection mode 3: LPM and SPM prohibited in bootloader section
* Mode 1 : no memory lock features enabled
* Boot reset vector enabled means mcu will start from 0x3C00

1. Now lets read the flash So I saw blinky codes at top of the flash and bootloader codes at the bottom
2. So I got my answer now going back to making arduino from raw m328 but how? These are the steps

* Open arduino IDE
* Choose usbasp as burner
* Click burn bootloader

1. So what I tried I tried the above steps to the same arduino from which I just read the flash and guess what Bootloader was burnt successfully :D
2. Now reading the flash by extremeburner to see inside of the flash and importantly there are codes from 007E00 which is the BLS section and From 000000 to 007DF0 is FFFF which is the Application section so bootloader is burnt the way it should be

|  |  |
| --- | --- |
|  |  |

1. Now check whether sketch will upload or not. it does upload and blinks :D

So we have got our arduino back

1. Now as it has blinky code inside lets checkout the flash inside examing I found bootloader section is at its portion and blinky code is at top starting from 000000

|  |  |
| --- | --- |
|  |  |

1. The other sections are FFFF as it should be
2. Now how did it happen ? Lets look at the ArduinoIDE output when bootloader was burnt . log is at a txt doc in / “UnobootoloaderBurnTom328ByArduinoIdeOutput”
3. At first device signature is checked
4. Chip is erased
5. Lock byte = 0x3F
6. EF = 0x05
7. HF = 0xDE
8. LF = 0xFF
9. Optiboot\_atmega328.hex burnt [C:\Program Files (x86)\Arduino\hardware\arduino\avr/bootloaders/optiboot/optiboot\_atmega328.hex:]
10. Lock Byte = 0x0F

So Now I got it . these upper steps should be done to make arduino from a raw m328p

Now lets go back to making a arduino from raw m328 . so did the upper things using extreme burner and guess what it worked and then I checked with a blinky upload Soooooooooo I have made a Arduino :D I am gonna be billionaire :P cheers :D

Now why it works ?

What ? why are you questioning this ? Lets slow down and think dude when you burnt the optiboot using extreme burner you never pointed out the burner to start writing from which address ! by default it always starts from writing the top application section so how it worked ?

So our question is how does optiboot\_atmega328.hex goes to the bootloader section ! as there are no option in avrdude [used by extreme burner and arduino IDE] which tells that write from this memory address to this memory address . To find out the answer lets load “optiboot\_ atmega328.hex” file in extreme burner and see what is inside:

|  |  |
| --- | --- |
|  |  |

So we got the answer :D

the Trick is that the optiboot hex is designed in a way that the total hex is the size of the flash of m328p and has bootloader side code at the lower part :D the upper parts are 000000 . that’s why if you burn it will burn from 000000 as usual but the lower portion will have the bootloader code which is the BLS actually and the fuse bit is such that boot section size matches the BLS portion code size in the hex [BLS section is 256 words] Also Note boot reset vector is enabled is that upon reset mcu will start from reading the BLS :D

So now if you upload a code the received bytes will be written from the 000000 address by the BLS section code which ran first when mcu started first. After writing it will jump to 000000 and your application code for example blinky will work. If your application code size is so much big that it exceeds the size of the application section I think [TODO test needed] bootloader wont be able to write your code at its own section as Bootloader protection mode 3 is set by the fuse bits :D

Haha It was brilliant from Arduino guyz :D

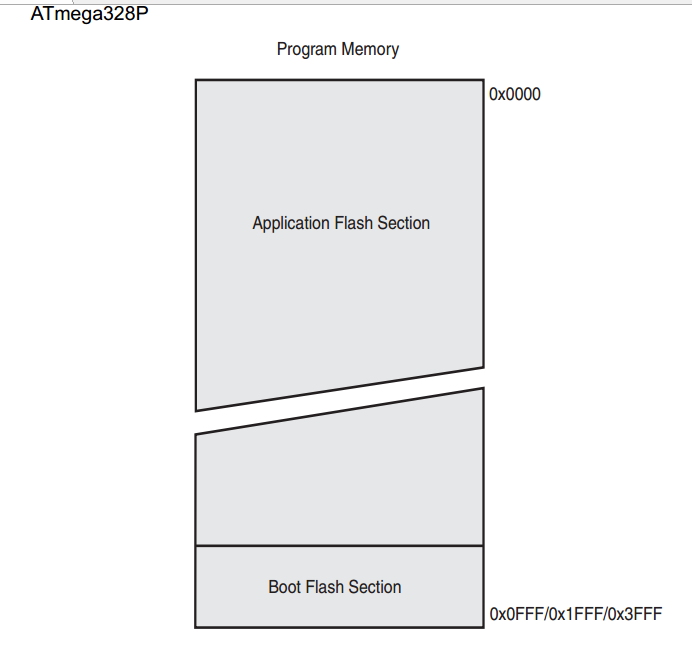
So I got it now if I want to make my bootloader how can I make a bootloader like arduino guys did ? well cant it be simple like say I will develop bootloader code using my fvrt IDE and then will burn the hex to the BLS section of the m328 …. Well its possible but for that I need to use atmel ICE, atmela studio 7 and a smart python script . well you can ask why Atmel ICE actually because usbasp wont be able to write from a specific address [as far as I know TODO: can avrdude do it ? ]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1st simple bootloader story Line \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

So I will try to develop the bootloader and burn it in the bootloader section only

Atmega 328p has a single flash and is divided into 2 sections

* Application program section
* Bootloader section



take a note of the following point from the datasheet:

* The size of the Boot Loader memory is configurable with fuses and the Boot Loader has two separate sets of Boot Lock bits which can be set independently.
* The Boot Loader program can use any available data interface and associated protocol to read code and write  
  (program) that code into the Flash memory
* Fuse bit is set as such so that The program code within the Boot Loader section has the capability to write into the entire Flash, but not its own.
* Any code which executes from the BLS (boot loader section) can use Self Programming Mode (SPM)

Using the SPM feature a code from the BLS section can read or write the entire flash memory including the BLS section where the code is running from.

* SPM mode has certain functions and as BLS section has SPM mode it can use these functions. for example:

1. boot\_page\_erase(address)
2. boot\_page\_fill(address,data)
3. boot\_page\_write(address)

* this functions are available at <avr/boot.h> library / api

[TODO: So application section cannot execute SPM instructions ? provided fuse bit allows it or if fuse bit do not permit can it ? ]

We are going develop using atmel AVR studio 7 IDE

1. Project created at /avr named “SPM\_test”. It’s a cpp project [TODO: delete SPM and start new project]
2. Don’t forget to add F\_CPU=8000000 at toolchain symbols. We will use the internal clock
3. Oh remember new operator needs some kind of cast to work in atmel studio that’s why singleton debug class did not work
4. Atmel ICE userguide is at / … remember you have to power m328, ICE won’t do that for you
5. SCK is pin 1 (red wire)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* try on Chinese mcu \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Finally I am choosing HT6015 mcu a Chinese mcu which don’t have any preconfigured bootloader section so it means I have the independence to choose my ranges for my bootloader. I will use keil IDE to to write program for the chip and will use ulink2 as the burner.

1. Does this Chinese mcu have any <avr/boot.h> like library?
2. Does it have any SPM kind feature? [self-programming mode] depends on 1

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*