NIELIT, Gorakhpur

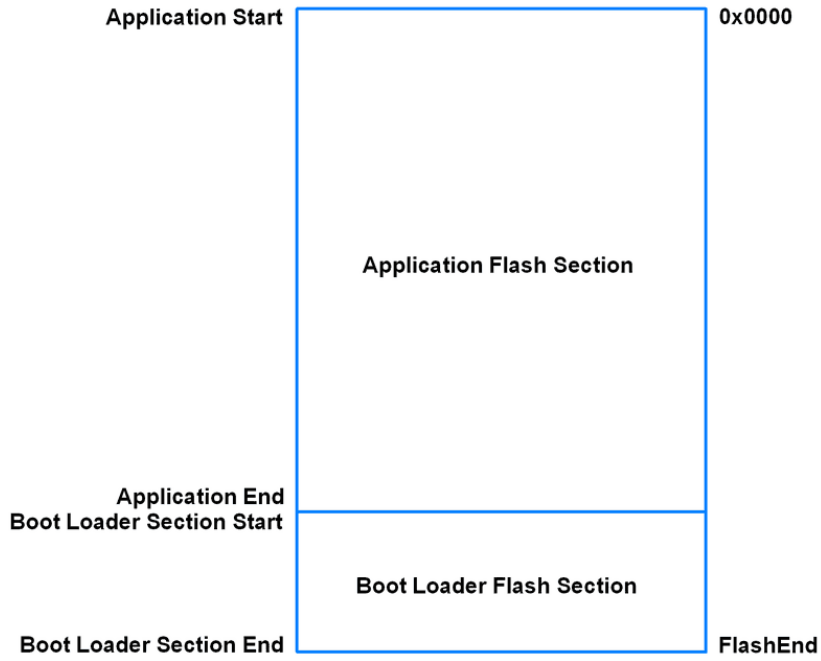
Course Name: A-level (1st Sem.) Subject: IoT

Topic: Developing Bootloader for Arduino Date: 24.04.2020

Introduction

Bootloader, is basically the initial piece of code which runs whenever any micro-controller is powered up or resets. It is similar to the concept of BIOS which run in our PC at the time we power up it. In case of BIOS, it waits for a user input for changing Boot options/settings. If it does not get any such inputs, it will start with the pre-installed OS.

A similar thing happens with Arduino bootloader. Whenever the Arduino is powered up or reset, it looks for external inputs (for uploading new program). If it receives no such inputs, it starts executing the program that was uploaded last.



Memory Sections

Arduino uses avr microcontrollers for their platforms which has program memory sections as shown in above figure. Boot Loader section is placed at the bottom of flash memory.

The bootloader program is written in the bootloader section, and the application program is written in the application section.

How Bootloader Starts

As we know that whenever a microcontroller is reset or is powered up, generally it starts program execution from the reset vector i.e. from 0x0000 program memory address.

We can change this reset vector address (0x0000) to bootloader section start address in case if we are using bootloader on the microcontroller. That means, **whenever the microcontroller is get reset/powered up, it starts program execution from the bootloader section**.

Arduino bootloaders do the same thing and execute the bootloader program when the microcontroller (used by Arduino) is reset/powered up i.e. the microcontrollers start execution of program from boot loader section's start address.

If we refer AVR microcontrollers (which are used for arduino) datasheet we can see that Boot Reset Fuse can be programmed so that Reset Vector is pointing to the Boot Flash start address after reset as shown in below figure.

Table 27-4. Boot Reset Fuse(1)

BOOTRST	Reset Address
1	Reset Vector = Application Reset (address 0x0000)
0	Reset Vector = Boot Loader Reset (see Table 27-7 on page 275)

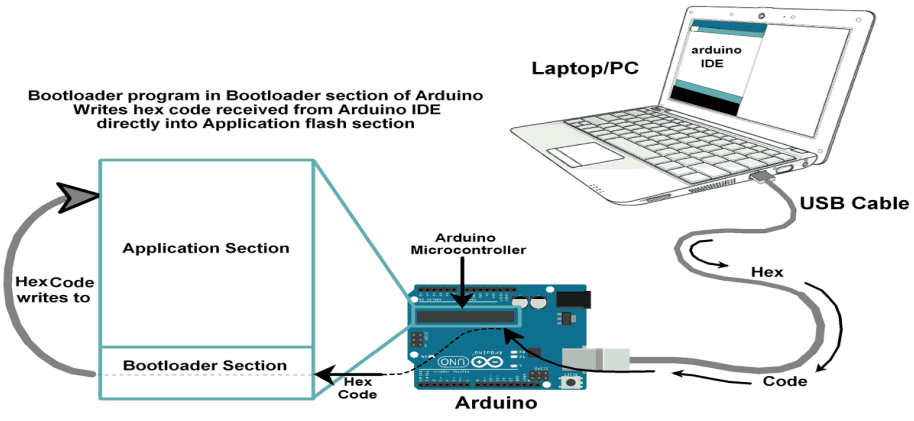
Note: 1. "1" means unprogrammed, "0" means programmed

Atmega328p datasheet (page no. 267)

Hence, we can set the reset vector to the start of the bootloader section on power up/reset.

Need of a Bootloader

Most of the times, bootloaders in microcontrollers are used to simplify the uploading of programs to the microcontrollers. They can also be used for initializing IO devices connected to the microcontrollers before they begin the main application program.Arduino bootloaders use the simple serial communication (UART) to download the hex file of program and write it in application section.



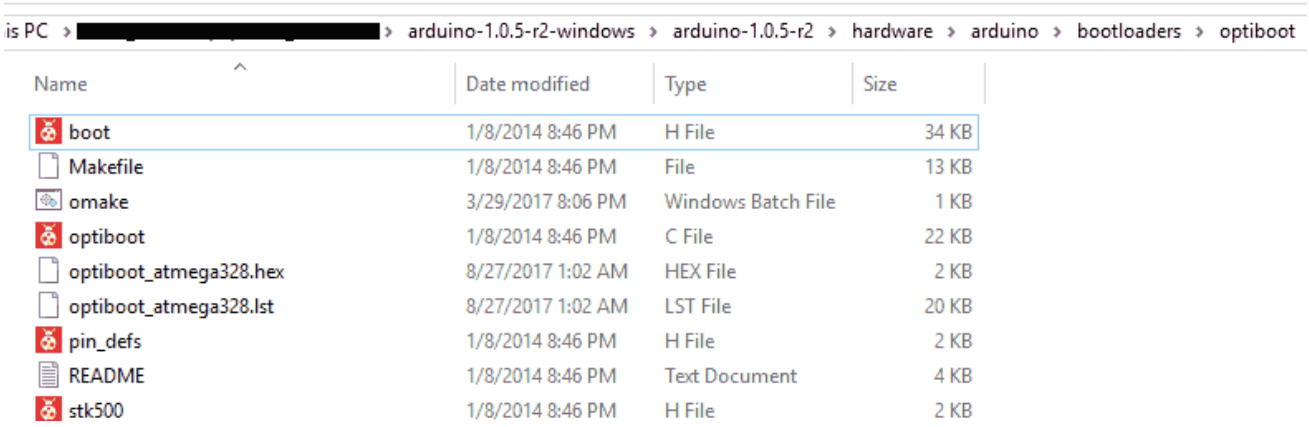
Inside the Bootloader

Now let's see in brief about how Arduino Bootloader is written and how it communicates with Arduino IDE while uploading programs.

We can find arduino bootloader program at

arduino-version\hardware\arduino\bootloaders\optiboot

As shown in below figure.



The boot headerfile (**boot.h**) is included from avr toolchain. This is modified/optimised version of avr toolchain boot header file(<avr/boot.h>). You can find avr boot header file at ***arduino-version\arduino-1.0.5-r2\hardware\tools\avr\avr\include\avr***

avr boot header file uses ***sts***(which requires two machine cycle) instruction to access SPM register whereas boot header file used in arduino bootloader uses ***out*** (which requires only one machine cycle) instruction to access SPM register. This important optimisation is already mentioned in avr toolchain boot header file for smaller devices.

Boot header file contains the function related to write/read flash memory (in manner of page by page). Also, it contains the function for writing/reading fuse, lock, and signature bits.

stk500 header file (**stk500.h**) contains the STK500 commands which are used for reliable handshaking communication in between arduino and avrdude program while uploading hex file.

Pin definition header file (**pin_defs.h**) contains port definition for LED (arduino on-board LED) which is used as status LED blink while flashing the arduino.

optiboot.c file contains the main program flow of bootloader (i.e. receiving hex serially and writing it to program memory). Other files (boot.h, pin_defs.h, stk500.h) are included in optiboot.c file.

Optiboot program starts with **MCUSR** (MCU Status Register) status register which provides information about the reset source that caused reset. If reset source is not external (by pulling

reset pin low), then it will directly start the application program. As shown in below figure, it will call appStart() function from where it jumps to direct 0x0000 reset address.

```
// Adaboot no-wait mod
ch = MCUSR;
MCUSR = 0;
if (!(ch & _BV(EXTRF))) appStart();

#if LED_START_FLASHES > 0
```

Note that here MCUSR is cleared after use hence we cannot use it again for getting the reset source in our application program if we need. This is not an issue since we can modify the bootloader as per our requirement if we want.

If reset source is external (by pulling reset pin low) then it will avoid jump to application code directly and prepare for serial communication with avrdude running at PC/laptop to read hex file and flash it into program memory.

Watchdog is prepared for 1 second timeout in program to get reset if there is any error while uploading code or to get reset while program memory write completes.

```
// Set up watchdog to trigger after 1s
watchdogConfig(WATCHDOG_1S);
```

It will then initialize serial communication (here UART) to communicate with arduino IDE running at pc/laptop.

```
UCSR0A = _BV(U2X0); //Double speed mode USART0
UCSR0B = _BV(RXEN0) | _BV(TXEN0);
UCSR0C = _BV(UCSZ00) | _BV(UCSZ01);
UBRR0L = (uint8_t)( (F_CPU + BAUD_RATE * 4L) / (BAUD_RATE * 8L) - 1 );
```

After above initializations, it starts its forever loop to read bytes (command/data byte) serially using protocol used by STK500.

```
/* Forever loop */
for (;;) {
  /* get character from UART */
  ch = getch();

  if(ch == STK_GET_PARAMETER) {
    unsigned char which = getch();
    verifySpace();
    if (which == 0x82) {
```

Program memory is written/updated in page by page fashion. The page size varies according to the controller. For example, Atmega328/328P has a page size of 64 words (i.e. 128 bytes) whereas Atmega88A/88PA has a page size of 32 words (i.e. 64 bytes).

The process of writing program memory is carried out in page by page manner as follow.

- In above mentioned forever loop hex bytes coming serially from arduino uploader running at pc/laptop are first copied to the temporary data memory (say RAM).
- After copying page sized hex bytes in temporary data memory, program memory first page erase is processed.
- After erase of page, first it is filled (just fill not write) with hex bytes stored in temporary data memory.
- Then using SPM page write instruction, page write/update is successfully carried out.
- The above process of reading data from serial and then writing it to program memory in page by page fashion is carried out until complete hex bytes are written/updated in program memory.
- After completion of hex file write operation, opposite process is carried out i.e. reading from program memory and sending it serially to pc/laptop in page by page manner to verify whether hex file got written/updated in program memory or not.

Below is sample page write function which is already given in boot header file

```
#include <inttypes.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
void boot_program_page (uint32_t page, uint8_t *buf)
{
        uint16_t i;
        uint8_t sreg;
        // Disable interrupts.
        sreg = SREG;
        cli();
        eeprom_busy_wait ();
        boot_page_erase (page);  //erase page
        boot_spm_busy_wait ();      // Wait until the memory is erased.

        for (i=0; i<SPM_PAGESIZE; i+=2)
```

```
        {
                // Set up word from temp buffer.
                uint16_t w = *buf++;
                w += (*buf++) << 8;
                boot_page_fill (page + i, w);  //fill (page + i ) address with word
        }
        boot_page_write (page);     // Store/write buffer in flash page.
        boot_spm_busy_wait();       // Wait until the memory is written.
        // Reenable RWW-section again. We need this if we want to jump back
        // to the application after bootloading.
        boot_rww_enable ();
        // Re-enable interrupts (if they were ever enabled).
        SREG = sreg;
}
```

All above is basic general idea about how hex file gets written in program memory.The functions used in above program i.e. boot_page_fill (page address, word data), boot_page_write(page address), boot_spm_busy_wait()etc.all are available in boot header (boot.h) file which are written with inline assembly instructions.

How does a program residing at the bottom of the program memory itself manages to write into the program memory? i.e. how program in boot section writes into the application section. This is possible since avr microcontrollers provides a **self-programming mechanism(SPM)** for downloading and uploading code by the microcontroller itself. The Self-Programming can use any available data interface and associated protocol to read code and write (program) that code into the Program memory.