

AMS Lab Exercise 6

“Boot Loader”

HH, February 5, 2017

Page 1 of 5

Purpose

To understand Boot Loading by using a Boot Loader for the Mega32.
The exercise is based on the Atmel Application Note “AVR109”.

Literature

- The AMS Boot Loading lesson.
- Atmel Application Note AVR109 “Self Programming”.
- “AVR Prog” User Guide.

The Exercise

Start this exercise by reading the Atmel Application note AVR109.

The application note explains the AVR boot loading methods in general and then implements a boot loader for interfacing the “AVR Prog” program.

The “AVR Prog” uses the Mega32 UART for communication.

The application note is available at Blackboard.

An Atmel Studio project called “HHBoot” (zipped and available at Blackboard) is a boot loader for the Mega32 based on the AVR109 application note (and therefore also uses the “AVR Prog” protocol). Download (and unzip) this project and carefully study the implementation.

The implementation makes use of the AVR GCC libraries <avr/boot.h> (and <avr/pgmspace.h>).

Defines

<avr/boot.h>: Bootloader Support Utilities

Defines

```
#define BOOTLOADER_SECTION __attribute__((section(".bootloader")))
#define boot_spm_interrupt_enable() (__SPM_REG |= (uint8_t)_BV(SPMIE))
#define boot_spm_interrupt_disable() (__SPM_REG &= (uint8_t)~_BV(SPMIE))
#define boot_is_spm_interrupt() (__SPM_REG & (uint8_t)_BV(SPMIE))
#define boot_rww_busy() (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
#define boot_spm_busy() (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
#define boot_spm_busy_wait() do{}while(boot_spm_busy())
#define GET_LOW_FUSE_BITS (0x0000)
#define GET_LOCK_BITS (0x0001)
#define GET_EXTENDED_FUSE_BITS (0x0002)
#define GET_HIGH_FUSE_BITS (0x0003)
#define boot_lock_fuse_bits_get(address)
#define boot_signature_byte_get(addr)
#define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
#define boot_page_erase(address) __boot_page_erase_normal(address)
#define boot_page_write(address) __boot_page_write_normal(address)
#define boot_rww_enable() __boot_rww_enable()
#define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
#define boot_page_fill_safe(address, data)
#define boot_page_erase_safe(address)
#define boot_page_write_safe(address)
#define boot_rww_enable_safe()
#define boot_lock_bits_set_safe(lock_bits)
```

As a result we don't have to implement the boot loader functions in assembly.

AMS Lab Exercise 6

“Boot Loader”

HH, February 5, 2017

Page 2 of 5

To locate the boot loader code in the Boot Section (in this case we use a 1k word Boot Size), we make use of the macro `BOOTLOADER_SECTION` (for locating the `main()` function):

```
#include <avr/io.h>
#include <avr/boot.h>
#include <avr/pgmspace.h>
#include "uart.h"

#define PAGESIZE 128
// Assume 1024 word boot loader
#define APP_END 0x3BFF

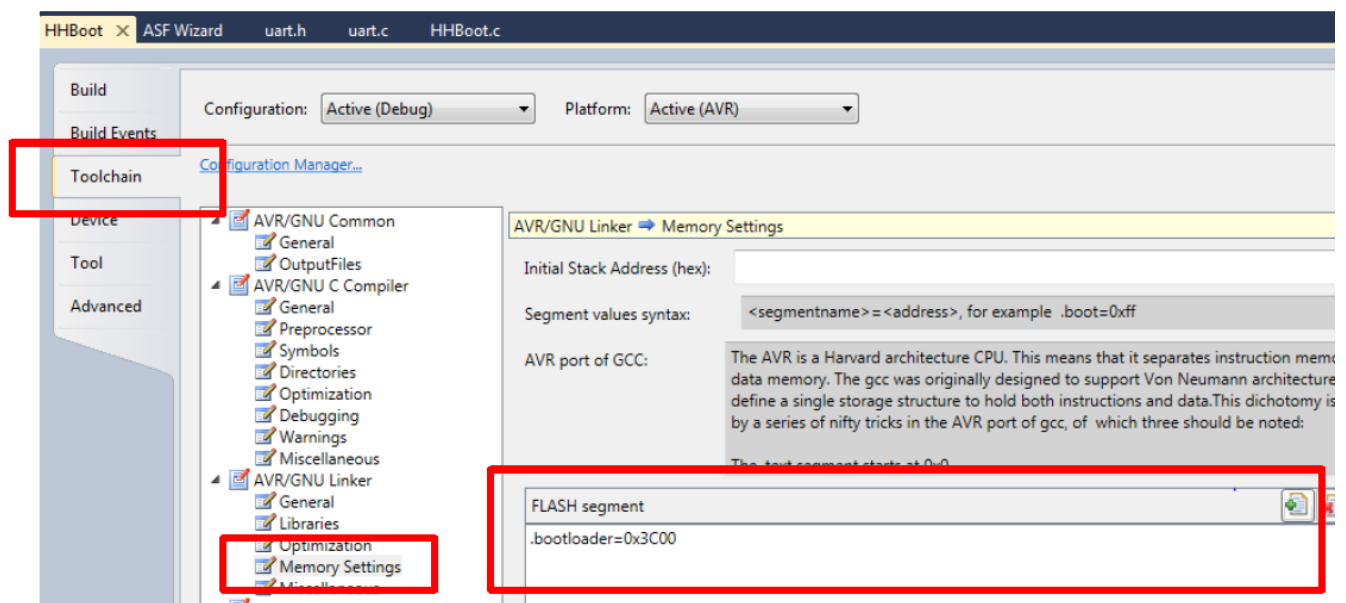
// Definitions for device recognition (ATMega32)
#define PARTCODE 0x73
#define SIGNATURE_BYTE_1 0x1E
#define SIGNATURE_BYTE_2 0x95
#define SIGNATURE_BYTE_3 0x02

BOOTLOADER_SECTION int main()
{
```

We locate the UART code by defining each function to also reside in the section “bootloader” as in this example:

```
__attribute__((section (".bootloader")))
char ReadChar()
{
    // Wait for new character received
    while ( (UCSRA & (1<<7)) == 0 )
    {}
    // Then return it
    return UDR;
}
```

To locate the “.bootloader” section to the beginning of the 1k word at the top of the flash memory, we set up the word address “**.bootloader=0x3C00**” in the “Memory Settings” of the tab “Toolchain” in the AVR Studio project properties:



AMS Lab Exercise 6

“Boot Loader”

HH, February 5, 2017

Page 3 of 5

To be able to start up the Boot Loader program correctly, we have to initialize the stack pointer etc. The assembly code in the start of the main function serves this purpose:

```
BOOTLOADER_SECTION int main()
{
    char val;
    unsigned int address;
    unsigned int temp_int;

    // Disable all interrupts while boot loading
    __asm ("cli");
    // Clear SREG
    __asm ("eor r1,r1");
    __asm ("out 0x3f,r1");
    // Stack pointer = 0x0800
    __asm ("ldi r28,0x5F");
    __asm ("ldi r29,0x08");
    __asm ("out 0x3e,r29");
    __asm ("out 0x3d,r28");

    // Initialize UART: 115200 bit/s, 8 data bits, no parity
    InitUART();
    while (1)
    {
```

To understand the rest of the boot loader , you have to study the (in fact quite simply) text based protocol of “AVR Prog” (explained in application note AVR109).

The code below needs some extra explanation:

```
// Exit bootloader.
else if(val=='E')
{
    boot_spm_busy_wait();
    boot_rww_enable();
    SendChar('\r');
    // Generate watchdog RESET (starting the application)
    WDTCSR = 0b00001000;
    while(1)
    {}
}
```

When the character ‘E’ is received (sent by AVR Prog when you hit the EXIT button), the Mega32 *watchdog timer* is enabled.

Then an endless loop is entered.

As we do not feed the watchdog in the endless loop, a RESET will be generated in a few milliseconds.

When the BOOTRST fuse is not set (ensure that), the application will start (leaving the boot loader).

AMS Lab Exercise 6

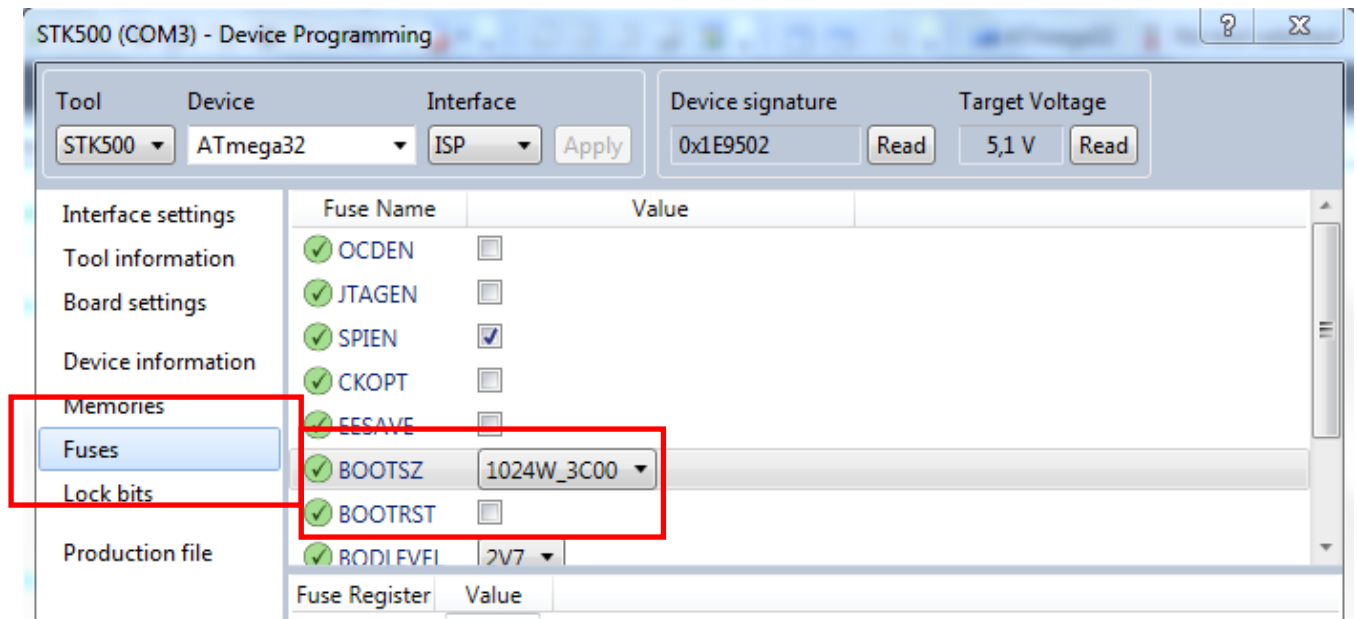
“Boot Loader”

HH, February 5, 2017

Page 4 of 5

Having a good understanding of how the boot loader works, it's time to test it:

First ensure that the BOOTRST fuse is not set, and set the BOOT Size to 1024 words:



Then download the boot loader to the Mega32.

Now write a simple application (the way we “use to”).

It could simply toggle a LED at the STK500 with some delay between.

Important: Also in the application you must implement code, so that we jump to the boot loader section (at 0x3C00) – for example when you press one of the STK500 switches.

Inline assembly code to jump to 0x3C00 looks like this in AVR GCC:

```
__asm ("jmp 0x3C00");
```

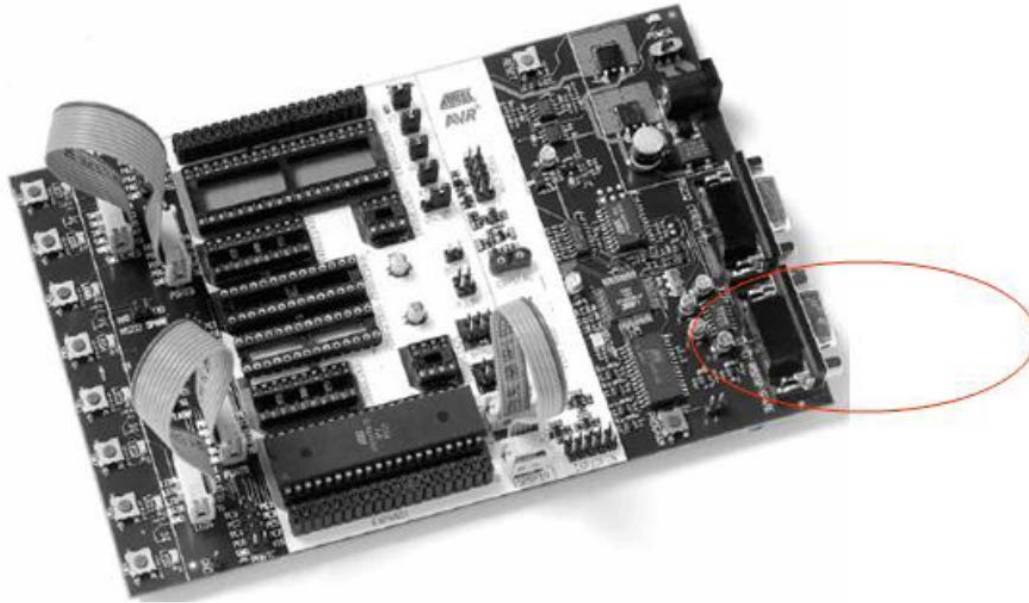
AMS Lab Exercise 6

“Boot Loader“

HH, February 5, 2017

Page 5 of 5

To use the boot loader for transferring the application to Mega32, first connect to the PC COM port (important: Has to be COM1 - COM4) to the Mega32 UART:



In addition, connect a 2 wire cable between pins "RS232 Spare" and 2 pins on PORTD:

RXD must be connected to PORTD, bit0.

TXD must be connected to PORTD, bit1.

Then start the “AVR Prog” program (when the boot loader is running), browse to the HEX-file of your application, “Program” the Flash and finally press “Exit”.

If succesfully, your application will start.

Then test your appllications ability to jump to the Boot Loader - and then download (another) application.

Extra:

Implement functions for an application to be able to use the Flash memory for non-volatile memory.

One function could copy data from an SRAM array to a flash page.

Another function could copy data from a flash page to an SRAM array.

Remember: The necessary SPM instruction only can be executed from the boot loader section.

Test the data for being non-volatile.