AMS Lab Exercise 6
"Boot Loader"

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

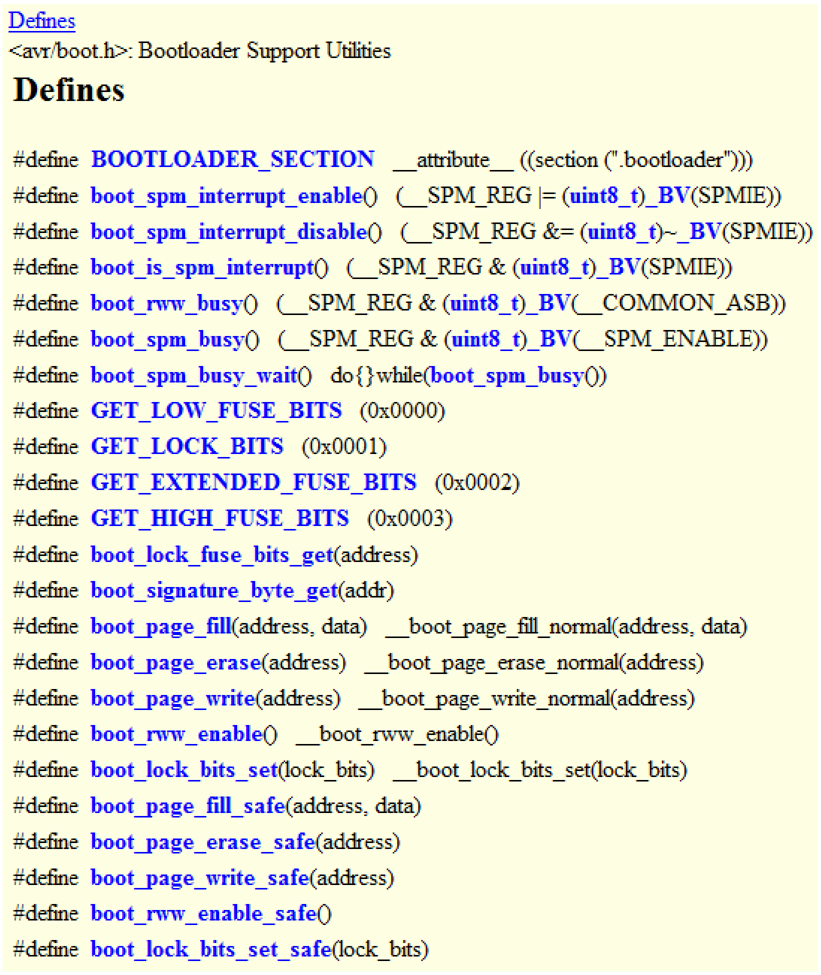
Literature
- The AMS Boot Loading lesson.
- Atmel Application Note AVR109 "Self Programming".
- "AVR Prog" User Guide.

The Exercise
Start this exercise by reading the Atmel Application note AVR109.
The application note explains the AVR Boot Loading architecture 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 Campusnet (folder "Files for LAB6").

This folder also contains a (zipped) AVR Studio project called "HHBoot" being a Boot Loader for the Mega32 based on the AVR109 application note (and 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.

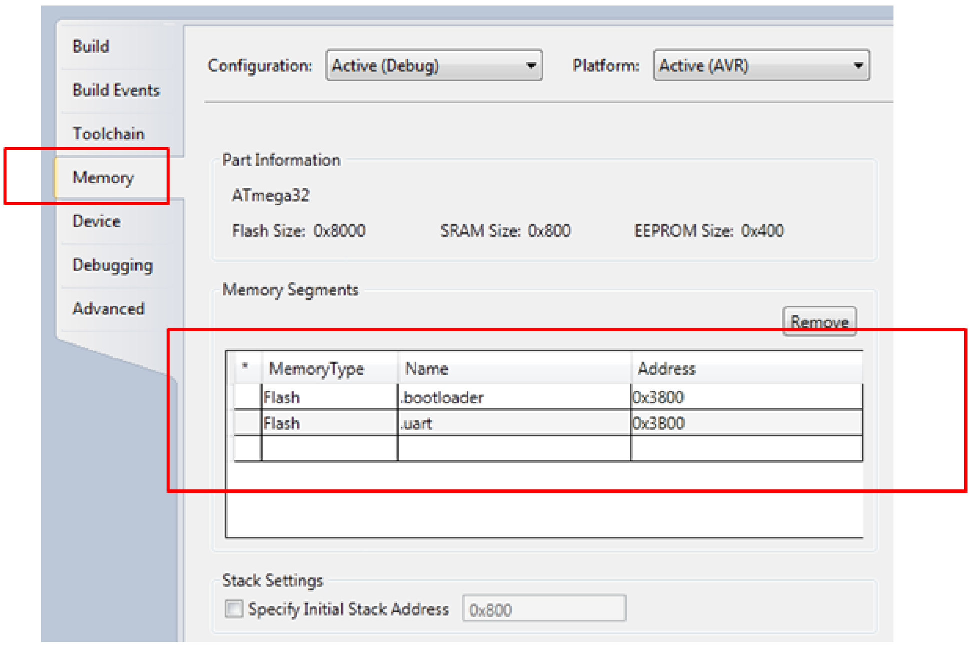
To locate the boot loader code in the Boot Section (in this case we use a 2 k word Boot Size), we make use of the macro BOOTLOADER_SECTION (for locating the main() function):

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

#define PAGESIZE 128
// Assume 2048 word boot loader
#define APP_END 0x3800

// 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 reside in the section "uart" as in this example:

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

To locate the 2 sections, we set up the addresses in the "memory" tab of the AVR Studio project properties:



Configuration: Active (Debug)   Platform: Active (AVR)

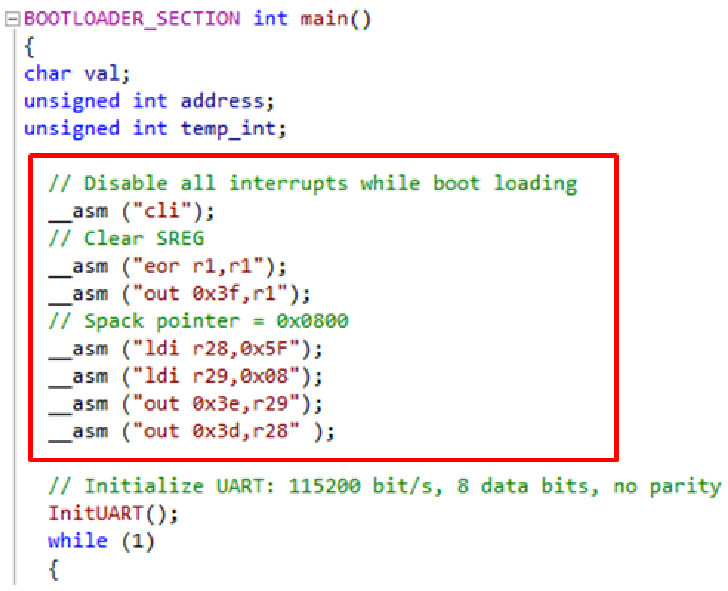
Part Information
ATmega32
Flash Size: 0x8000   SRAM Size: 0x800   EEPROM Size: 0x400

Memory Segments

MemoryType	Name	Address
Flash	.bootloader	0x3800
Flash	.uart	0x3B00

Stack Settings
Specify Initial Stack Address 0x800

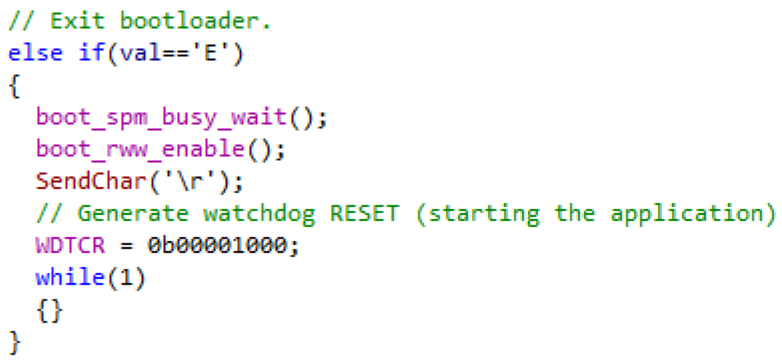
To start up the Boot Loader program correctly, we have to initialize the stack pointer etc.
The assembly code in the start of the main function serves this purpose:



```
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");
  // Spack 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 (quite simply) text based protocol of “AVR Prog” (explained in application note AVR109).

The code below deserves some extra explanation:



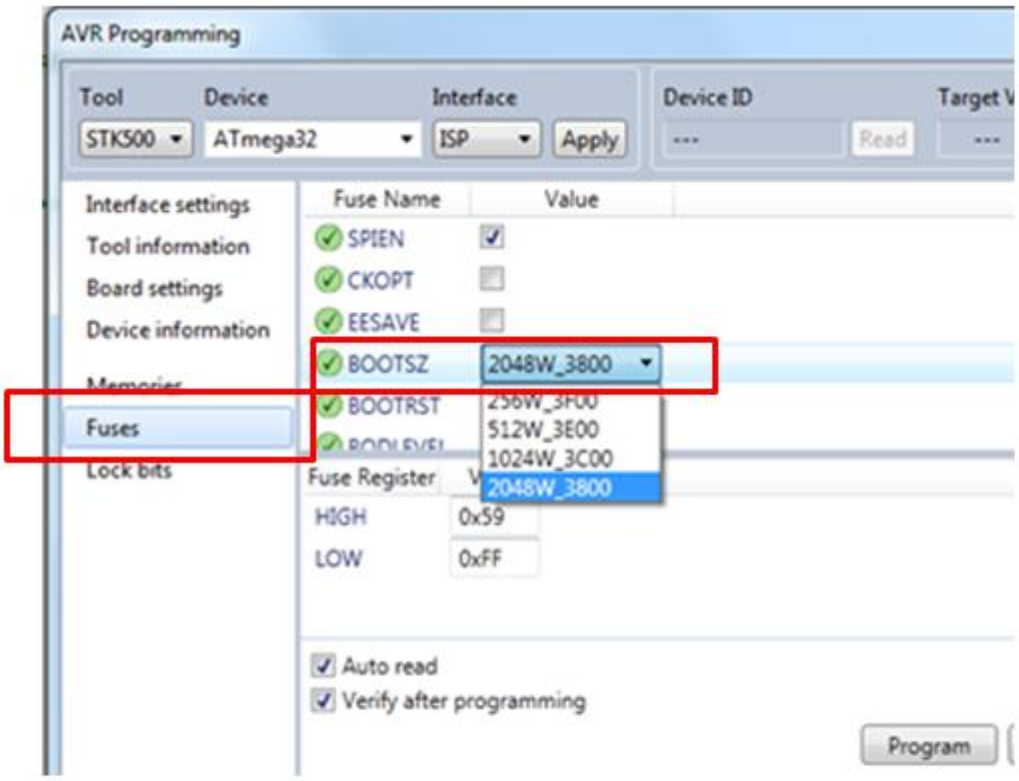
```
// Exit bootloader.
else if(val=='E')
{
  boot_spm_busy_wait();
  boot_rww_enable();
  SendChar('\r');
  // Generate watchdog RESET (starting the application)
  WDTCR = 0b00001000;
  while(1)
  {}
}
```

When the character ‘E’ is received (sent by AVR Prog when you hit the EXIT botton), the Mega32 watchdog timer is enabled.
Then an endless loop is entered. Since we do <u>not</u> 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).

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 2048 words:



Tool: STK500, Device: ATmega32, Interface: ISP, Device ID: ---

Fuse Name	Value
SPIEN	✓
CKOPT	
EESAVE	
BOOTSZ	2048W_3800 (256W_3F00, 512W_3E00, 1024W_3C00, 2048W_3800)
BOOTRST	
BODLEVEL	

Fuse Register	Value
HIGH	0x59
LOW	0xFF

✓ Auto read
✓ Verify after programming

Then download the boot loader to the Mega32.

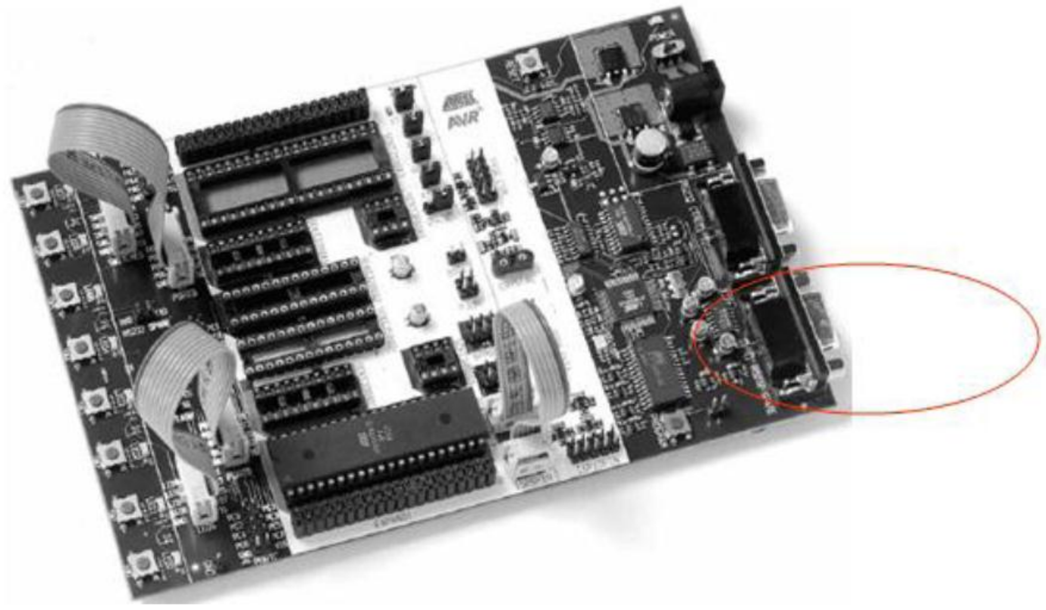
Now write a simple application (the way we “use to”).
It could simply toggle a LED at the STK500 with some delay between.

Important: Also in the application implement code, so that we jump to the Boot Loader section (at 0x3800) – for example when you press one of the STK500 switches.

In line assemly code to jump to 0x3800 looks like this in AVR GCC:

```
__asm ("jmp 0x3800");
```

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-conductor wire 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 apllication, “Program” the Flash and finally press “Exit”.

If everything went right, your application will start.

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

Extra (if you are bored):
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.