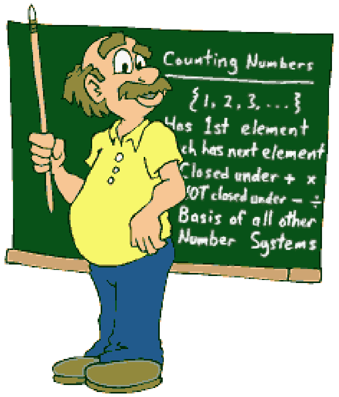
Ingeniørhøjskolen i Århus

iha.dk

AMS

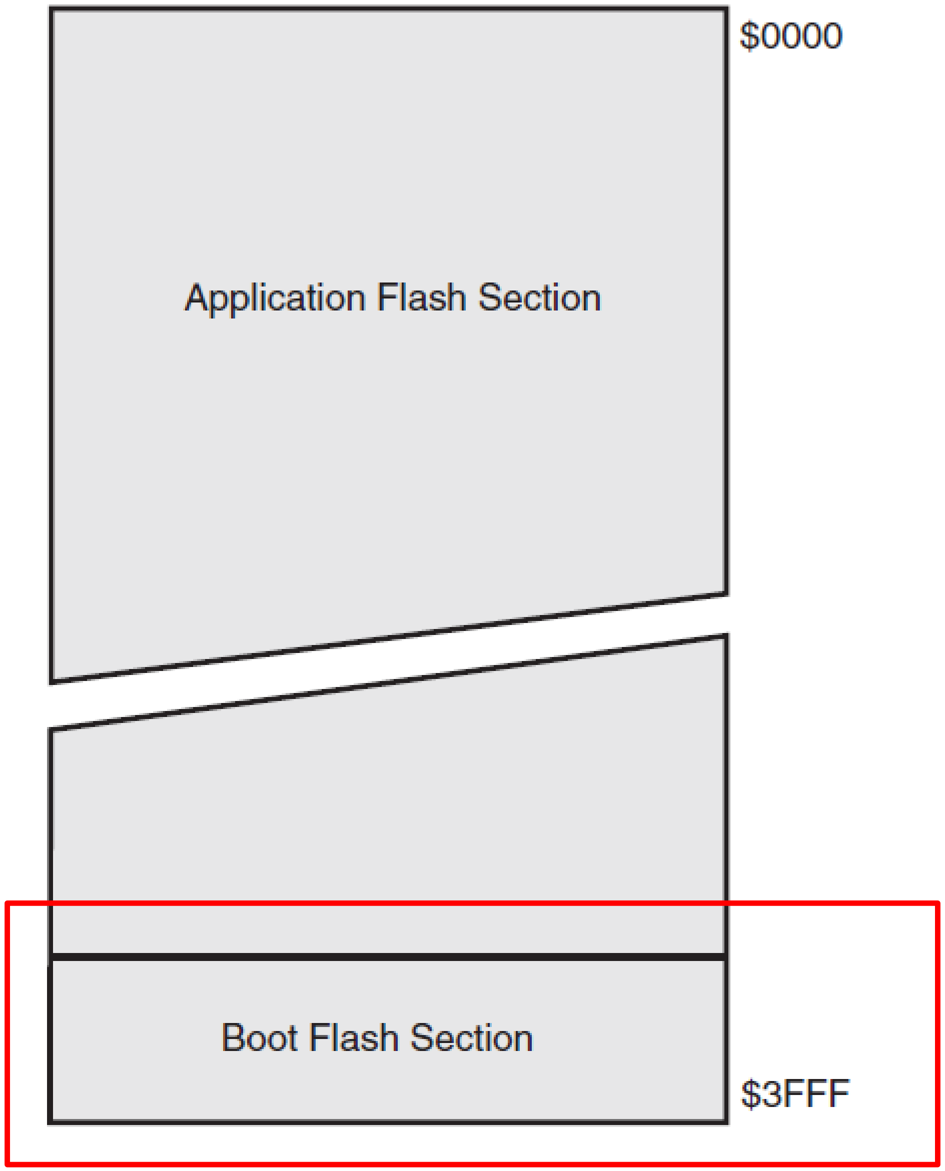
Applied Microcontroller Systems

Lesson 4: Boot Loading

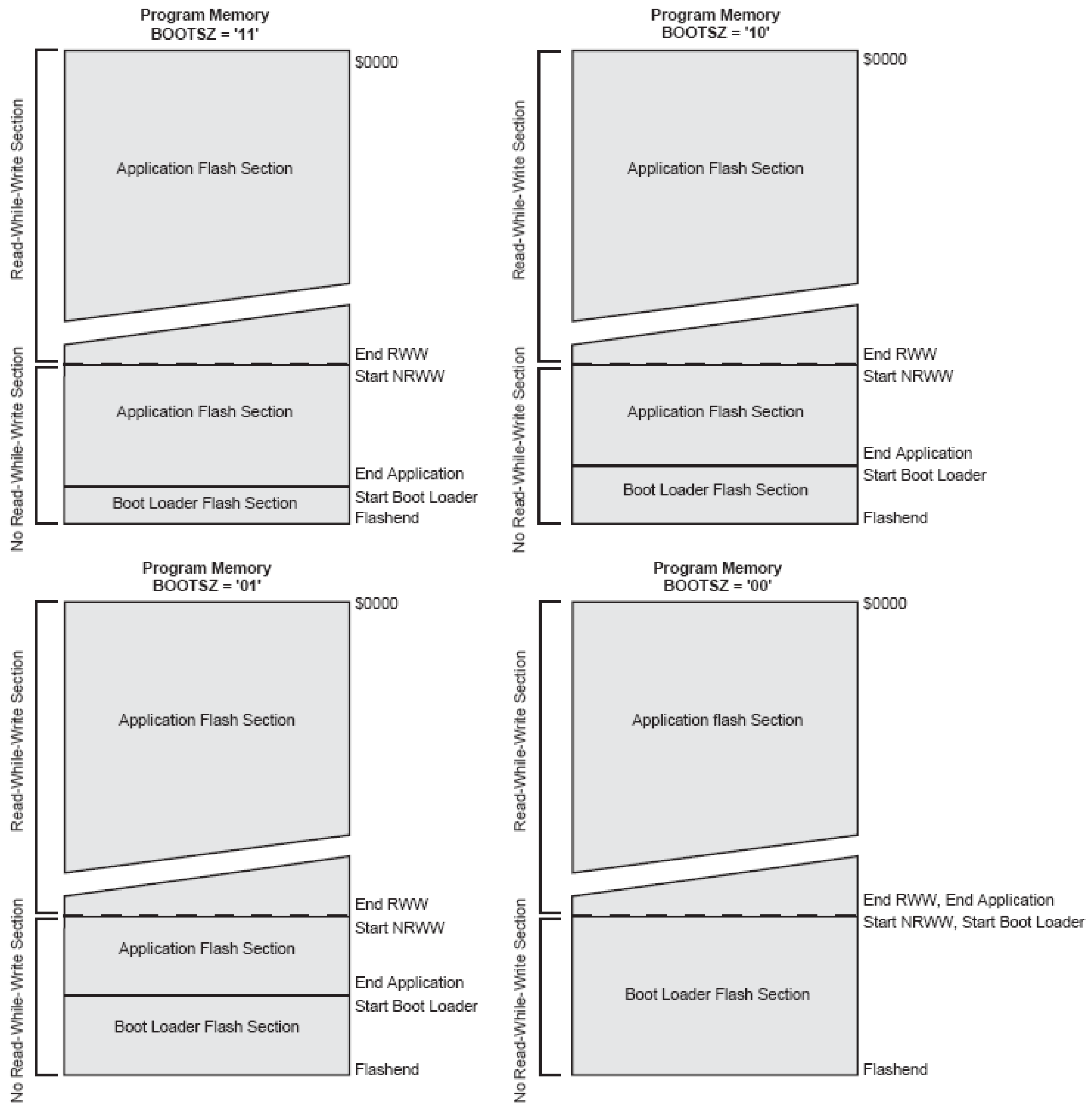


Version: 5-2-2017, Henning Hargaard

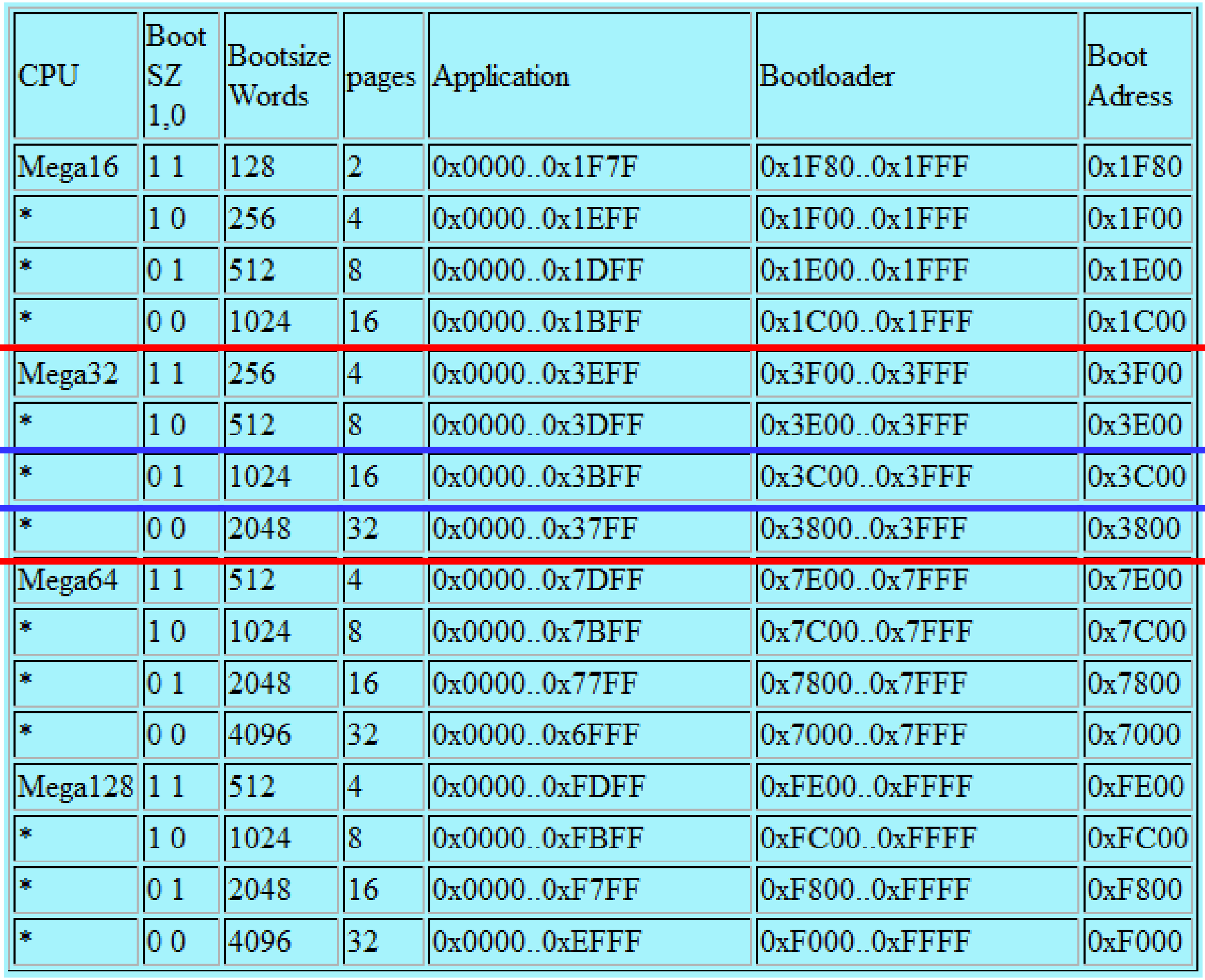
Mega32 Program Memory



Selectable Boot Section Size (Fuses)

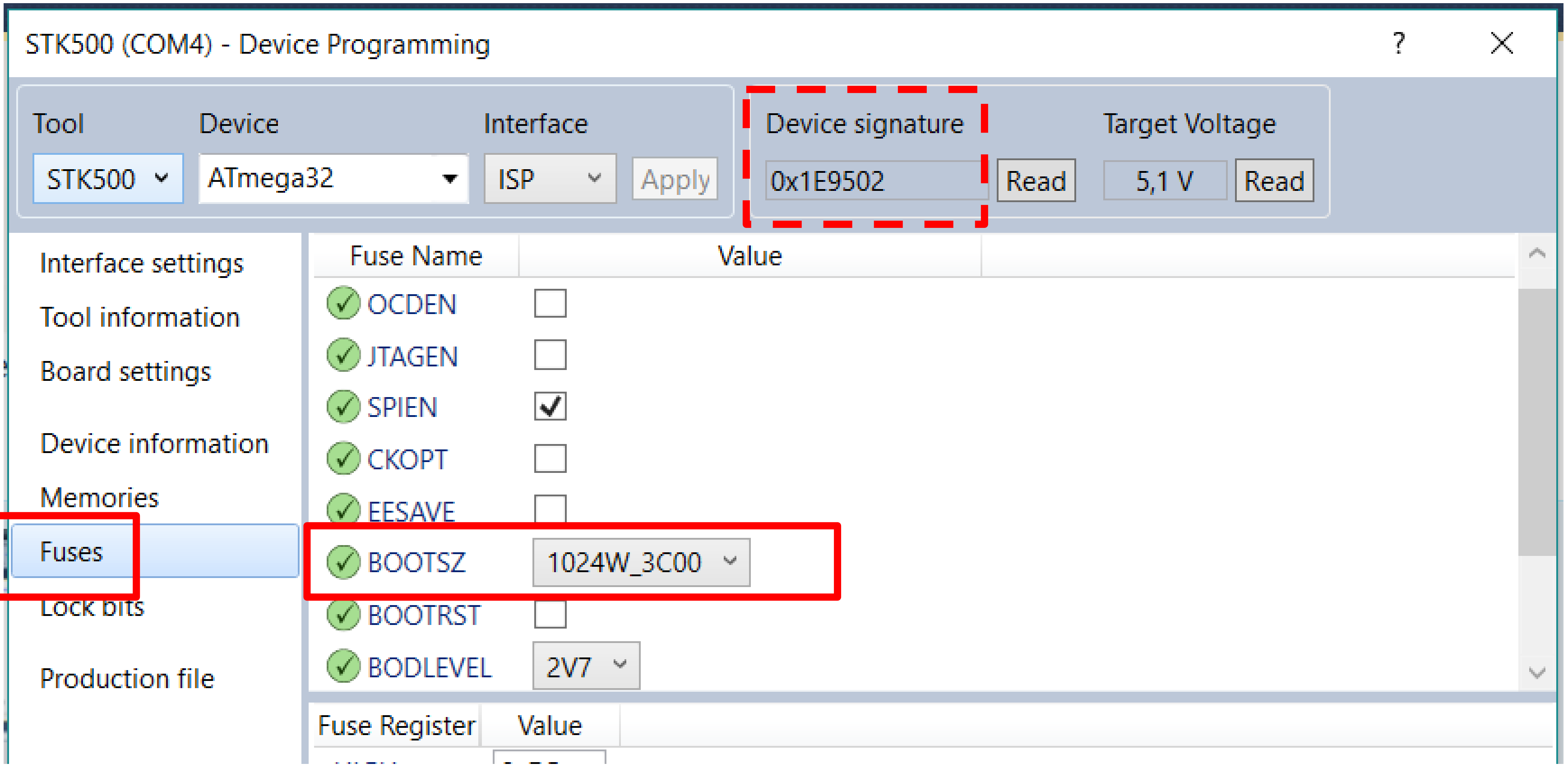


Bootblock Size



CPU	Boot SZ 1,0	Bootsize Words	pages	Application	Bootloader	Boot Adress
Mega16	1 1	128	2	0x0000..0x1F7F	0x1F80..0x1FFF	0x1F80
*	1 0	256	4	0x0000..0x1EFF	0x1F00..0x1FFF	0x1F00
*	0 1	512	8	0x0000..0x1DFF	0x1E00..0x1FFF	0x1E00
*	0 0	1024	16	0x0000..0x1BFF	0x1C00..0x1FFF	0x1C00
Mega32	1 1	256	4	0x0000..0x3EFF	0x3F00..0x3FFF	0x3F00
*	1 0	512	8	0x0000..0x3DFF	0x3E00..0x3FFF	0x3E00
*	0 1	1024	16	0x0000..0x3BFF	0x3C00..0x3FFF	0x3C00
*	0 0	2048	32	0x0000..0x37FF	0x3800..0x3FFF	0x3800
Mega64	1 1	512	4	0x0000..0x7DFF	0x7E00..0x7FFF	0x7E00
*	1 0	1024	8	0x0000..0x7BFF	0x7C00..0x7FFF	0x7C00
*	0 1	2048	16	0x0000..0x77FF	0x7800..0x7FFF	0x7800
*	0 0	4096	32	0x0000..0x6FFF	0x7000..0x7FFF	0x7000
Mega128	1 1	512	4	0x0000..0xFDFF	0xFE00..0xFFFF	0xFE00
*	1 0	1024	8	0x0000..0xFBFF	0xFC00..0xFFFF	0xFC00
*	0 1	2048	16	0x0000..0xF7FF	0xF800..0xFFFF	0xF800
*	0 0	4096	32	0x0000..0xEFFF	0xF000..0xFFFF	0xF000

Atmel Studio: Setting Bootblock Size



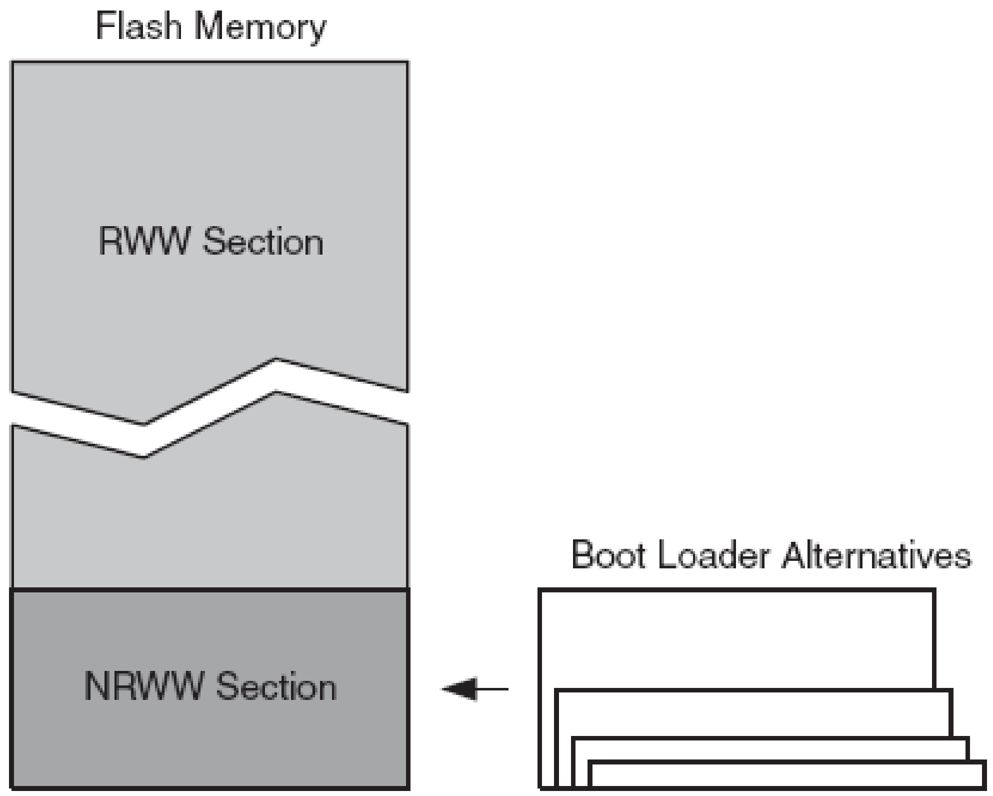
STK500 (COM4) - Device Programming

Tool	Device	Interface	Device signature	Target Voltage
STK500	ATmega32	ISP	0x1E9502	5,1 V

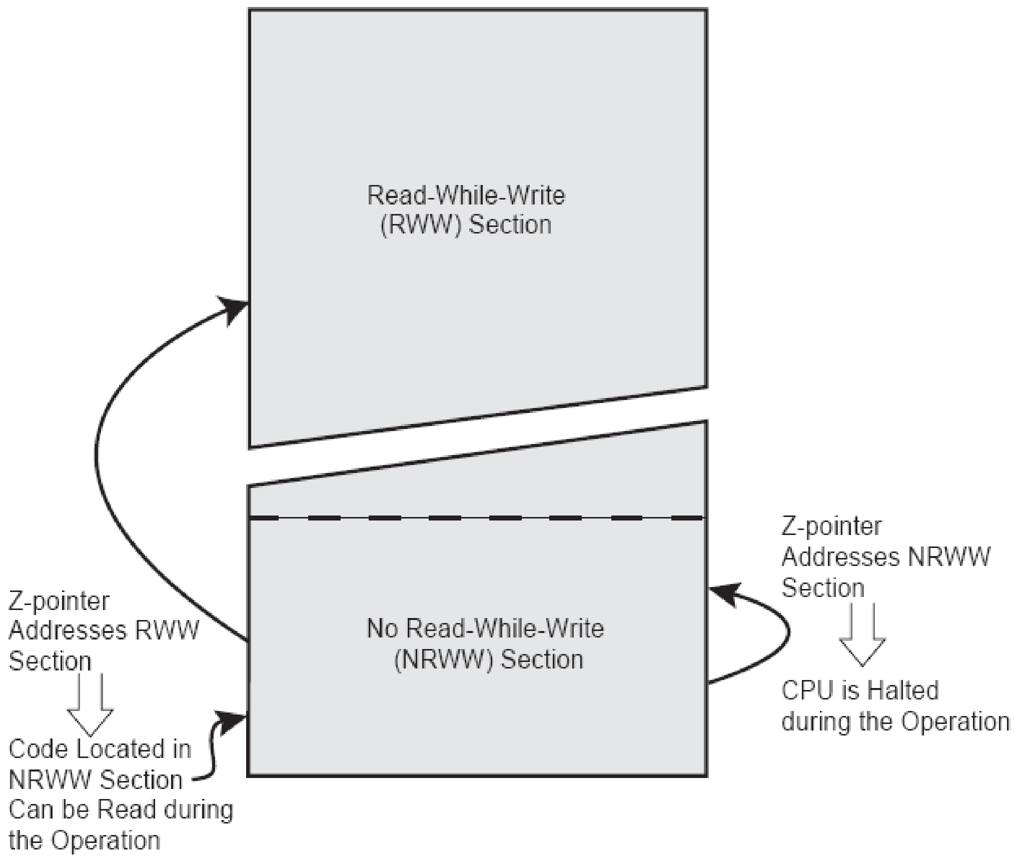
Fuse Name	Value
✓ OCDEN	☐
✓ JTAGEN	☐
✓ SPIEN	☑
✓ CKOPT	☐
✓ EESAVE	☐
✓ BOOTSZ	1024W_3C00
✓ BOOTRST	☐
✓ BODLEVEL	2V7

Fuse Register	Value

RWW and NRWW Sections

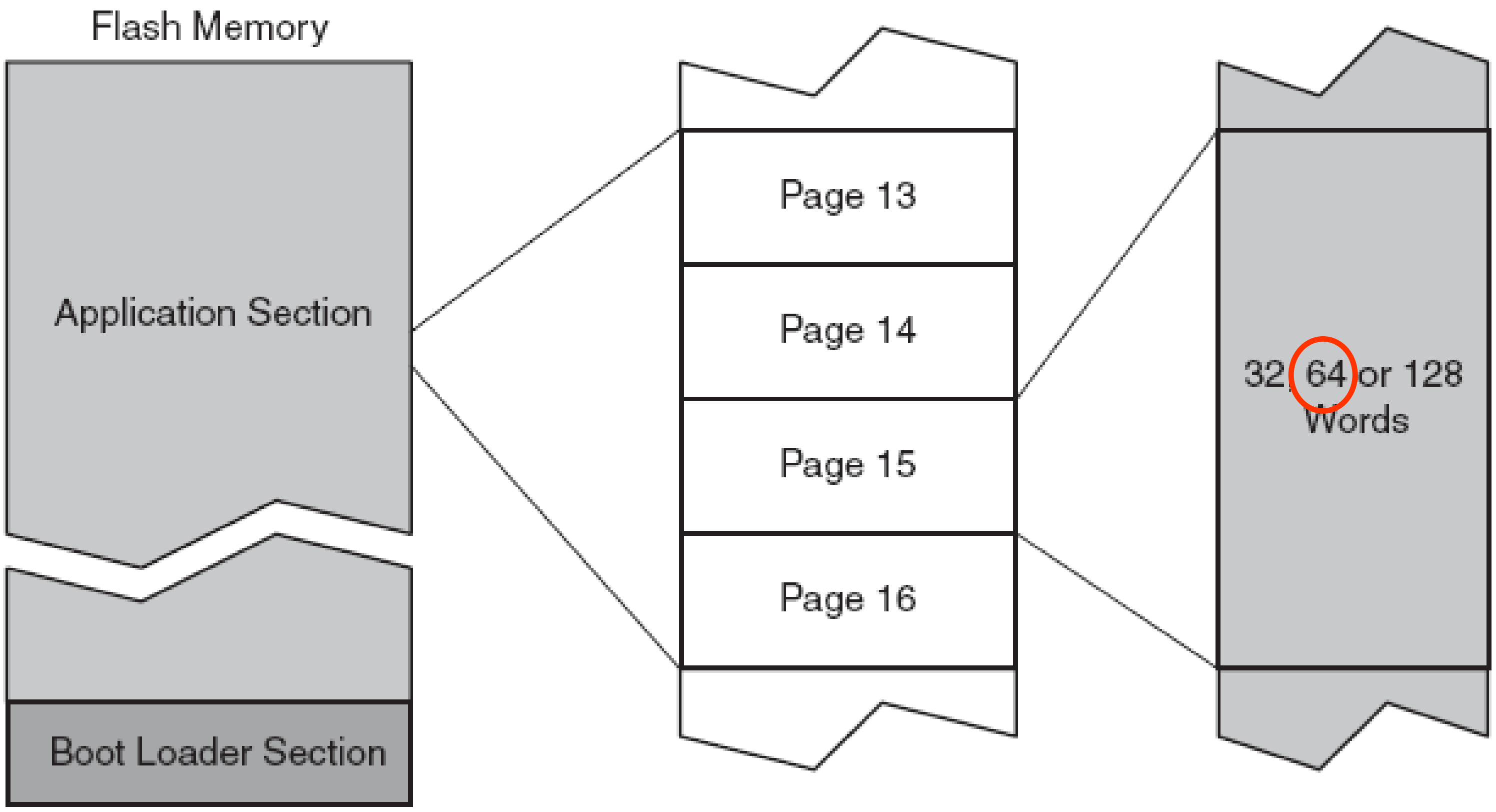


RWW and NRWW differences


Read-While-Write (RWW) Section
No Read-While-Write (NRWW) Section
Z-pointer Addresses RWW Section
Code Located in NRWW Section Can be Read during the Operation
Z-pointer Addresses NRWW Section
CPU is Halted during the Operation

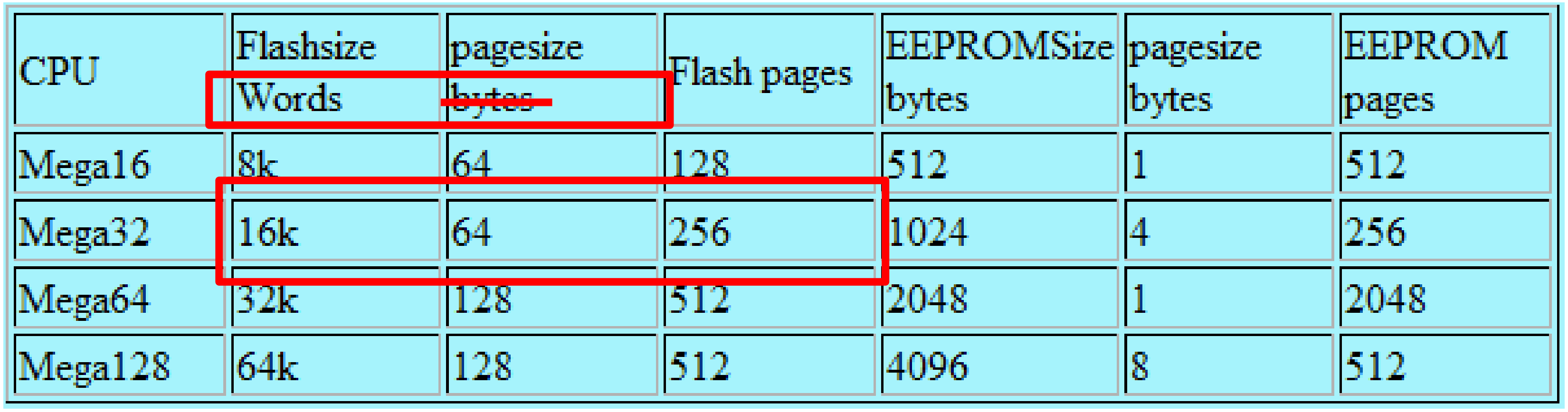
Which Section does the Z-pointer Address during the Programming?	Which Section can be Read during Programming?	Is the CPU Halted?	Read-While-Write Supported?
RWW section	NRWW section	No	Yes
NRWW section	None	Yes	No

Program Memory Paging



Mega32: 64 Words / Page
(= 128 Bytes / Page).

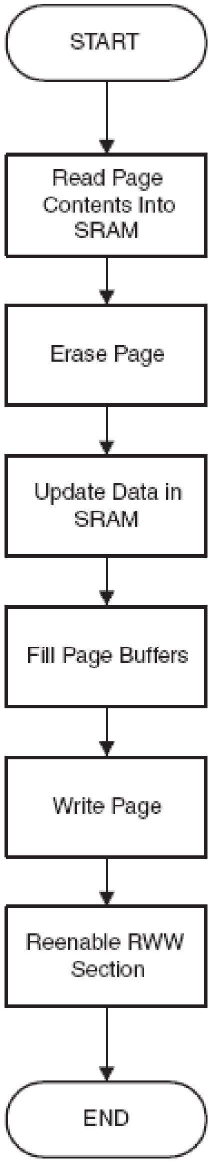
Page Size

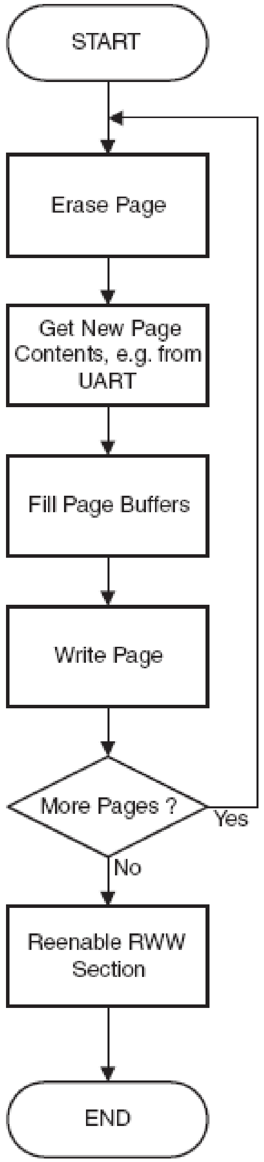


CPU	Flashsize Words	pagesize ~~bytes~~	Flash pages	EEPROMSize bytes	pagesize bytes	EEPROM pages
Mega16	8k	64	128	512	1	512
Mega32	16k	64	256	1024	4	256
Mega64	32k	128	512	2048	1	2048
Mega128	64k	128	512	4096	8	512

Typical Update Procedures

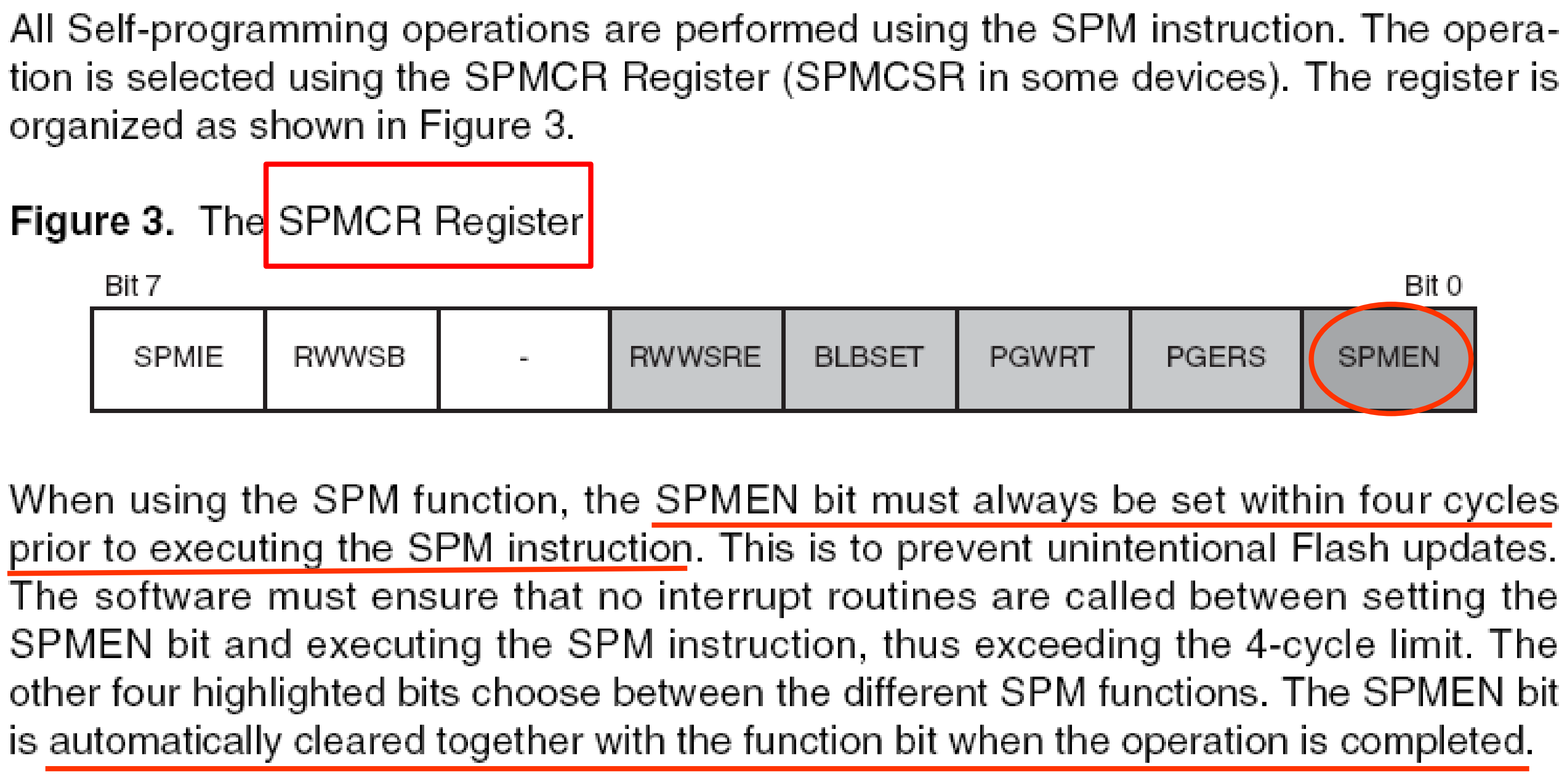
(Read-Modify-Write)





Page Write

Using the SPM instruction



All Self-programming operations are performed using the SPM instruction. The operation is selected using the SPMCR Register (SPMCSR in some devices). The register is organized as shown in Figure 3.

Figure 3. The SPMCR Register

Bit 7							Bit 0
SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN

When using the SPM function, the SPMEN bit must always be set within four cycles prior to executing the SPM instruction. This is to prevent unintentional Flash updates. The software must ensure that no interrupt routines are called between setting the SPMEN bit and executing the SPM instruction, thus exceeding the 4-cycle limit. The other four highlighted bits choose between the different SPM functions. The SPMEN bit is automatically cleared together with the function bit when the operation is completed.

Inline assembly (AVR GCC):

```
  __asm("SPM");
```

Page Erase

All Flash memory updates are done page by page. Before writing new data to a page, the page must be erased.

The Z-register is used to select the page to be erased. Set up the Z-register to point to a byte in the page to be erased. The lower bits selecting the byte within the page are ignored. For instance, on a device with a page size of 32 words (64 bytes), the lower six bits of the Z-register are ignored.

To erase a page, set the PGERS and SPMEN bits in the SPMCR Register and execute the SPM instruction.

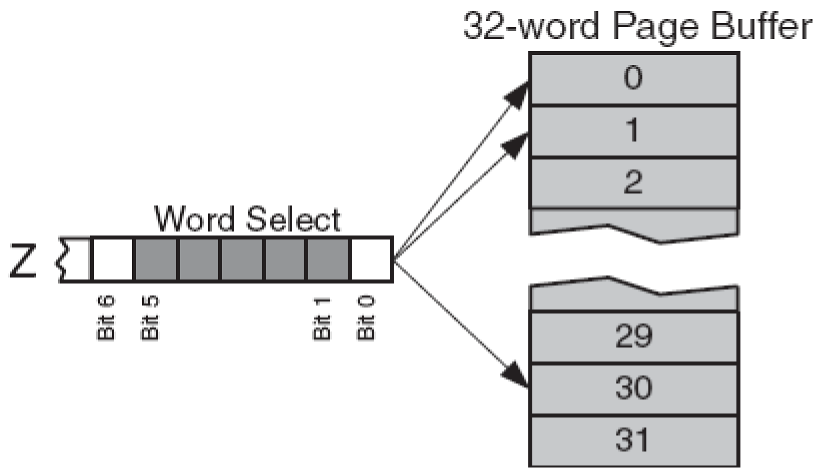
Bit 7							Bit 0
SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN

Loading Page Buffer

To write new data to a page, the Page Buffer must be filled first. The Page Buffer is a separate (not SRAM) write-only buffer holding one temporary page. This buffer must be filled word by word. The buffer is copied to Flash memory in one operation.

The Z-register is used to select the word to be written into the buffer. The LSB of Z is ignored, as an entire word is always written in one operation. Single byte access is thus not possible. The higher bits of Z selecting the page are ignored when writing to the Page Buffers. The Z-register bit structure for a 32-word (64-byte) page is shown in Figure 4. Larger page sizes use more bits for word selection.

Figure 4. Writing to Page Buffer

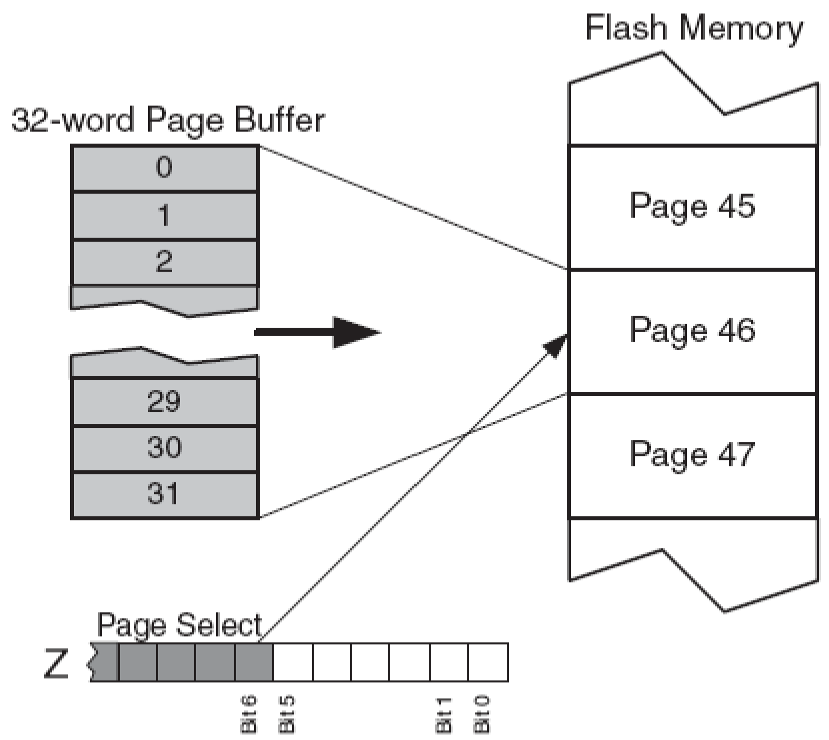


To write a word to the Page Buffer, load the word into the R1:R0 Registers. Set the Z-register to point to the correct word and set only the SPMEN bit in the SPMCR Register. The SPM instruction must then be executed within four cycles.

Page Write

When the Page Buffer is loaded with new data, it must be written to Flash memory. To do this, set up the Z-register the same way as described in the section regarding Page Erase. Then set the PGWRT and SPMEN bits in the SPMCR Register and execute the SPM instruction within four cycles. The R1:R0 Register contents are ignored. The use of the Z-register for 32-word (64-byte) page write is shown in Figure 5.

Figure 5. Writing a Page to Flash



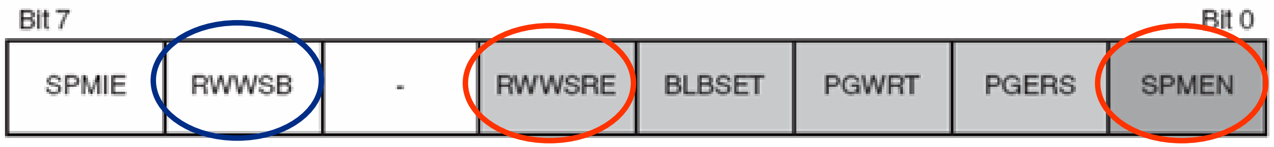
The SPMEN bit can be polled to find out when the CPU is ready for further page updates. The update procedure can also be interrupt controlled. See the section on interrupts below for more information.

The RWW Section Busy Flag

When performing a Page Erase or Page Write operation on the RWW section, the RWWSB Flag is set by hardware, indicating that the section is inaccessible. The RWWSB Flag should be cleared in software when the SPM operation is completed. This is done by setting the RWWSRE and SPMEN bits in the SPMCR Register, followed by an SPM instruction within four cycles. Alternatively, the flag is automatically cleared by starting to load the Page Buffers. The RWWSB Flag can be used by other parts of the application to check the RWW section's current accessibility. Refer to the devices' data sheet for more details.

Note that the contents of the Z-register and the R1:R0 Registers are ignored when using the RWWSRE function.

Note that if the RWW section accessed without re-enabling it after an erase or write operation, all addresses in the RRW section read 0xFFFF. This applies both when reading the Flash using LPM and if performing calls or jumps into the RWW section. The consequence of performing a jump into the RWW section without enabling it will therefore be that the program code "0xFFFF" is executed, eventually leading to that the program counter "falls" through the code space until it meets the first executable code. The first executable code would in that case be encountered on the first address of the NRWW section.



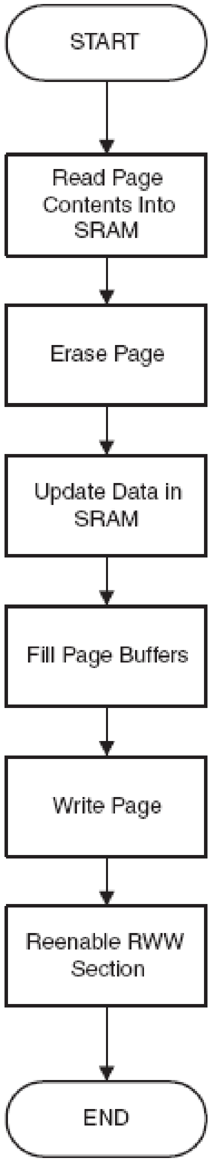
Bit 7							Bit 0
SPMIE	RWWSB	-	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN

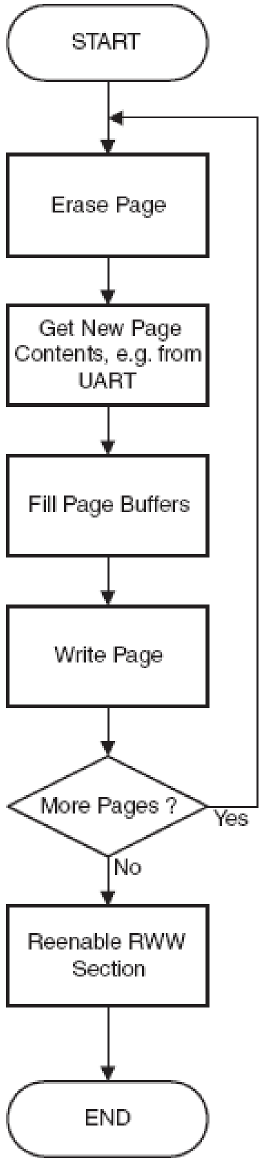
Avoid EEPROM Conflicts

Note that all write operations to the EEPROM must be finished before executing the SPM instruction and vice versa. Write/erase of the Flash and EEPROM cannot occur simultaneously.

Typical Update Procedures

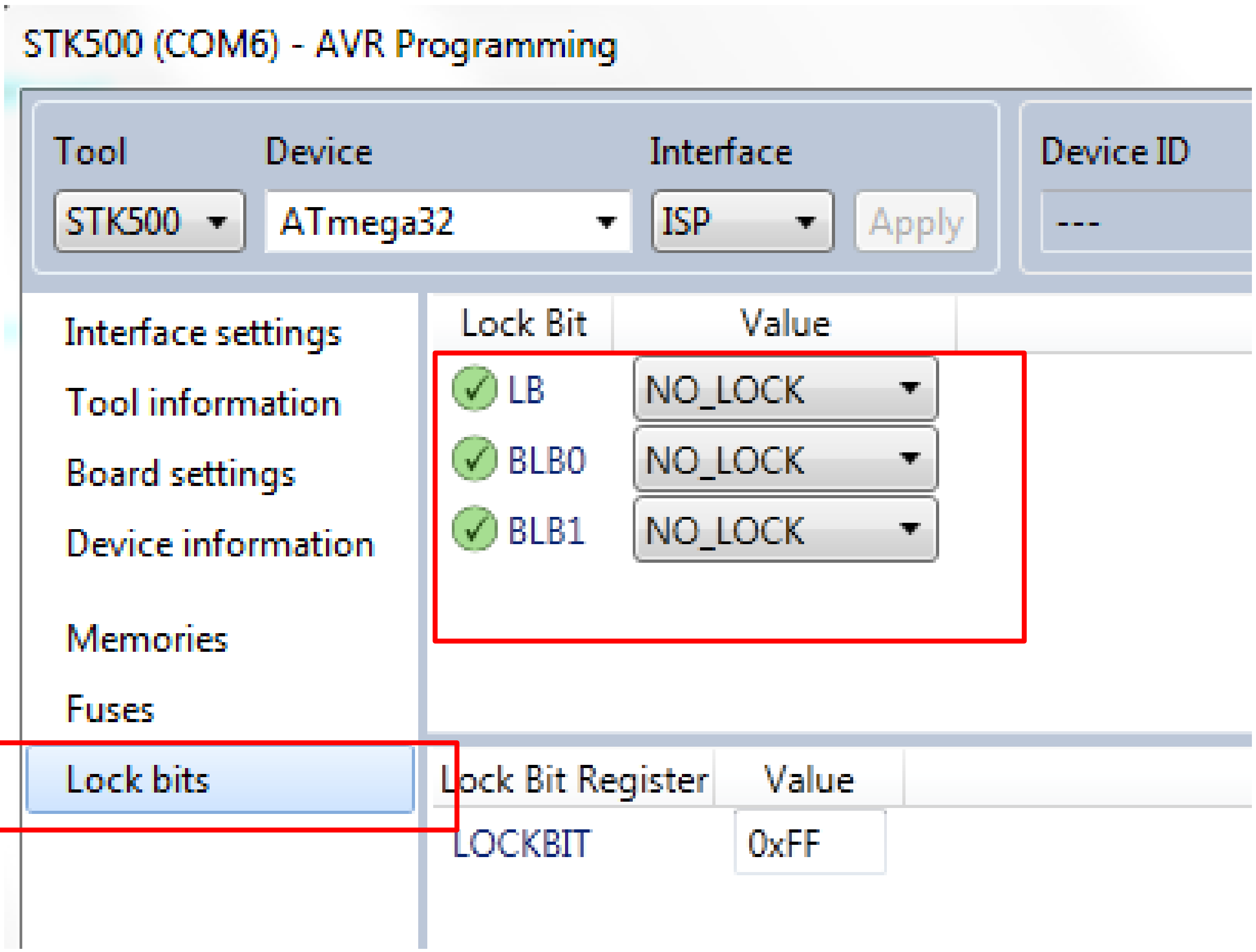
Read-Modify-Write





Page Write

Atmel Studio: Mega32 Lock Bits

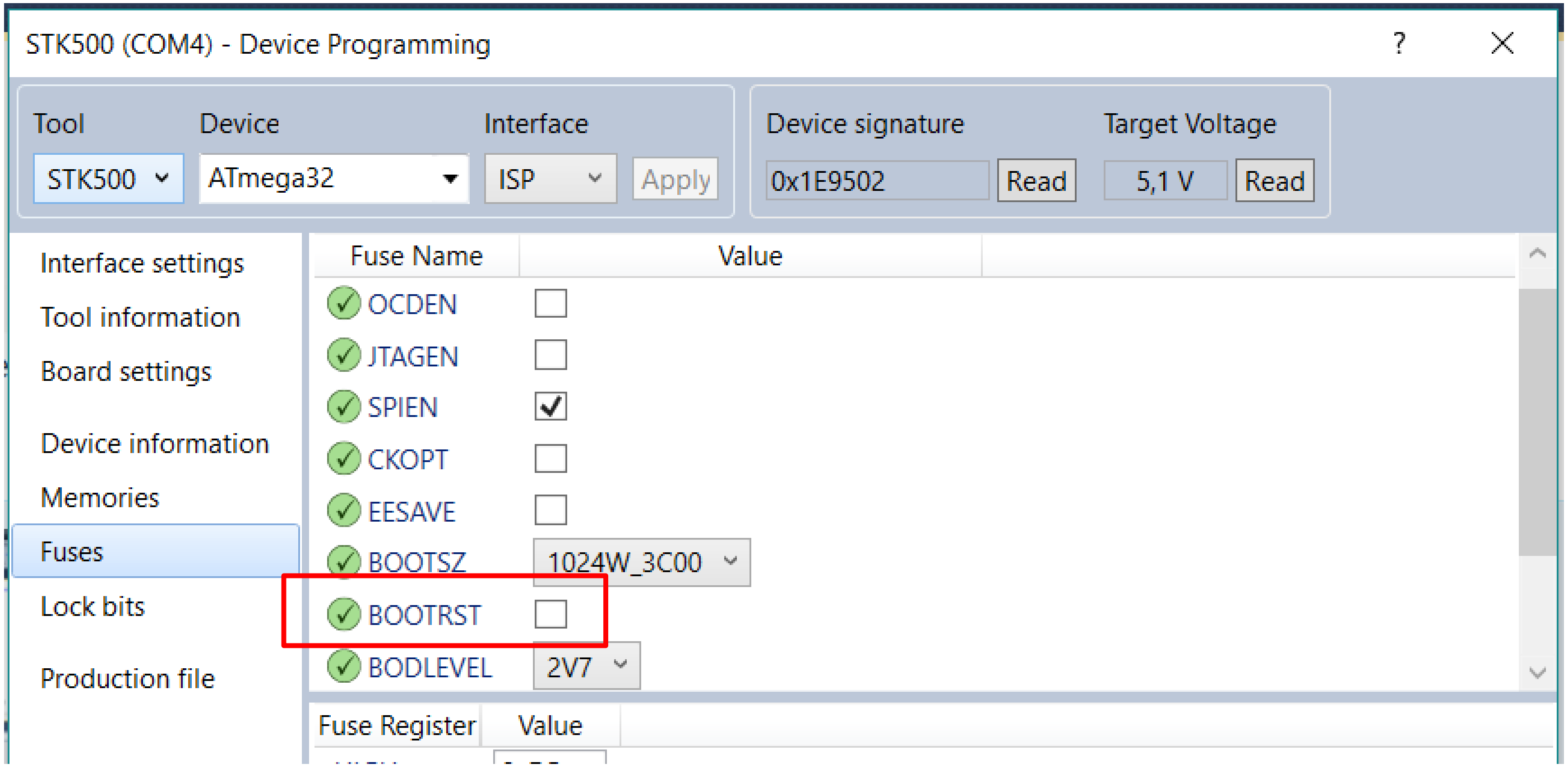


STK500 (COM6) - AVR Programming

Lock Bit	Value
✓ LB	NO_LOCK
✓ BLB0	NO_LOCK
✓ BLB1	NO_LOCK

Lock Bit Register	Value
LOCKBIT	0xFF

Atmel Studio: Mega32 Reset Fuse


STK500 (COM4) - Device Programming

Tool	Device	Interface	Device signature	Target Voltage
STK500	ATmega32	ISP	0x1E9502	5,1 V

Fuse Name	Value
✓ OCDEN	☐
✓ JTAGEN	☐
✓ SPIEN	☑
✓ CKOPT	☐
✓ EESAVE	☐
✓ BOOTSZ	1024W_3C00
✓ BOOTRST	☐
✓ BODLEVEL	2V7

Fuse Register	Value

If BOOTRST set: Program execution starts in Boot Loader.
If BOOTRST not set: Program execution starts in application (0).

Store Program Mem. Control Register

Bit	7	6	5	4	3	2	1	0	
	SPMIE	RWWSB	–	RWWSRE	BLBSET	PGWRT	PGERS	SPMEN	SPMCR
Read/Write	R/W	R	R	R/W	R/W	R/W	R/W	R/W	
Initial value	0	0	0	0	0	0	0	0	

- **Bit 7 – SPMIE: SPM Interrupt Enable**

When the SPMIE bit is written to one, and the I-bit in the Status Register is set (one), the SPM ready interrupt will be enabled. The SPM ready Interrupt will be executed as long as the SPMEN bit in the SPMCR Register is cleared.

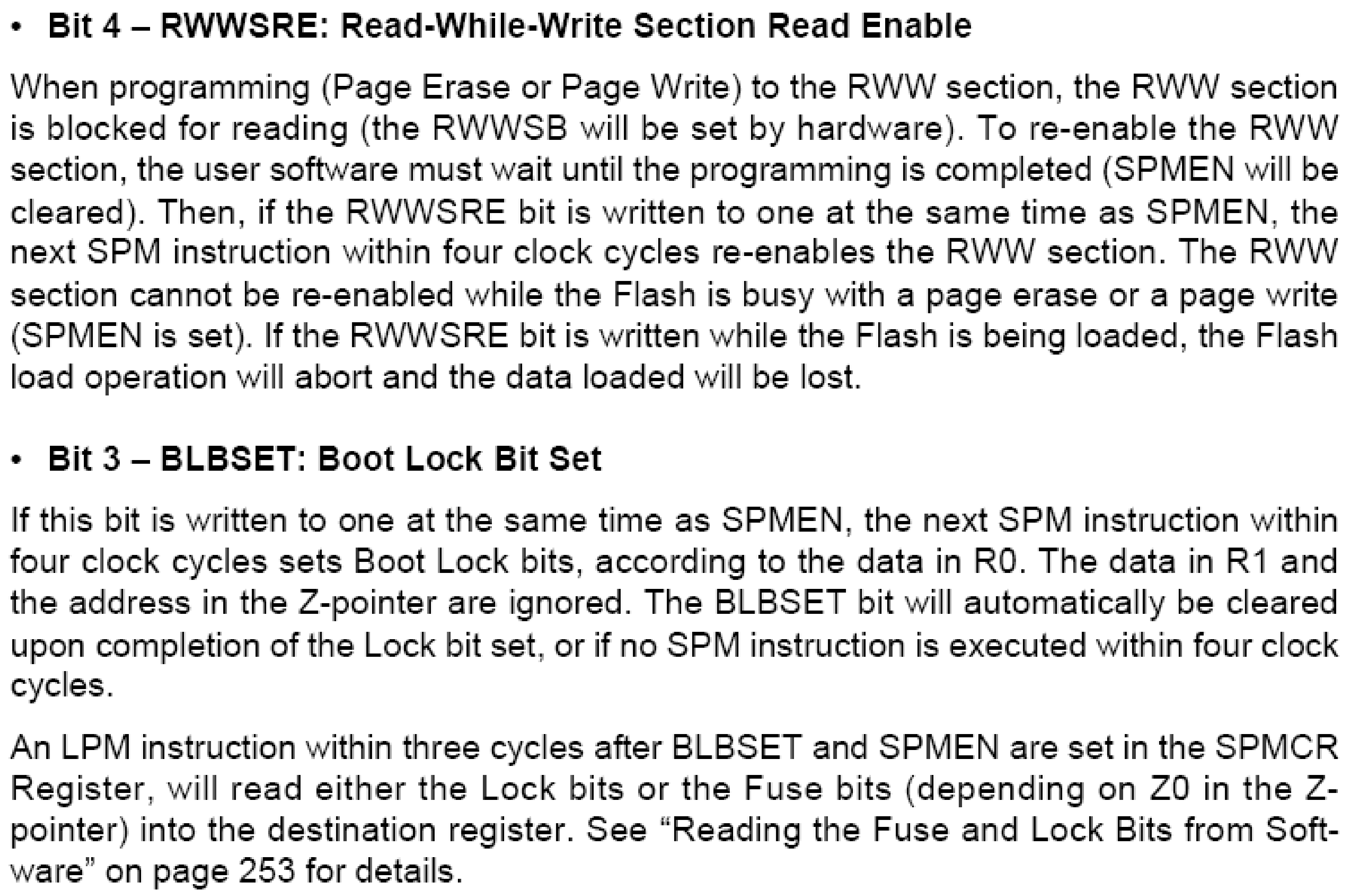
- **Bit 6 – RWWSB: Read-While-Write Section Busy**

When a self-programming (Page Erase or Page Write) operation to the RWW section is initiated, the RWWSB will be set (one) by hardware. When the RWWSB bit is set, the RWW section cannot be accessed. The RWWSB bit will be cleared if the RWWSRE bit is written to one after a Self-Programming operation is completed. Alternatively the RWWSB bit will automatically be cleared if a page load operation is initiated.

- **Bit 5 – Res: Reserved Bit**

This bit is a reserved bit in the ATmega16 and always read as zero.

Store Program Mem. Control Register



- **Bit 4 – RWWSRE: Read-While-Write Section Read Enable**

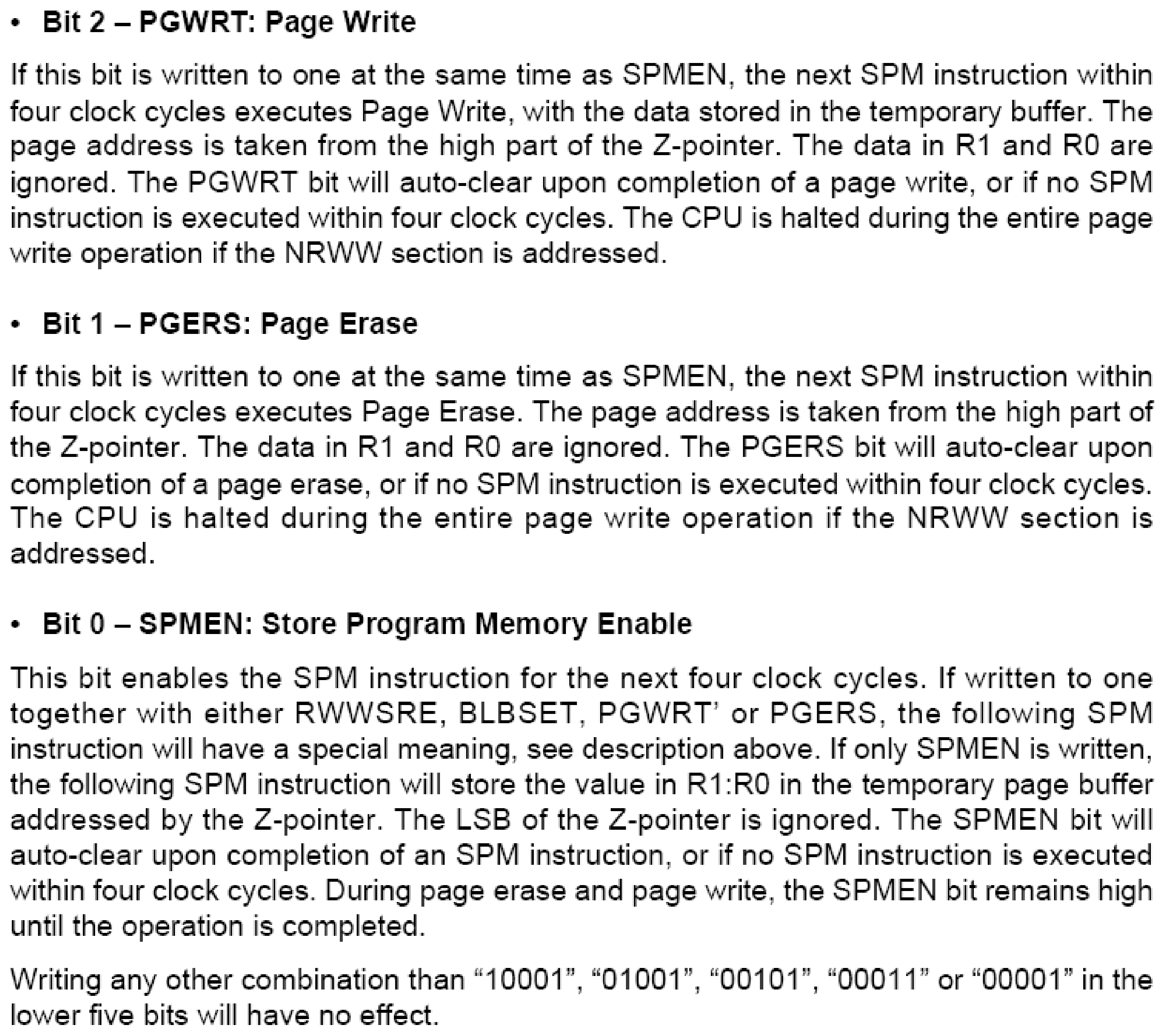
When programming (Page Erase or Page Write) to the RWW section, the RWW section is blocked for reading (the RWWSB will be set by hardware). To re-enable the RWW section, the user software must wait until the programming is completed (SPMEN will be cleared). Then, if the RWWSRE bit is written to one at the same time as SPMEN, the next SPM instruction within four clock cycles re-enables the RWW section. The RWW section cannot be re-enabled while the Flash is busy with a page erase or a page write (SPMEN is set). If the RWWSRE bit is written while the Flash is being loaded, the Flash load operation will abort and the data loaded will be lost.

- **Bit 3 – BLBSET: Boot Lock Bit Set**

If this bit is written to one at the same time as SPMEN, the next SPM instruction within four clock cycles sets Boot Lock bits, according to the data in R0. The data in R1 and the address in the Z-pointer are ignored. The BLBSET bit will automatically be cleared upon completion of the Lock bit set, or if no SPM instruction is executed within four clock cycles.

An LPM instruction within three cycles after BLBSET and SPMEN are set in the SPMCR Register, will read either the Lock bits or the Fuse bits (depending on Z0 in the Z-pointer) into the destination register. See “Reading the Fuse and Lock Bits from Software” on page 253 for details.

Store Program Mem. Control Register



- **Bit 2 – PGWRT: Page Write**

If this bit is written to one at the same time as SPMEN, the next SPM instruction within four clock cycles executes Page Write, with the data stored in the temporary buffer. The page address is taken from the high part of the Z-pointer. The data in R1 and R0 are ignored. The PGWRT bit will auto-clear upon completion of a page write, or if no SPM instruction is executed within four clock cycles. The CPU is halted during the entire page write operation if the NRWW section is addressed.

- **Bit 1 – PGERS: Page Erase**

If this bit is written to one at the same time as SPMEN, the next SPM instruction within four clock cycles executes Page Erase. The page address is taken from the high part of the Z-pointer. The data in R1 and R0 are ignored. The PGERS bit will auto-clear upon completion of a page erase, or if no SPM instruction is executed within four clock cycles. The CPU is halted during the entire page write operation if the NRWW section is addressed.

- **Bit 0 – SPMEN: Store Program Memory Enable**

This bit enables the SPM instruction for the next four clock cycles. If written to one together with either RWWSRE, BLBSET, PGWRT' or PGERS, the following SPM instruction will have a special meaning, see description above. If only SPMEN is written, the following SPM instruction will store the value in R1:R0 in the temporary page buffer addressed by the Z-pointer. The LSB of the Z-pointer is ignored. The SPMEN bit will auto-clear upon completion of an SPM instruction, or if no SPM instruction is executed within four clock cycles. During page erase and page write, the SPMEN bit remains high until the operation is completed.

Writing any other combination than “10001”, “01001”, “00101”, “00011” or “00001” in the lower five bits will have no effect.

Addressing the Flash

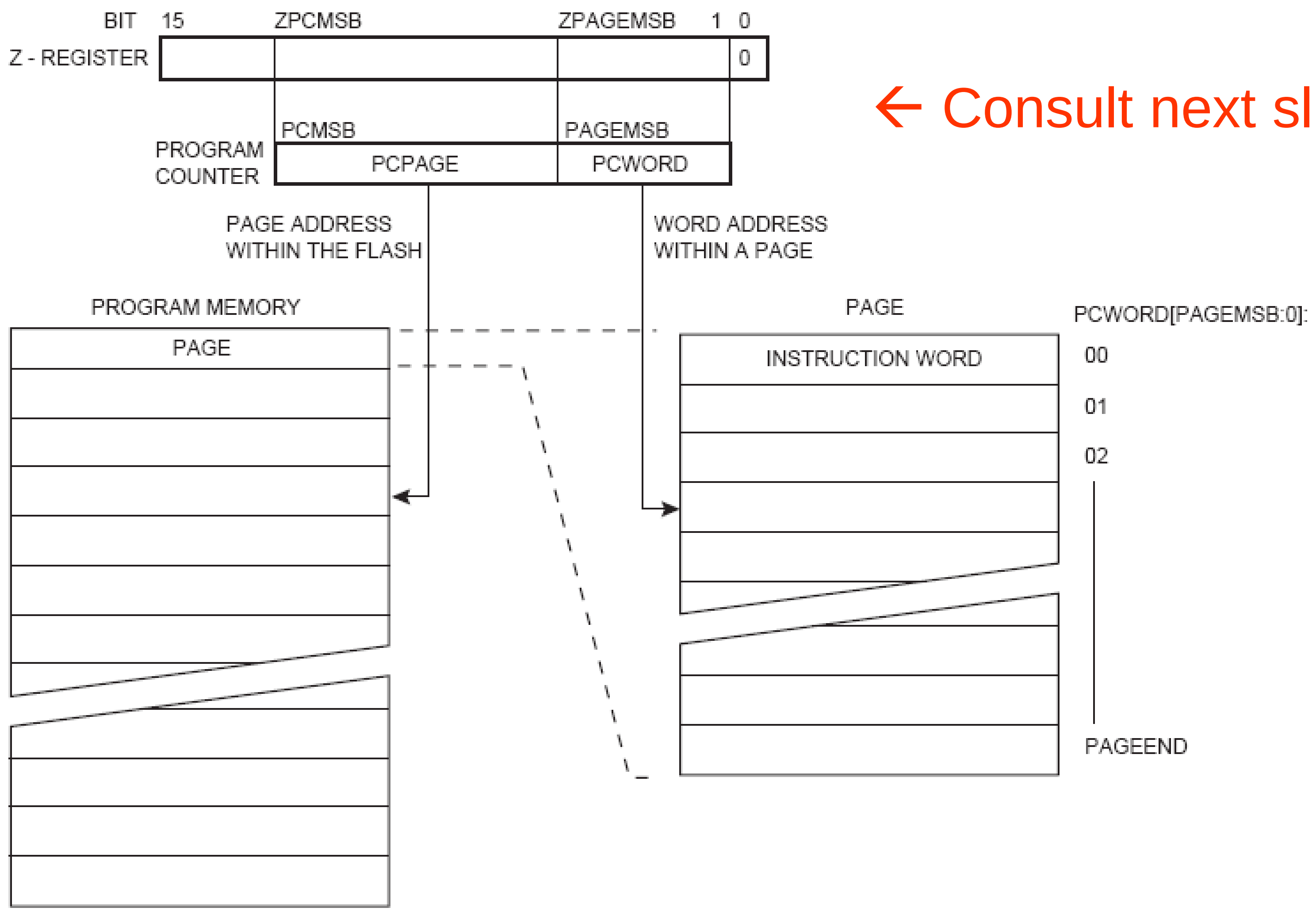
The Z-pointer is used to address the SPM commands.

Bit	15	14	13	12	11	10	9	8
ZH (R31)	Z15	Z14	Z13	Z12	Z11	Z10	Z9	Z8
ZL (R30)	Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0
	7	6	5	4	3	2	1	0

Since the Flash is organized in pages (see Table 111 on page 263), the Program Counter can be treated as having two different sections. One section, consisting of the least significant bits, is addressing the words within a page, while the most significant bits are addressing the pages. This is shown in Figure 126. Note that the Page Erase and Page Write operations are addressed independently. Therefore it is of major importance that the Boot Loader software addresses the same page in both the Page Erase and Page Write operation. Once a programming operation is initiated, the address is latched and the Z-pointer can be used for other operations.

The only SPM operation that does not use the Z-pointer is Setting the Boot Loader Lock bits. The content of the Z-pointer is ignored and will have no effect on the operation. The LPM instruction does also use the Z pointer to store the address. Since this instruction addresses the Flash byte by byte, also the LSB (bit Z0) of the Z-pointer is used.

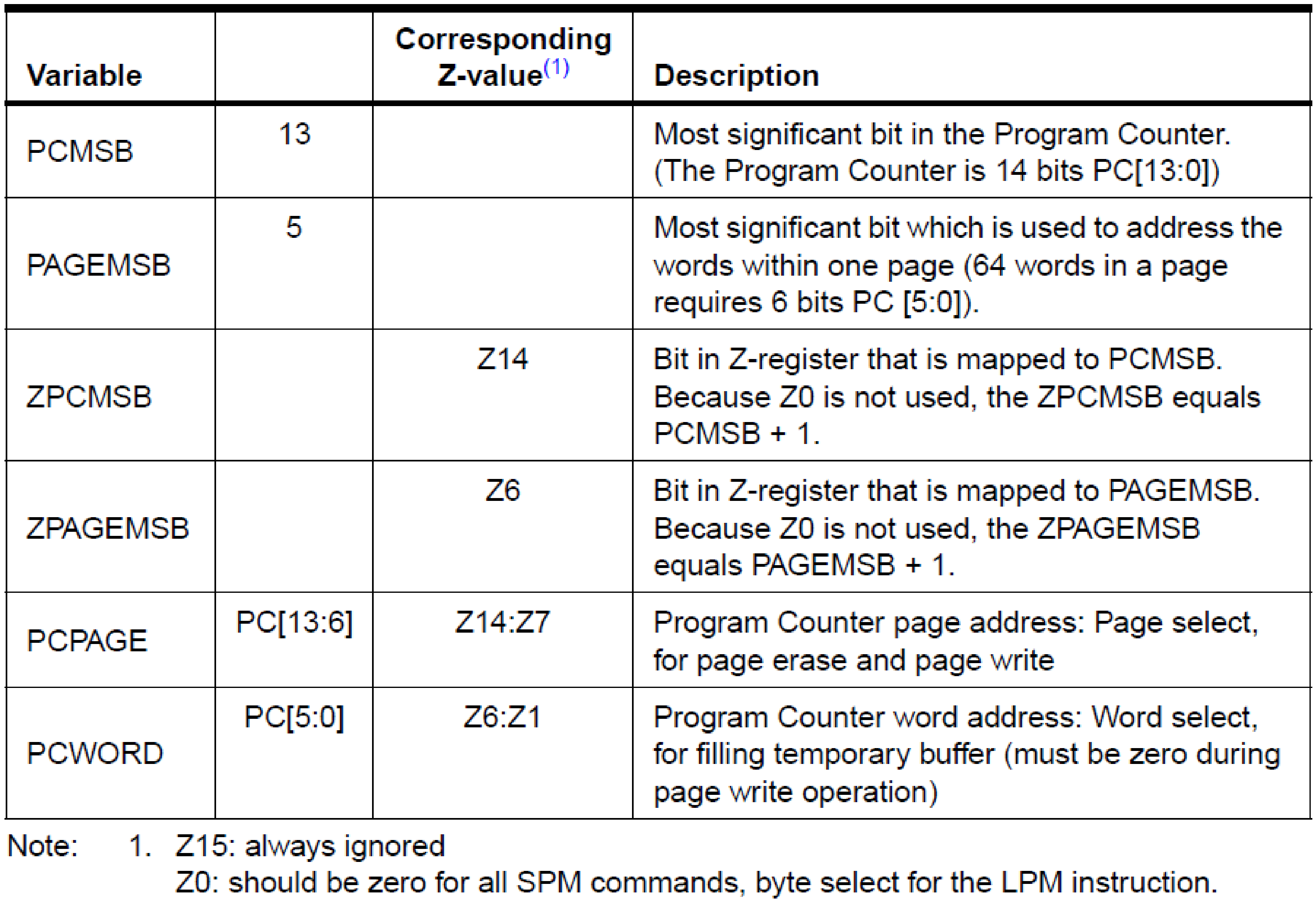
Addressing the Flash (Mega32)



← Consult next slide !

Notes: 1. The different variables used in Figure 126 are listed in Table 102 on page 257.
2. PCPAGE and PCWORD are listed in Table 111 on page 263.

Addressing the Flash (Former Slide)



Variable		Corresponding Z-value[1]	Description
PCMSB	13		Most significant bit in the Program Counter. (The Program Counter is 14 bits PC[13:0])
PAGEMSB	5		Most significant bit which is used to address the words within one page (64 words in a page requires 6 bits PC [5:0]).
ZPCMSB		Z14	Bit in Z-register that is mapped to PCMSB. Because Z0 is not used, the ZPCMSB equals PCMSB + 1.
ZPAGEMSB		Z6	Bit in Z-register that is mapped to PAGEMSB. Because Z0 is not used, the ZPAGEMSB equals PAGEMSB + 1.
PCPAGE	PC[13:6]	Z14:Z7	Program Counter page address: Page select, for page erase and page write
PCWORD	PC[5:0]	Z6:Z1	Program Counter word address: Word select, for filling temporary buffer (must be zero during page write operation)

Note: 1. Z15: always ignored
Z0: should be zero for all SPM commands, byte select for the LPM instruction.

Flash Programming Time

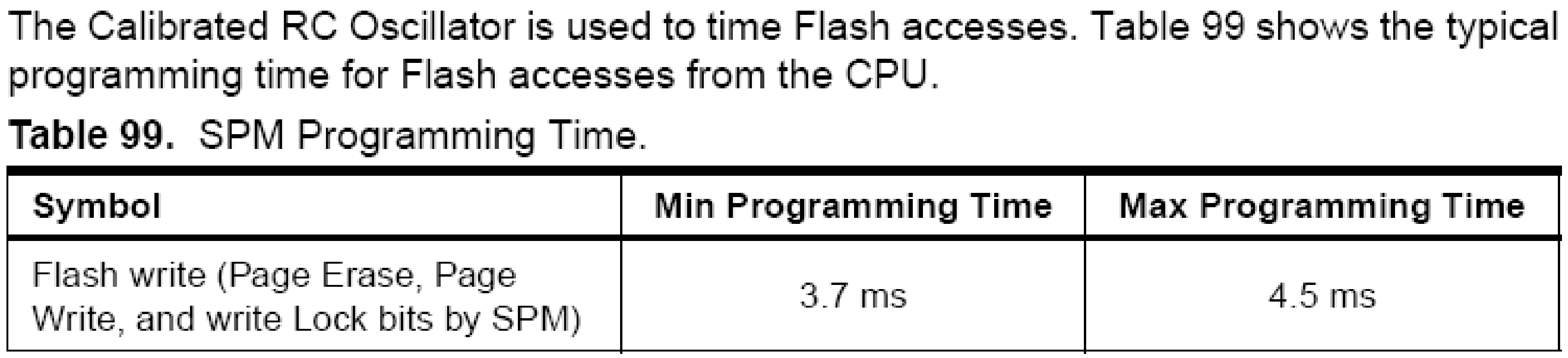
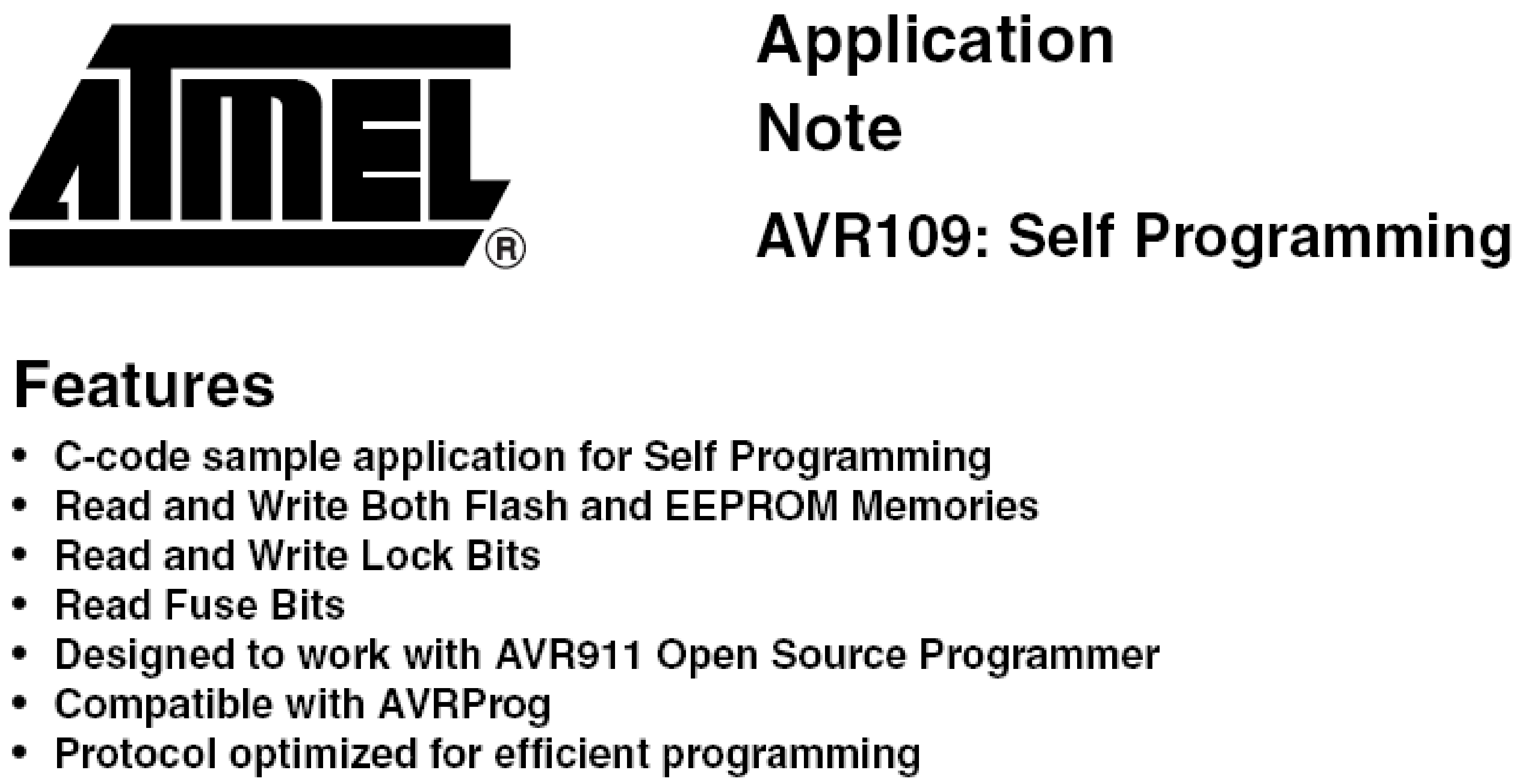

The Calibrated RC Oscillator is used to time Flash accesses. Table 99 shows the typical programming time for Flash accesses from the CPU.

Table 99. SPM Programming Time.

Symbol	Min Programming Time	Max Programming Time
Flash write (Page Erase, Page Write, and write Lock bits by SPM)	3.7 ms	4.5 ms

AVR109: Self Programming



Application Note

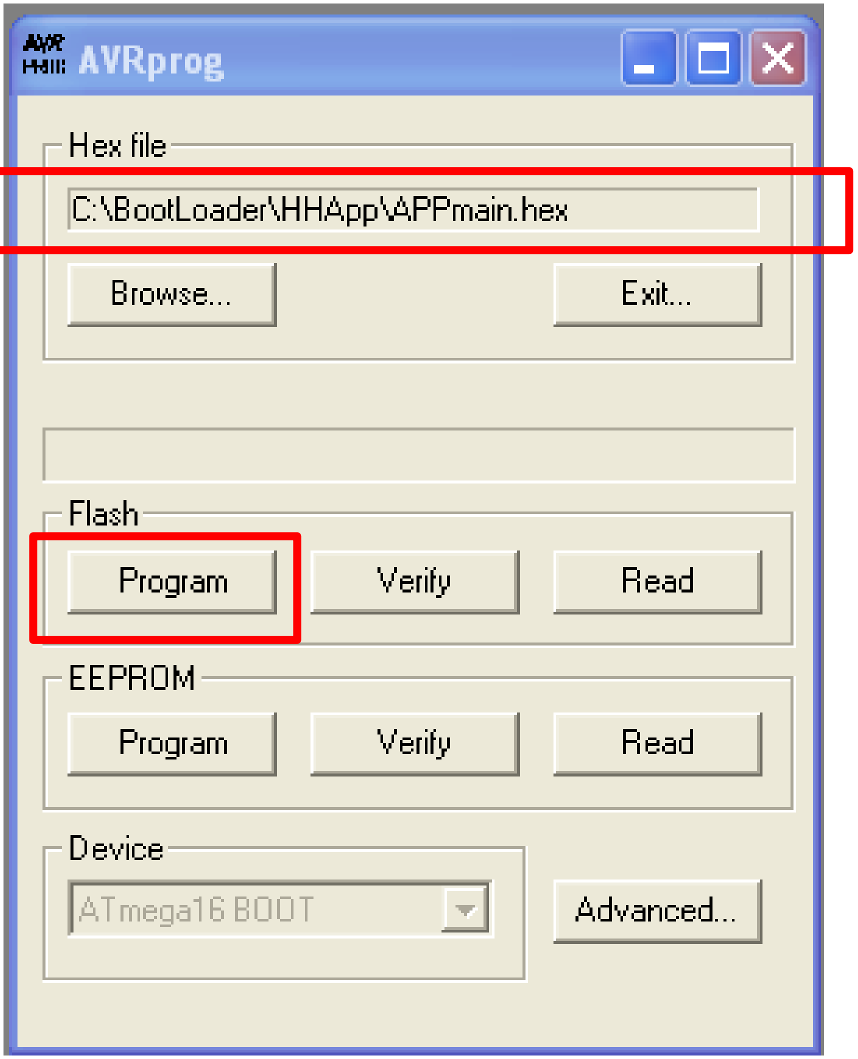
AVR109: Self Programming

Features

- C-code sample application for Self Programming
- Read and Write Both Flash and EEPROM Memories
- Read and Write Lock Bits
- Read Fuse Bits
- Designed to work with AVR911 Open Source Programmer
- Compatible with AVRProg
- Protocol optimized for efficient programming

Available at AMS Blackboard

AVR Prog (version 1.40)



Hex file: C:\BootLoader\HHApp\APPmain.hex

Device: ATmega16 BOOT

IMPORTANT:
Use only COM ports:

COM1
COM2
COM3
COM4

Problems?
Try COM setting: ”FIFO” = 1.

The ”AVR Prog” is available at AMS Blackboard.

”AVR Prog”, Protokol (1 of 2)

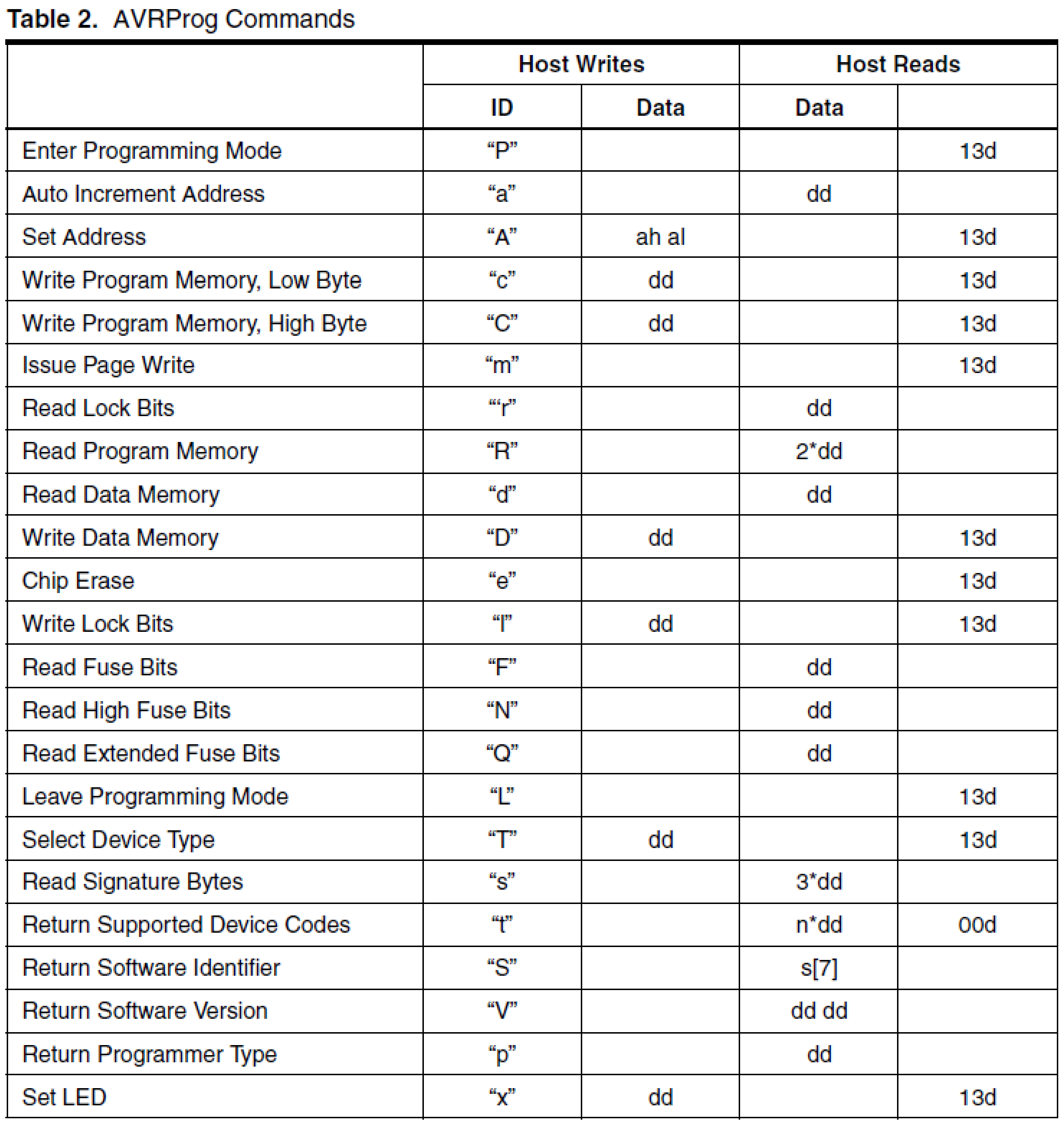


Table 2. AVRProg Commands

	Host Writes		Host Reads	
	ID	**Data**	**Data**	
Enter Programming Mode	“P”			13d
Auto Increment Address	“a”		dd	
Set Address	“A”	ah al		13d
Write Program Memory, Low Byte	“c”	dd		13d
Write Program Memory, High Byte	“C”	dd		13d
Issue Page Write	“m”			13d
Read Lock Bits	“r”		dd	
Read Program Memory	“R”		2*dd	
Read Data Memory	“d”		dd	
Write Data Memory	“D”	dd		13d
Chip Erase	“e”			13d
Write Lock Bits	“l”	dd		13d
Read Fuse Bits	“F”		dd	
Read High Fuse Bits	“N”		dd	
Read Extended Fuse Bits	“Q”		dd	
Leave Programming Mode	“L”			13d
Select Device Type	“T”	dd		13d
Read Signature Bytes	“s”		3*dd	
Return Supported Device Codes	“t”		n*dd	00d
Return Software Identifier	“S”		s[7]	
Return Software Version	“V”		dd dd	
Return Programmer Type	“p”		dd	
Set LED	“x”	dd		13d

”AVR Prog”, Protokol (2 of 2)

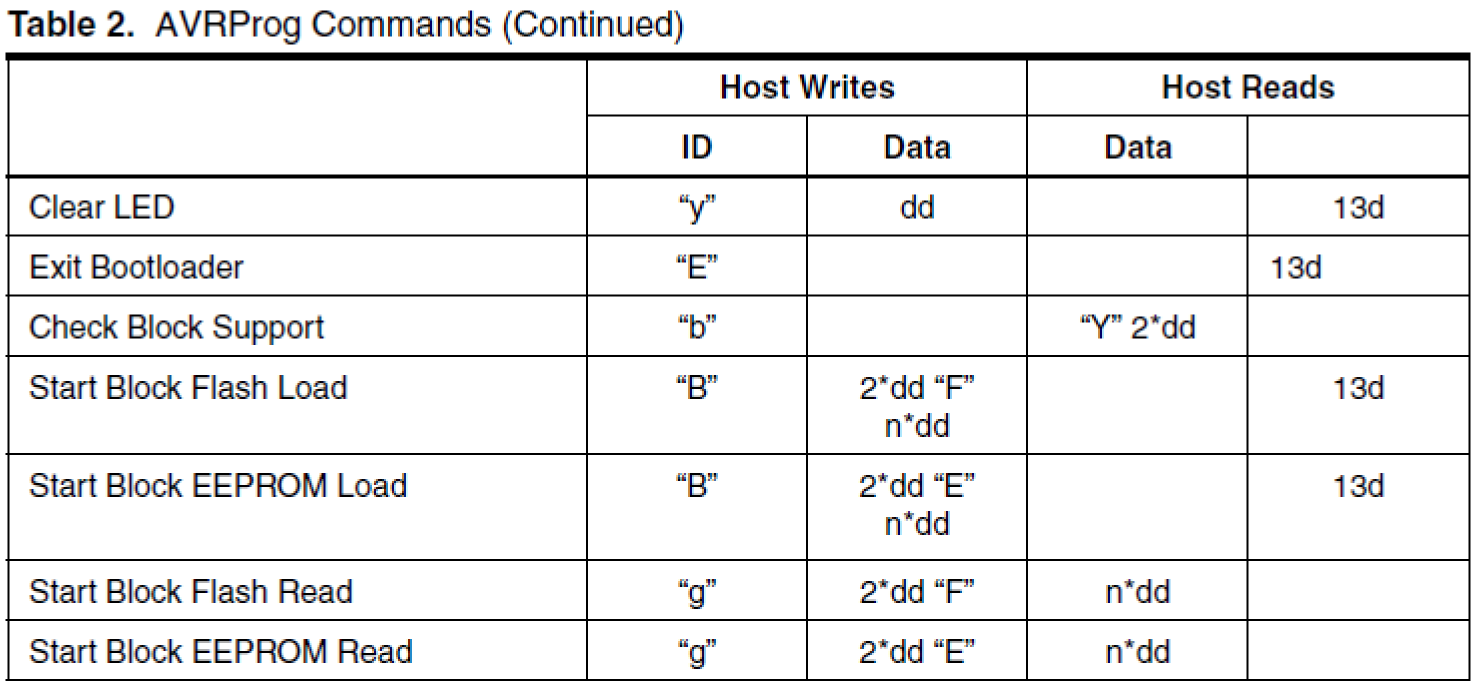
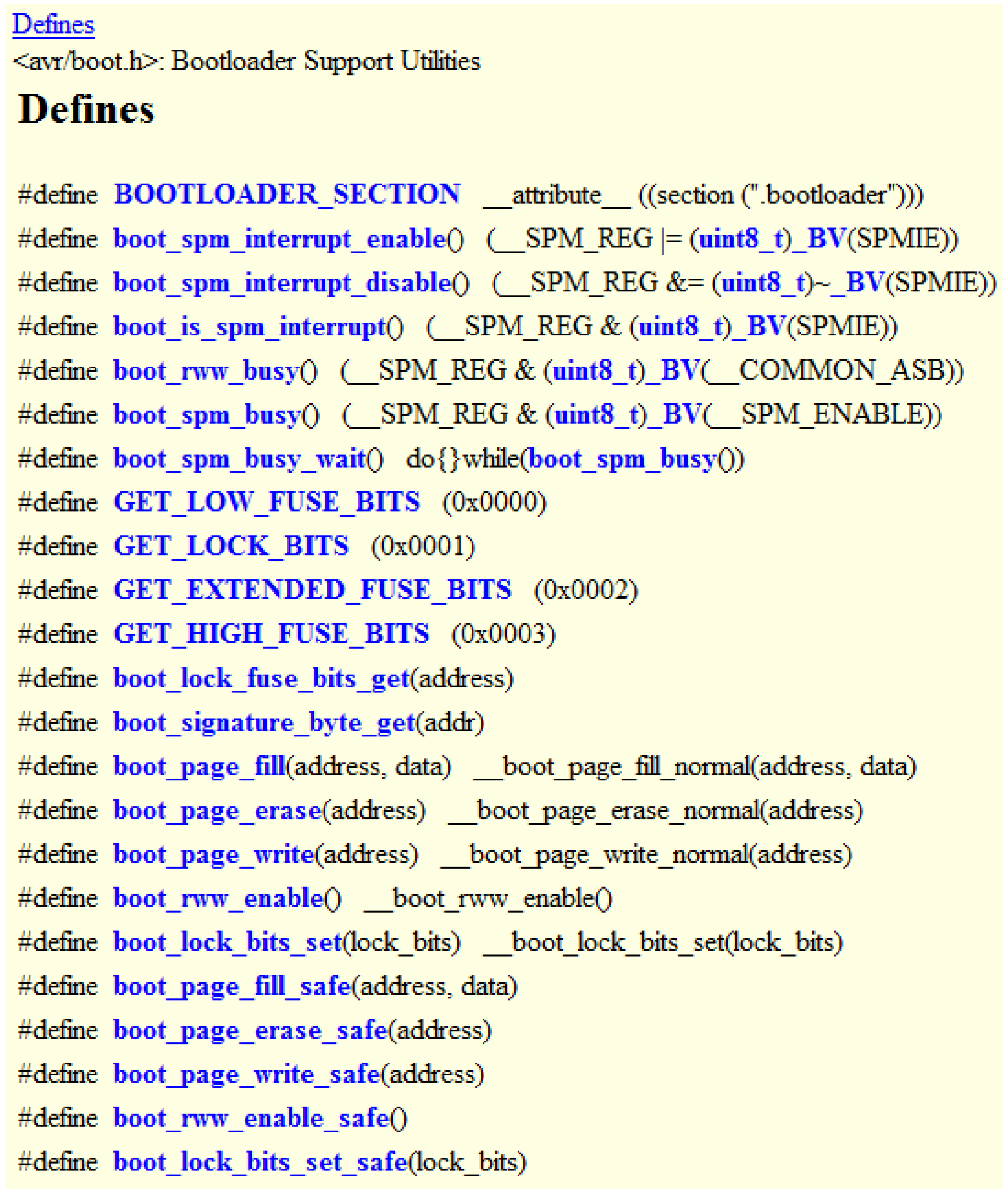


Table 2. AVRProg Commands (Continued)

	Host Writes		Host Reads	
	ID	**Data**	**Data**	
Clear LED	“y”	dd		13d
Exit Bootloader	“E”			13d
Check Block Support	“b”		“Y” 2*dd	
Start Block Flash Load	“B”	2*dd “F” n*dd		13d
Start Block EEPROM Load	“B”	2*dd “E” n*dd		13d
Start Block Flash Read	“g”	2*dd “F”	n*dd	
Start Block EEPROM Read	“g”	2*dd “E”	n*dd	

More informations:
See the AVR109 application note and/or the ”AVR Prog” manual.

AVR libc: Bootloader Support Utilities



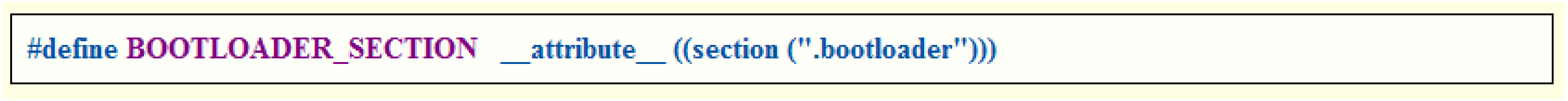
Defines

<avr/boot.h>: Bootloader Support Utilities

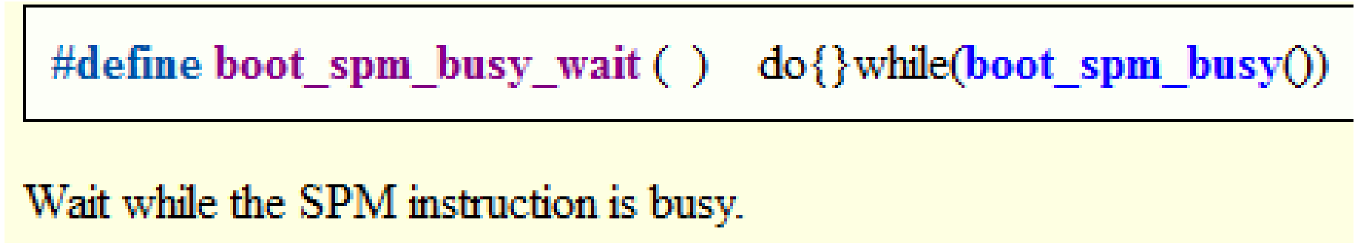
Defines

```
#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)
```

<avr/boot.h>

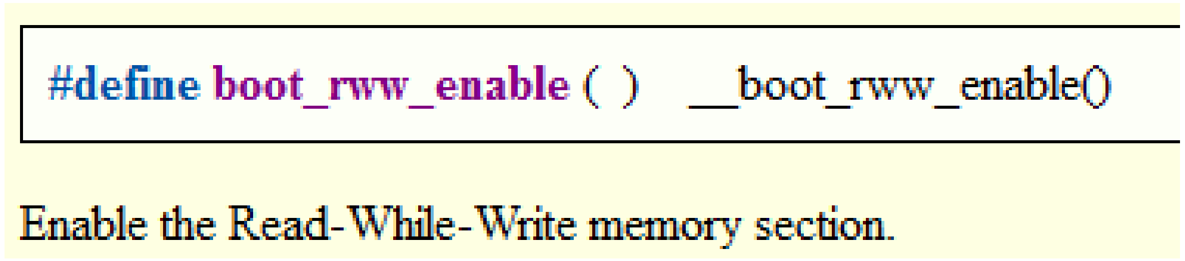


```
#define BOOTLOADER_SECTION   __attribute__ ((section (".bootloader")))
```



```
#define boot_spm_busy_wait ( )   do{}while(boot_spm_busy())
```

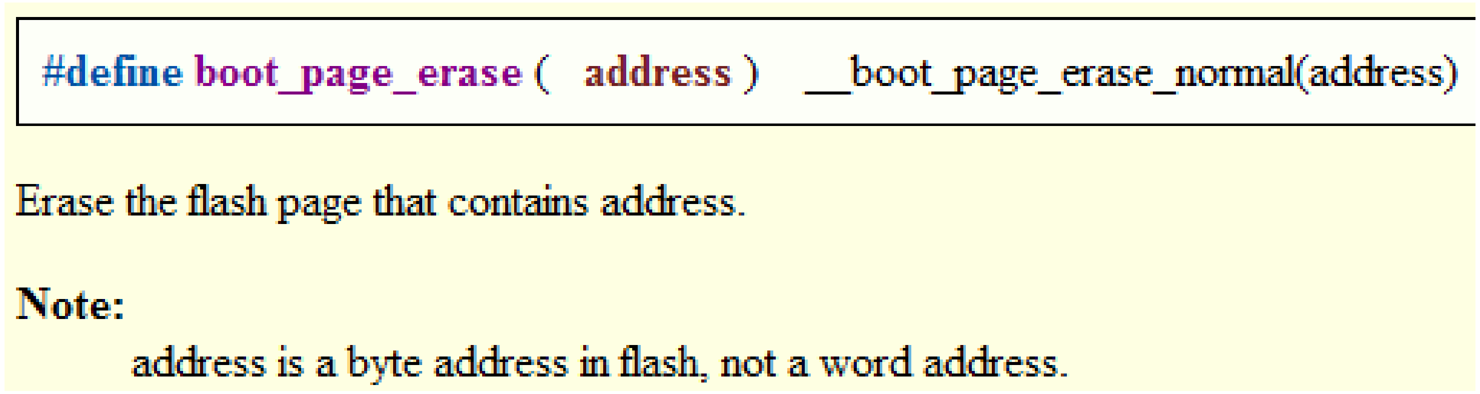
Wait while the SPM instruction is busy.



```
#define boot_rww_enable ( )   __boot_rww_enable()
```

Enable the Read-While-Write memory section.

<avr/boot.h>

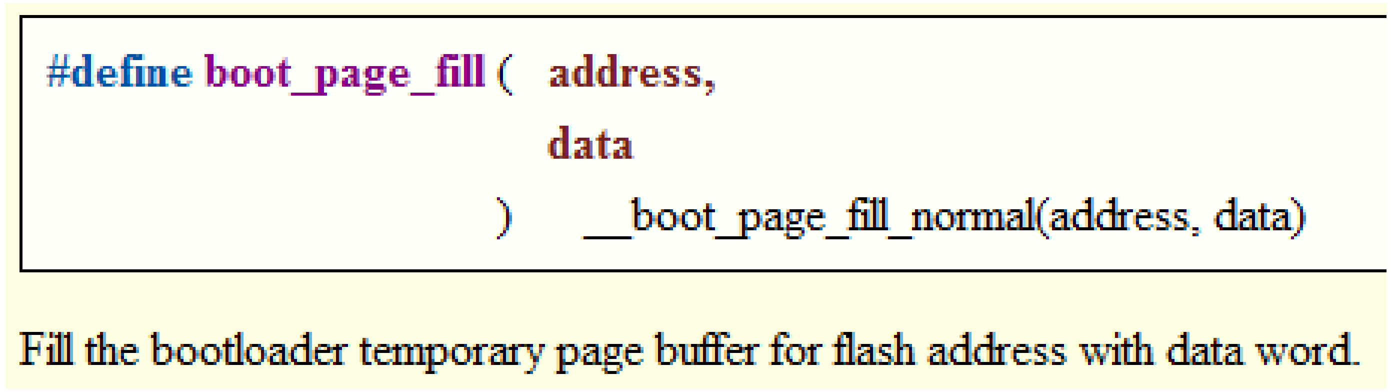


```
#define boot_page_erase (   address )   __boot_page_erase_normal(address)
```

Erase the flash page that contains address.

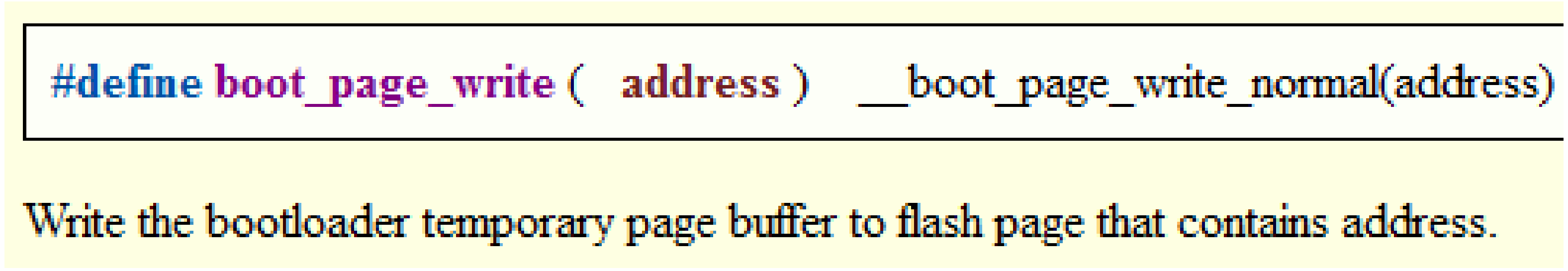
Note:

address is a byte address in flash, not a word address.



```
#define boot_page_fill (   address,
                           data
                         )     __boot_page_fill_normal(address, data)
```

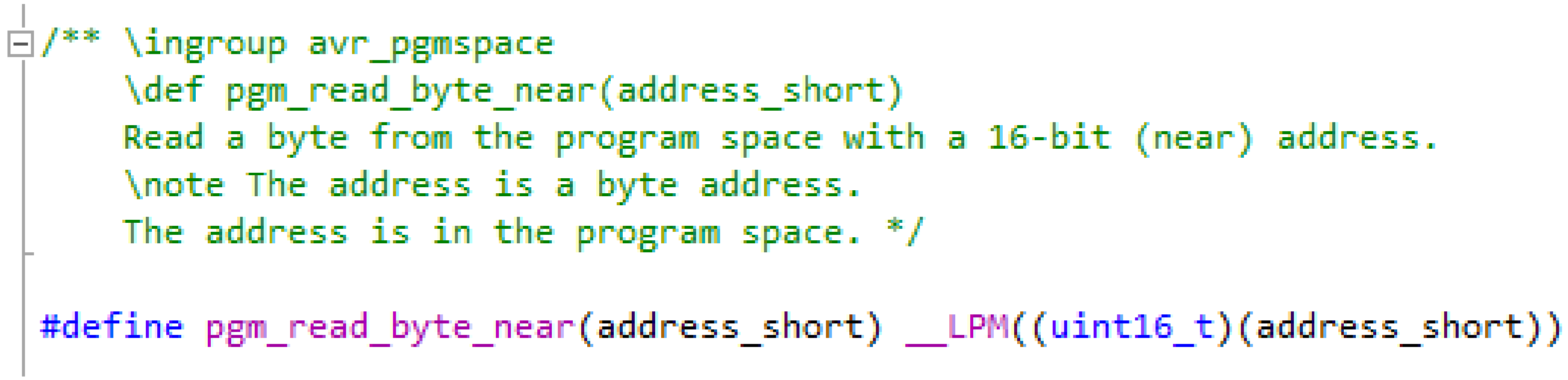
Fill the bootloader temporary page buffer for flash address with data word.



```
#define boot_page_write (   address )   __boot_page_write_normal(address)
```

Write the bootloader temporary page buffer to flash page that contains address.

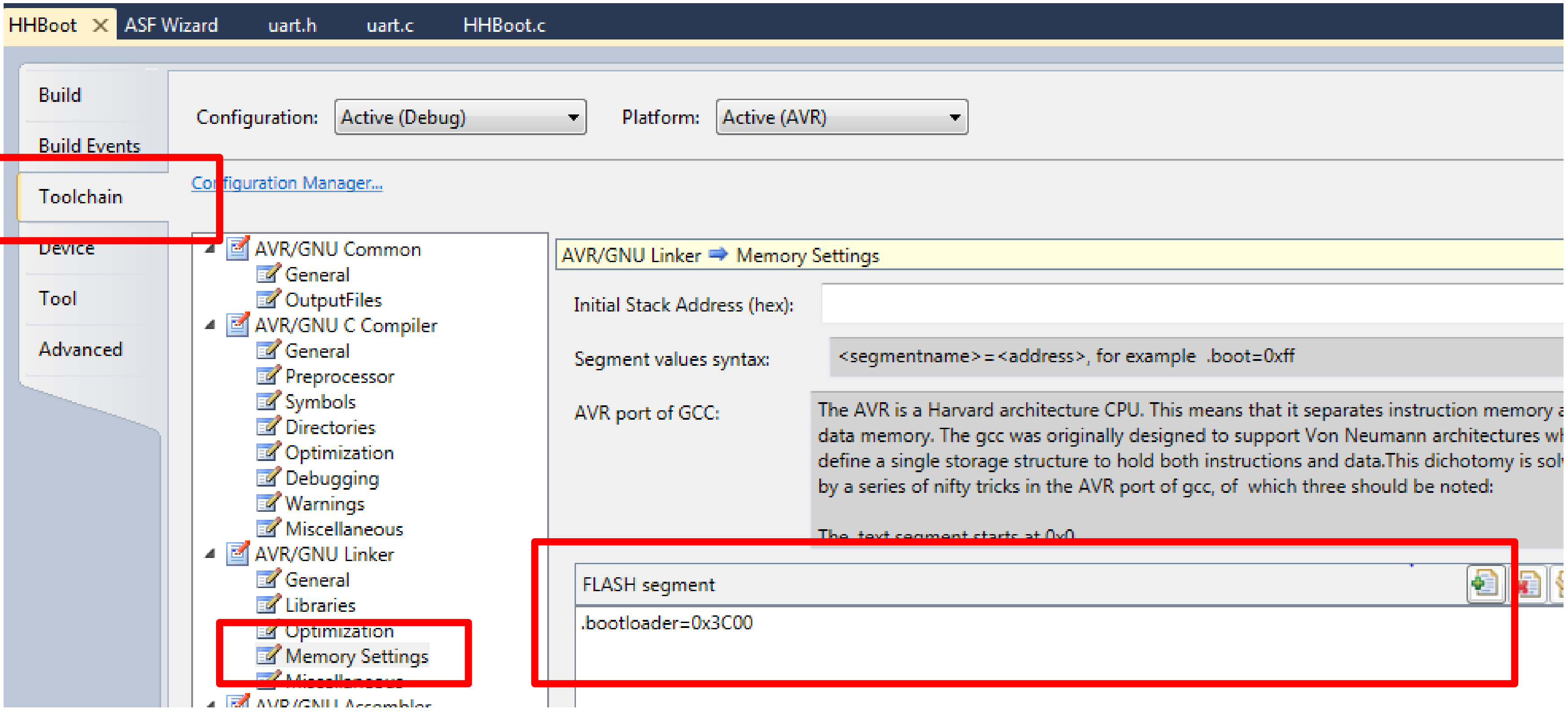
<avr/pgmspace.h>



```
/** \ingroup avr_pgmspace
    \def pgm_read_byte_near(address_short)
    Read a byte from the program space with a 16-bit (near) address.
    \note The address is a byte address.
    The address is in the program space. */

#define pgm_read_byte_near(address_short) __LPM((uint16_t)(address_short))
```

Locating the bootloader code



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

AVR/GNU Linker ➡ Memory Settings

Initial Stack Address (hex):

Segment values syntax: <segmentname>=<address>, for example .boot=0xff

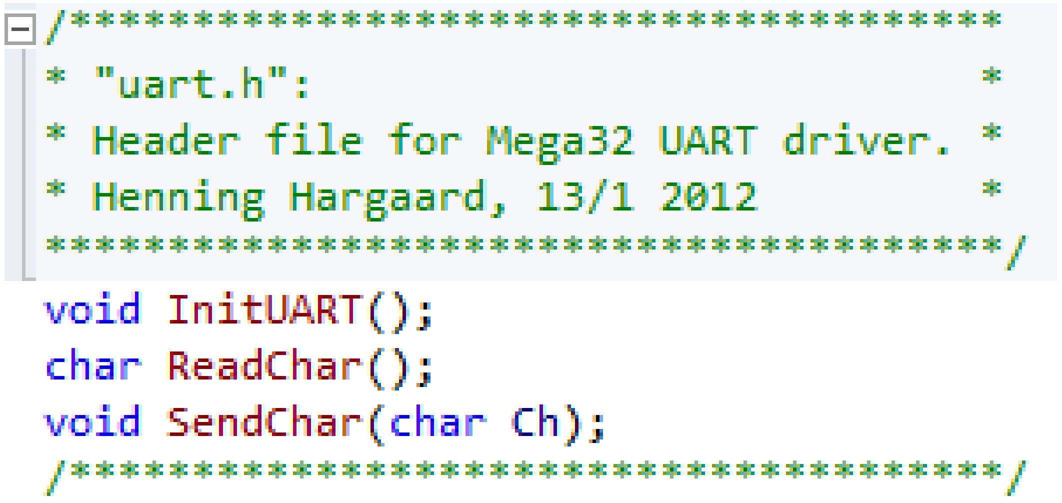
AVR port of GCC: The AVR is a Harvard architecture CPU. This means that it separates instruction memory a data memory. The gcc was originally designed to support Von Neumann architectures wh define a single storage structure to hold both instructions and data.This dichotomy is sol by a series of nifty tricks in the AVR port of gcc, of which three should be noted:

The .text segment starts at 0x0

FLASH segment

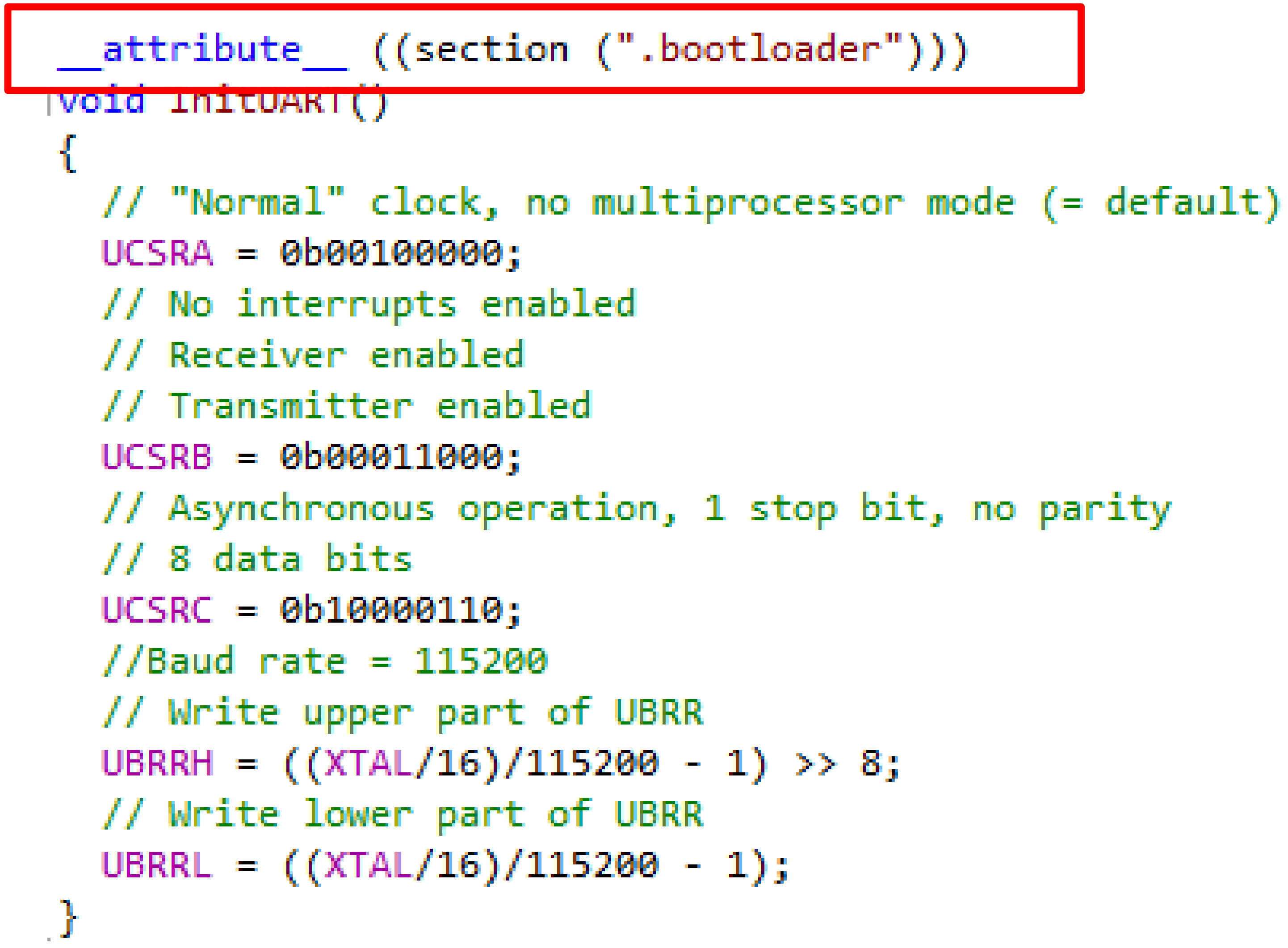
```
.bootloader=0x3C00
```

Minimum UART driver (header file)



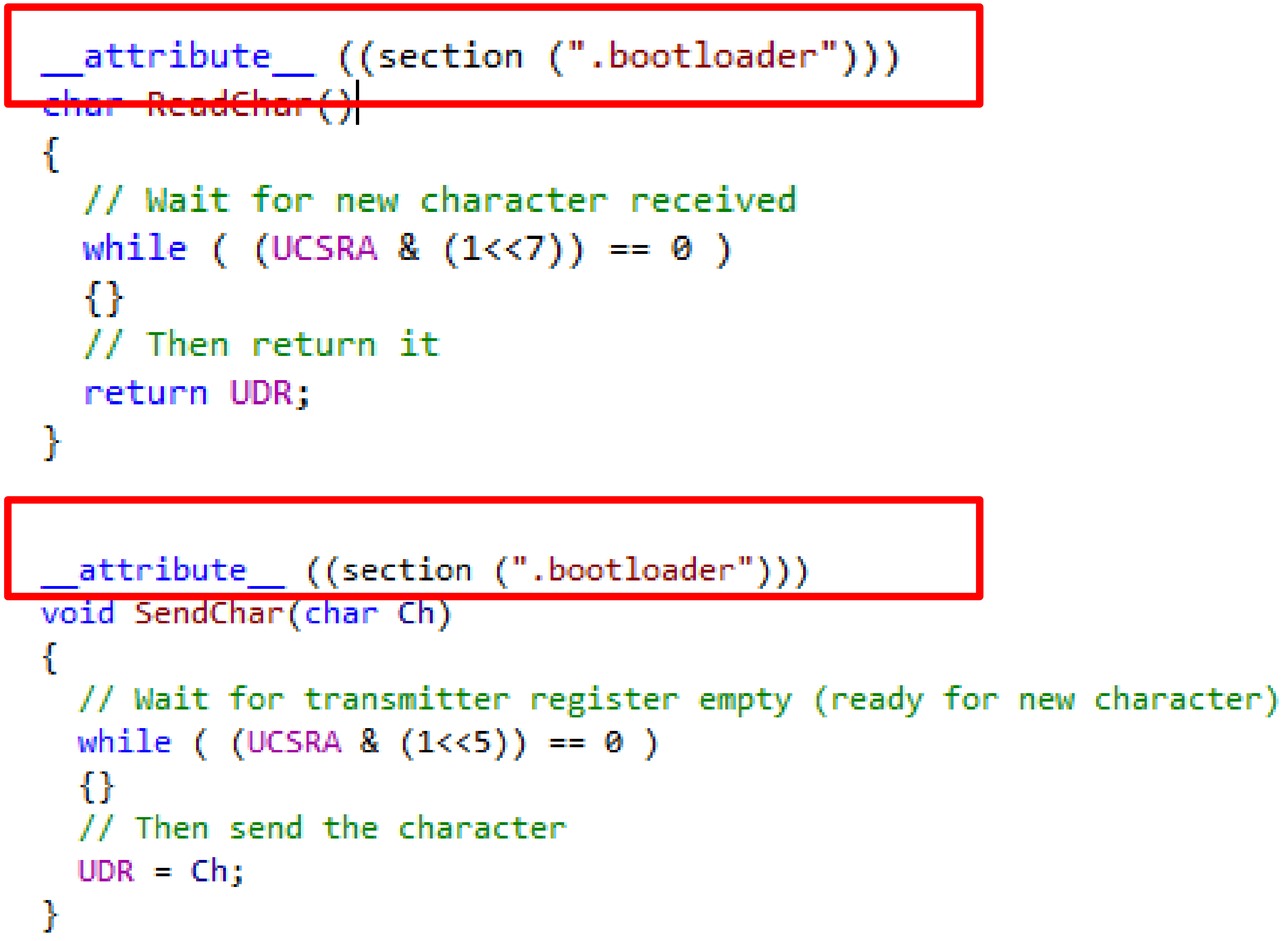
```
/***************************************
* "uart.h":                             *
* Header file for Mega32 UART driver.  *
* Henning Hargaard, 13/1 2012          *
****************************************/
void InitUART();
char ReadChar();
void SendChar(char Ch);
/***************************************/
```

Locating the UART driver



```
__attribute__ ((section (".bootloader")))
void InitUART()
{
  // "Normal" clock, no multiprocessor mode (= default)
  UCSRA = 0b00100000;
  // No interrupts enabled
  // Receiver enabled
  // Transmitter enabled
  UCSRB = 0b00011000;
  // Asynchronous operation, 1 stop bit, no parity
  // 8 data bits
  UCSRC = 0b10000110;
  //Baud rate = 115200
  // Write upper part of UBRR
  UBRRH = ((XTAL/16)/115200 - 1) >> 8;
  // Write lower part of UBRR
  UBRRL = ((XTAL/16)/115200 - 1);
}
```

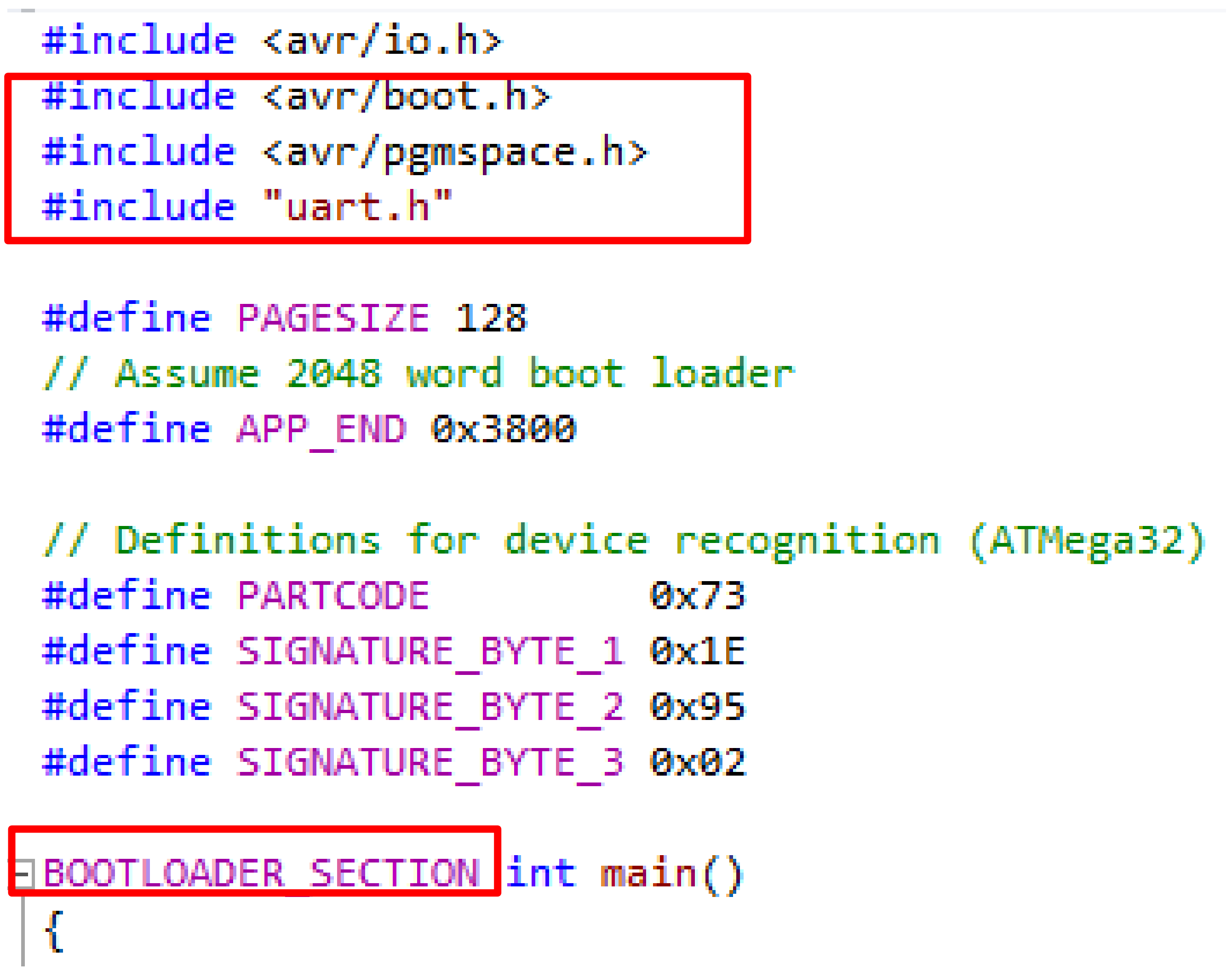
Locating the UART driver



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


__attribute__ ((section (".bootloader")))
void SendChar(char Ch)
{
  // Wait for transmitter register empty (ready for new character)
  while ( (UCSRA & (1<<5)) == 0 )
  {}
  // Then send the character
  UDR = Ch;
}
```

Locating the Boot Loader main()



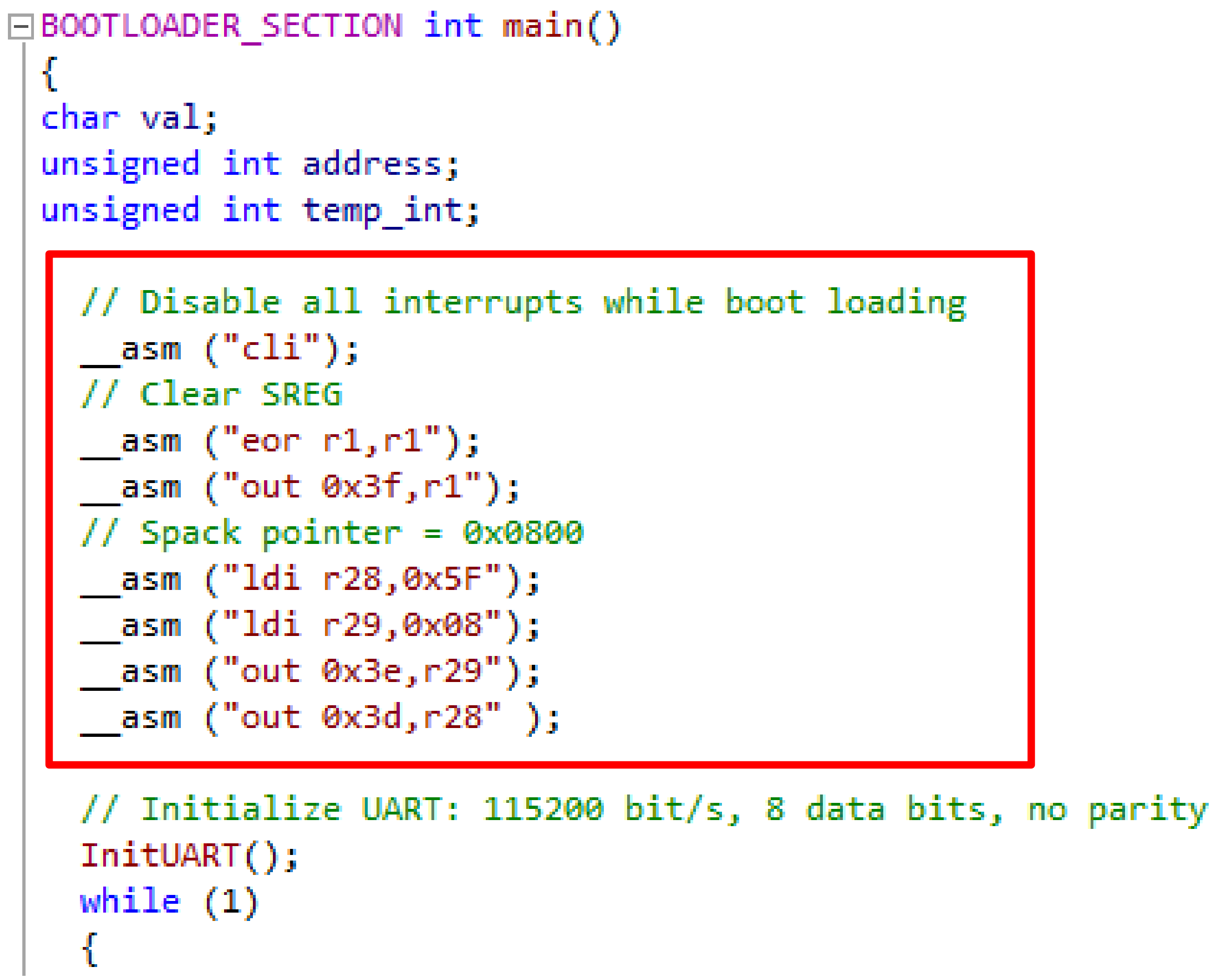
```
#include <avr/io.h>
#include <avr/boot.h>
#include <avr/pgmspace.h>
#include "uart.h"

#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()
{
```

Boot Loader basic init code



```
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)
  {
```

Common AVR protokols

- STK500, version 1
  Used by many Arduino Bootloaders.

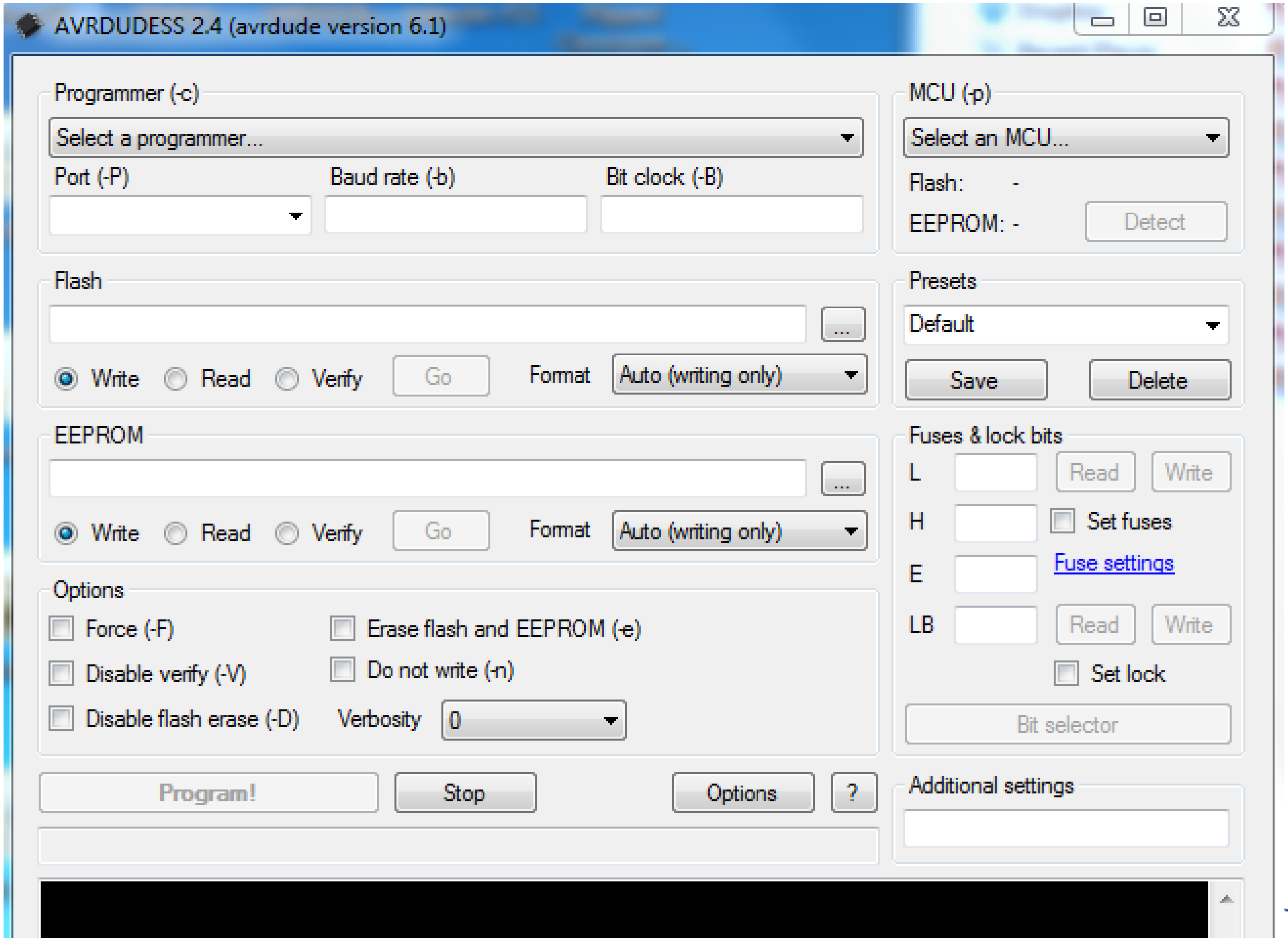
- STK500, version 2
  Used by Arduino Mega2560 + other Arduino board Bootloaders.

- Protokol desciption can be found at AMS Blackboard.

Programmer: AVRdude

- Great PROGRAM for interfacing MANY programmers and Bootloaders with MANY protokols and MANY AVR controllers.
- Open source project.
- Command line driven, but a good version with Windows GUI is available at AMS Blackboard.

AVRdude + Windows GUI



AVRDUDESS 2.4 (avrdude version 6.1)

Programmer (-c): Select a programmer...

Port (-P) | Baud rate (-b) | Bit clock (-B)

MCU (-p): Select an MCU...

Flash: -

EEPROM: -

Flash: [✓ Write] Read Verify — Format: Auto (writing only)

EEPROM: [✓ Write] Read Verify — Format: Auto (writing only)

Presets: Default

Options:
- Force (-F)
- Disable verify (-V)
- Disable flash erase (-D)
- Erase flash and EEPROM (-e)
- Do not write (-n)
- Verbosity: 0

Fuses & lock bits: L, H, E, LB — Set fuses — Set lock — Fuse settings

Additional settings

End of lesson 4