AVR109: Self-programming

8-bit AVR® RISC Microcontroller

Application Note

Features

- **AVR109 Code Fits in All AVR® Microcontrollers with Boot Block**
- **Read and Write Both Flash and EEPROM Memories**
- **Uses the AVRProg Protocol**
- **Read and Write Lock Bits**

Introduction

This application note describes how an AVR with the Store Program Memory (SPM) instruction can be configured for Self-programming. The AVR communicates via the UART with a PC running the AVRprog programming software. This enables Flash and EEPROM programming without the need for an external programmer.

A Boot Loader program is placed inside the Boot Section of the Flash memory. This program handles communication with the host PC, and facilitates programming of both Flash and EEPROM. Once programmed, different levels of protection can be individually applied to both the boot and application portion of the Flash memory. The AVR thus offers a unique flexibility, allowing the user extensive degrees of memory protection.

SPM Explained

To get a better understanding of the AVRs' Self-programming capabilities, the basics of this feature are explained below.

Memory Organization

The Flash memory is divided into two sections, one Application section and one Boot Loader section. The Application section contains the main code for the application, while the Boot Loader section contains the code for the actual Self-programming. The SPM instruction can only be executed from the Boot Loader section. (Note: The Boot Loader section can also be used for ordinary application code.)

The Flash memory is divided into pages containing 32, 64, or 128 words each. The usage of pages is explained later. The entire memory span, both Application and Boot Loader sections, is divided into pages. For instance, a device with 8 kbytes of Flash and page size of 32 words (64 bytes) will therefore have a total of 128 pages. The memory organization is shown in Figure 1.

Rev. 1644D–AVR–1/03

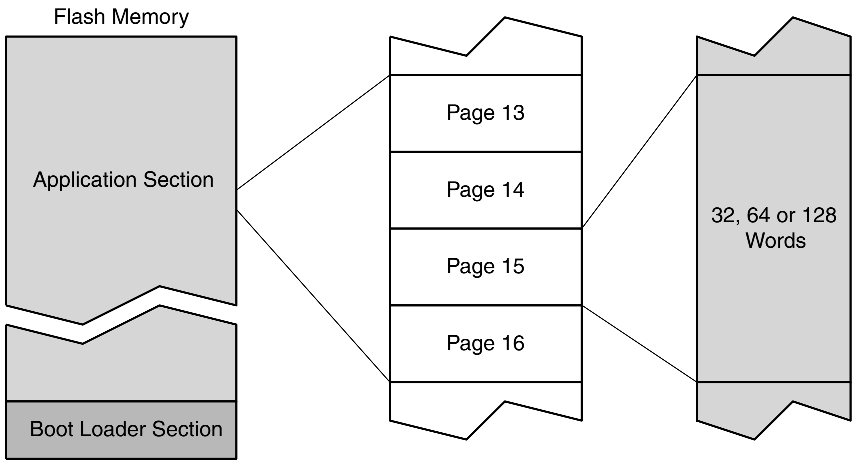


Figure 1. Memory Organization

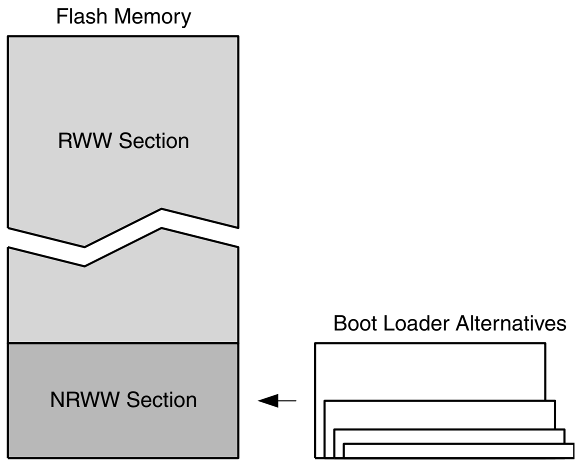
The size of the Boot Loader section can be selected using the two BOOTSZx Fuses. The fuses select one of four predefined sizes. The BOOTSZx Fuses can be changed using Serial or Parallel Programming. Refer to the devices' data sheet for details.

If a Boot Loader is implemented, it can be called either directly from the Application code using calls or jumps, or by programming the BOOTRST Fuse. When the BOOTRST Fuse is programmed, the CPU will start execution in the Boot Loader section on Reset, instead of starting at address 0. The BOOTRST Fuse can be changed using Serial or Parallel Programming.

Read-While-Write Capabilities

In addition to the selectable division between the application and Boot Loader sections, the Flash is also divided into two fixed-size sections. The first section is the Read-While-Write (RWW) section. The second is the No-Read-While-Write (NRWW) section. The NRWW section size always equals the largest selectable Boot Loader section size, thus the Boot Loader section occupies all or part of the NRWW section. This is illustrated in Figure 2.

Figure 2. RWW and NRWW Sections



The difference between the sections is that the NRWW section is accessible while updating the RWW section. It is not possible to access the RWW section when it's being updated. When the NRWW is updated (e.g., updating the Boot Loader code itself), the

CPU is halted during the whole operation. In other words, No-Read-While-Writing to the NRWW section, but possible to Read-While-Writing to the RWW section. Refer to the devices' data sheet for details.

This functionality makes it possible to continue execution of critical code while updating the RWW section. Note that this critical code must be contained within the NRWW section (not necessarily in the Boot Loader section). See the section on interrupts below for more information.

The ATmega163 and ATmega323 devices don't have NRWW and RWW sections, only the selectable division into application and Boot Loader sections. Any updates to Flash memory on these devices halt the CPU during the whole operation.

Using the SPM Instruction

All Self-programming operations are performed using the SPM instruction. The operation is selected using the SPMCR Register. 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.

The SPM functions are described below.

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.

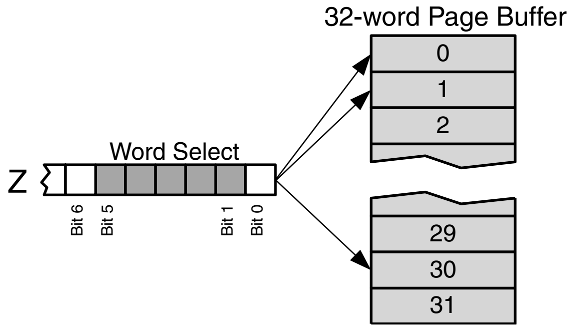
To erase a page, set the PGERS and SPMEN bits in the SPMCR Register and execute the SPM instruction.

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

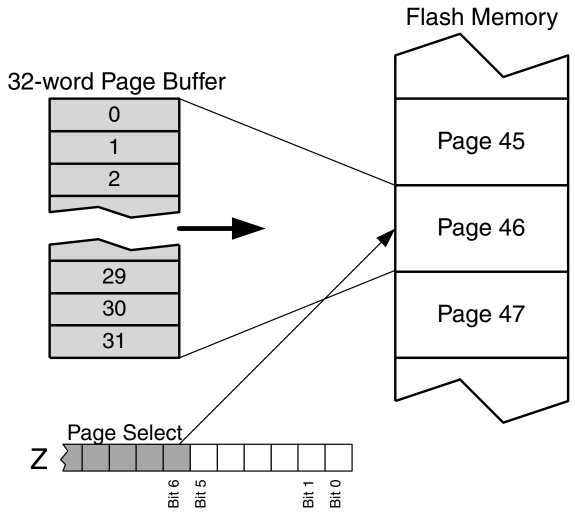


To write a word to the Page Buffer, load the word into the R1:R0 Registers. Set the Z-register to point to the correct word and set only the SPMEN bit in the SPMCR Register. The SPM instruction must then be executed within four cycles.

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 PGERS 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.

The Boot Lock Bits

The application and Boot Loader section can be protected on different levels. There are four levels of protection for both sections. A short description of the modes follows.

Table 1. Boot Lock Modes

Mode	Bits	Description
Mode 1	11	Full read/write access
Mode 2	10	No write access
Mode 3	00	No write access and no read access (data or interrupt execution) from the other section.
Mode 4	01	No read access (data or interrupt execution) from the other section.

Note that once programmed (cleared), it is impossible to unprogram the bits again without using serial or parallel programming. For instance, to implement an application that is to be updated once, set Boot Lock mode 1 on the Application section, and mode 4 on the Boot Loader section. This prevents the application from accessing the Boot Loader, while giving the Boot Loader full access to update the application section. Once updated, the Boot Loader would set mode 3 on the Application section, thus blocking all further access.

To program the Boot Lock bits, load the R0 Register with the correct bits, set the BLBSET and SPMEN bits in the SPMCR Register and execute the SPM instruction within four cycles. The contents of the Z-register are ignored.

Using the LPM instruction instead of the SPM instruction will read the bits.

Interrupt Considerations

It is possible to use interrupts while writing to the RWW section, but the software must prevent any other access to the RWW section. In other words, interrupt service routines to be executed while updating the RWW section must be placed in the NRWW section, including the Interrupt Vectors.

Using the IVSEL bit in the GICR Register, the application can be used to implement two separate Interrupt Vector tables. One in the Application section, and one in the Boot Loader section to be used when updating the RWW section. This enables the application to continue critical processes, e.g., safety monitoring, during Self-programming. Refer to the devices' data sheet for more details on interrupts and the IVSEL Flag.

If the secondary Interrupt Vectors are not used, the interrupts must be disabled during RWW section updates.

The SPM Interrupt

On all devices supporting Self-programming, except the ATmega163 and ATmega323 devices, it is possible to control the Flash update operations using interrupts. Setting the SPMIE bit in the SPMCR Register will enable the SPM ready interrupt. This can be used to indicate when the current SPM operation is finished.

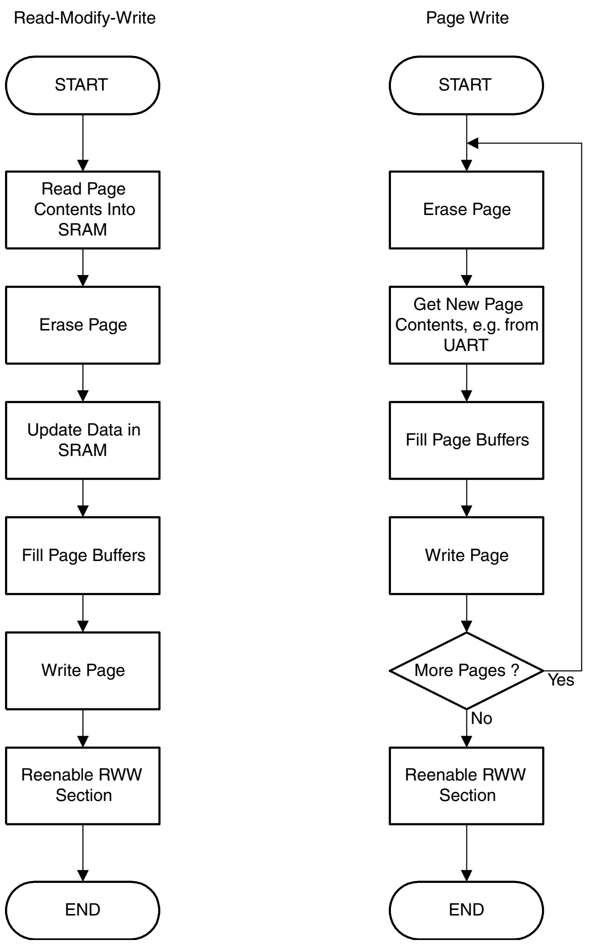
EEPROM Conflicts

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

Two common update procedures are shown in Figure 6. The flowchart to the left describes a Read-Modify-Write operation used to update small parts of the Flash, e.g., a constant string contained in Flash memory. The flowchart to the right describes a Page Write operation used to write whole pages without reading previous contents, e.g., write data received from an UART.

Figure 6. Typical Update Flowcharts



Boot Loader Example

The Boot Loader software presented in this application note uses AVRprog (available from www.atmel.com) as the user interface. The example application implements functions to read or update the Flash and EEPROM memories on the target device. It is also possible to read and update the Lock bits and read the Fuse bits of the device.

Protocol

The Boot Loader software presented in this application note uses AVRprog (available free from *www.atmel.com*) as the user interface. The protocol used by the Boot Loader program is the protocol defined for AVRprog, although the Boot Loader software does not support the complete command set. A list of supported commands is shown in Table 2. All commands start with a single letter. The programmer returns 13d (carriage return) or the requested data after the command is finished. Unknown commands are replied with a "?".

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"		dd (dd)	
Read Data Memory	"d"		dd	
Write Data Memory	"D"	dd		13d
Chip Erase	"e"			13d
Write Lock Bits	"l"	dd		13d
Write Fuse Bits	"f"	dd		13d
Read Fuse Bits	"F"		dd	
Read High Fuse Bits	"N"		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 Hardware Version	"v"		dd dd	
Return Programmer Type	"p"		dd	
Set LED	"x"	dd		13d
Clear LED	"y"	dd		13d

When AVRprog.exe is executed, it searches for any supported programmers on the available COM ports. It uses 19.2 Kbps 8N1 (8 data bits, no parity bits and one stop bit) communication; The receiving UART should for this reason be configured to match this speed and mode.

Assuming communication with an ATmega161, the sequence for determining programmer type is as follows:

```
AVRprog :4 'ESC': flushing the UART buffers.
AVRprog :'S' to get software identifier
MegaAVR :'AVRB161' (boot loader). AVRprog accepts any string consisting of seven
characters starting with the three characters 'AVR'.
AVRprog :'a'  to ask for auto address incrementing
megaAVR :'y' (Yes)
AVRprog :'t' to ask for supported devices?
megaAVR :'60' for mega161 and '00' to indicate end of list
AVRprog :'T' and '60' to tell programmer that ATmega161 is selected
AVRprog :'y' +dd 'y' +dd 'y' +dd 'x' +dd to activate LEDs.
```

The sequence for programming is as follows:

```
AVRprog :3 'ESC': flushing the UART buffers.
AVRprog :'T' and '60' to tell programmer that ATmega161 is selected
AVRprog :'P' to enable programming
AVRprog :'e' to erase application area
AVRprog :'P' to enable programming
AVRprog :'A' to set address=0x0000
AVRprog :'A' to set address to start programming from
AVRprog :'c' to send low data byte
AVRprog :'C' to send high data byte
```

When the temporary buffer is full:

```
AVRprog :'A' to set address of page
AVRprog :'m' to write page
```

Then programming continues with:

```
AVRprog :'c' to send low data byte
AVRprog :'C' to send high data byte
```

When all data is transferred:

```
AVRprog :'A' to set address of last page
AVRprog :'m' to write last page
AVRprog :'L' to leave programming mode
```

Data is then verified by executing the following sequence:

```
AVRprog :'P' to enable programming
AVRprog :'A' to set address
AVRprog :'R' to read program memory
ATmega161:two bytes containing program data.
```

AVRprog will continue sending “R” until all data is read and will finish the sequence by sending “L” to leave Programming mode.

Program Description

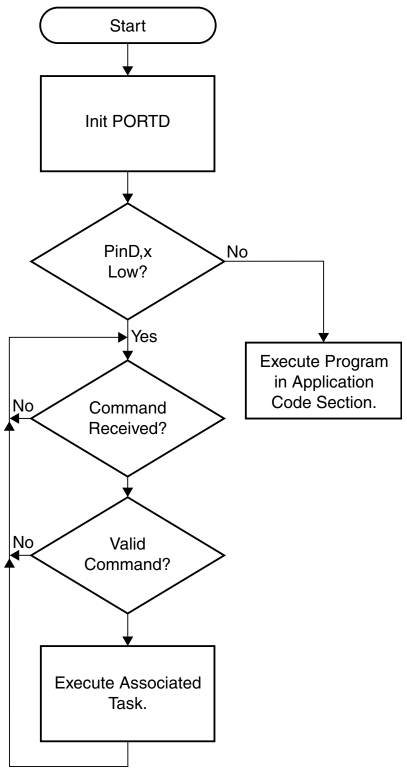
The main program starts by checking if programming is to be done, or if the program in the Application code section is to be executed. In this application, this is indicated by the value of PIND. If a user-specified pin on port D is held low during reset, the program will enter Programming mode (the pin is specified in the main.c source code). If this pin is high, program execution starts from address $0000 (as if an ordinary reset had occurred).

In Programming mode, the program receives commands from AVRprog via the UART. Each command executes an associated task. This program does not use the LED commands but they are implemented in order to prevent AVRprog from losing synchronization. Any command not recognized by the boot loader program results in a “?” being sent back to AVRprog.

Main.c

The main.c program handles communication with the host PC and executes the received commands. Figure 7 shows a flowchart illustrating the operation.

Figure 7. Main Program Execution[1]



Note: 1. As shown in Figure 7, once the Boot Loader routine is entered, the only way to exit is through a reset of the device.

Serial.c

The UART routine (serial.c) implements simple polled UART routines. As described earlier, the reason for doing this polled is that interrupts are not permitted in the Boot section for certain Boot Lock bit settings.

Assembly.s90

All routines using SPM are written in assembly. This is done in order to avoid conflicts in the code. The SPM commands require data to be placed in both the Z-register (r31:r30) and in the r1:r0 register pair. This could be done in C code, but writing in assembly simplifies the task of controlling these register pairs and reduces overhead C code.

Calling the Assembly Routines

Two of the assembly routines have dual functions, depending on the second argument sent to them:

```
void write_page (unsigned int addr, unsigned char function);
```

The first argument is the address of the page to write. The second argument indicates the function to be performed. Function = 0x05 results in writing the page to program memory. Function = 0x03 results in erasing the page.

```
unsigned int read_program_memory(unsigned int addr, unsigned char function);
```

In this routine, the first argument is the address of the page to be read. The second argument indicates the function to be performed. If function = 0x00, the routine returns the program data at the specified location. If function = 0x09 and address = 0x0000, 0x0001, or 0x0003, the routine returns the fuse, Lock Bits or the Fuse High bits, respectively. In this case, the main program ignores the 8 MSB of the returned integer.

Below is a listing of the assembly part of the program.

```
NAME      assembly(16)
RSEG      CODE(0)
RSEG      UDATA0(0)
PUBLIC    fill_temp_buffer
PUBLIC    write_page
PUBLIC    write_lock_bits
PUBLIC    read_program_memory
EXTERN    ?CL0T_1_40_L08
RSEG      CODE
#include "iom161.h"
write_page:

MOV R31,R17
MOV R30,R16                    ; move address to z pointer (R31=ZH R30=ZL)
OUT SPMCR,R20                  ; argument 2 decides function
SPM                            ; perform pagewrite
RET

fill_temp_buffer:
  MOV R31,R21
  MOV R30,R20                  ; move address to z pointer (R31=ZH R30=ZL)
  MOV R1,R17
  MOV R0,R16                   ; move data to reg 0 and 1
  LDI R18,0x01
  OUT SPMCR,R16
  SPM                           ; Store program memory
  RET

read_program_memory:
  MOV R31,R17                   ; R31=ZH R30=ZL
  MOV R30,R16                   ; move address to z pointer
  SBRC R20,0                    ; read lockbits? (second argument=0x09)
  OUT SPMEN,R20                 ; if so, place second argument in SPMEN register
  LPM                           ; read LSB
  MOV R16,R0
  INC R30
  LPM
  MOV R17,R0                    ; read MSB (ignored when reading lockbits)
  RET

write_lock_bits:
  MOV R0,R16
  LDI R17,0x09
  OUT SPMCR,R17
  SPM                           ; write lockbits
  RET
END
```

Special Considerations

1. In the ATmega161and ATmega163, the Boot Loader section stretches from $3C00-$3FFF, so the Linker File must be modified to place the program in these locations. Change the “Program address space” line to:

   ```
   // Program address space (internal Flash memory)
   -Z(CODE)INTVEC,RCODE,CDATA0,CDATA1,CCSTR,SWITCH,
   FLASH,CODE=3C00-3FFF
   ```

   This will place the code in the appropriate Boot Block. In addition, the BOOTRST Fuse must be programmed in order to move the Reset Vector to $1E00.
2. The Boot Loader program must have a way to determine whether to enter Programming mode or run the program residing in the application code section.

   This is implemented by doing a check to see if a specific I/O pin is held low during power-up. If all pins are high, the Boot Loader program executes a software jump to address $0000 and starts executing the application code. This jump can be implemented in C language by defining a function pointer pointing to address $0000,

   ```
   void (*funcptr)( void ) = 0x0000; // Set up function pointer
   ```

   and then execute the jump by calling this pointer:

   ```
   funcptr();
   ```

3. If the pin is pulled low, Programming mode is entered. It is not possible to exit Programming mode. To return to normal operation, the pin should be released and the device reset. After a reset, the port is by default configured as an input with internal pull-up disabled. The selected pin should be pulled high by an external pull-up resistor.
4. Depending on the state of the Boot Lock bits, interrupts may or may not be available when executing instructions from the Flash Boot section. For this reason, the UART routines implemented in this application note use polling instead of interrupts.
5. The SPM operations utilize the Z-register to indicate page address/temp buffer address. This register is also used as a data pointer by the IAR C compiler. This causes conflicts. All sequences dealing with SPM are therefore written in assembly, thereby achieving full control over register use.
6. It takes overhead code to gain direct control over registers writing in C. This is also a reason why all routines dealing with SPM are written in assembly.
7. The program is size optimized to 504 bytes, and will therefore fit in all parts that have a Boot Loader section of 512 bytes or more.

   In order to reduce the code size, a number of optimizations have been done:
   - *If, then, else if* statements are used instead of *case*.
   - *For(;;) {}* used instead of *while(1){}.*
   - In the CSTARTUP.S90 file, all unused references have been deleted. That includes all references to "__low_level_init", all "#if #endif" statements and the C_EXIT module.
   - All variables are implemented using the smallest data type possible.
   - Unsigned datatypes are used where this is possible.

   Consult application note "AVR035: Efficient C Coding for AVR" for more details on efficient C programming.

Atmel Headquarters

Corporate Headquarters
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 487-2600

Europe
Atmel Sarl
Route des Arsenaux 41
Case Postale 80
CH-1705 Fribourg
Switzerland
TEL (41) 26-426-5555
FAX (41) 26-426-5500

Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimhatsui
East Kowloon
Hong Kong
TEL (852) 2721-9778
FAX (852) 2722-1369

Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
TEL (81) 3-3523-3551
FAX (81) 3-3523-7581

Atmel Operations

Memory
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131
TEL 1(408) 441-0311
FAX 1(408) 436-4314

La Chantrerie
BP 70602
44306 Nantes Cedex 3, France
TEL (33) 2-40-18-18-18
FAX (33) 2-40-18-19-60

ASIC/ASSP/Smart Cards
Zone Industrielle
13106 Rousset Cedex, France
TEL (33) 4-42-53-60-00
FAX (33) 4-42-53-60-01

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Scottish Enterprise Technology Park
Maxwell Building
East Kilbride G75 0QR, Scotland
TEL (44) 1355-803-000
FAX (44) 1355-242-743

RF/Automotive
Theresienstrasse 2
Postfach 3535
74025 Heilbronn, Germany
TEL (49) 71-31-67-0
FAX (49) 71-31-67-2340

1150 East Cheyenne Mtn. Blvd.
Colorado Springs, CO 80906
TEL 1(719) 576-3300
FAX 1(719) 540-1759

Biometrics/Imaging/Hi-Rel MPU/
High Speed Converters/RF Datacom
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
TEL (33) 4-76-58-30-00
FAX (33) 4-76-58-34-80

e-mail
literature@atmel.com

Web Site
http://www.atmel.com