AVR109: Self-programming

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

SPM can be used to erase a page in program memory, to write a page in program memory that has already been erased, filling the temporary page buffer, and to set boot loader lock bits. An entire page is programmed simultaneously after first filling the temporary page buffer. The program memory must be erased one page at a time. When erasing or writing the program memory, the Z register is used as page address. When writing the program memory or the lock bits, the r1:r0 register pair contains the data. For a more in-depth description on how this is done, please refer to the Memory Programming section of the device datasheet, where a detailed description of SPM usage is presented. The SPM instruction can access the entire Flash, but can only be executed from the boot section.

Memory Organization

The memory is divided into pages of 128 or 256 bytes each. For instance, an 8 KB device with 128 bytes in each page will have 64 pages.

Figure 1. Memory Sections

AVR microcontrollers with the self-programming feature have their memory organized in two main sections: the application Flash section and the boot loader Flash section. The size of these two sections is device dependent, and
for most devices, scalable. In the latter case, the boot loader size can be changed through the BOOTSZ fuses. This feature allows for utilizing the memory where it is needed. Applications requiring a simple boot loader can allocate more Flash to the application section.

Security Issues

Writing to the Flash memory is only allowed during a limited SPM Enable Time window. To prevent accidental writing to the Flash memory, all write operations must be executed with the SPMEN I/O bit set. This bit enables the SPM instruction for the next four cycles. If set together with either BLBSET, PGWRT or PGERS, the following SPM instruction will result in setting boot lock bits, doing a Page Write or a Page Erase, respectively. If only SPMEN is set, the following SPM instruction will store the value in r1:r0 in the temporary page buffer.

The boot lock bits allow the user to:
- Protect the entire Flash from a software update by the MCU
- Protect the boot loader Flash section from a software update by the MCU
- Allow software update in the entire Flash
- Prevent code executed in the application section to read from the boot section
- Prevent code executed in the boot section to read from the application section

A more detailed description of the BLB0x and BLB1x bits can be found in the Memory Programming section of the device datasheet.

The program code within the boot loader section has the capability to read from and write to the entire Flash, including the Flash boot section. The boot loader can thus modify itself, and it can also erase itself from the code if the feature is not needed anymore. Special care must be taken if the user, by leaving boot lock bit 11 unprogrammed, allows the boot loader section to be updated. An accidental write to the boot loader itself can corrupt the entire boot loader, so further software updates might be impossible. If it is not necessary to change the boot loader software itself, it is recommended to program the boot lock bit 11 to protect the boot loader software from unintentional software changes.

The self-programming feature makes it possible to upload code to the application section, which enables unauthorized data readback from the boot section. The security issue lies in the possibility for interrupt routines to make illegitimate re-entries to the application section when supposedly protected data from the boot section is still in registers. To prevent this, no interrupts are available while executing instructions in the boot section if LPM read is disabled for this section. Thus, special care should be taken when deciding the lock bit configuration to eliminate the possibility of hostile attacks on the boot section code.

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 1. 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 1. AVRProg Commands

<table>
<thead>
<tr>
<th>Command Description</th>
<th>Host Writes</th>
<th>Host Reads</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter Programming Mode</td>
<td>ID</td>
<td>Data</td>
</tr>
<tr>
<td>Auto Increment Address</td>
<td>“a”</td>
<td>dd</td>
</tr>
<tr>
<td>Set Address</td>
<td>“A” ah al</td>
<td>13d</td>
</tr>
<tr>
<td>Write Program Memory, Low Byte</td>
<td>“c” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Write Program Memory, High Byte</td>
<td>“C” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Issue Page Write</td>
<td>“m”</td>
<td>13d</td>
</tr>
<tr>
<td>Read Lock Bits</td>
<td>“r”</td>
<td>dd</td>
</tr>
<tr>
<td>Read Program Memory</td>
<td>“R” dd</td>
<td>(dd)</td>
</tr>
<tr>
<td>Read Data Memory</td>
<td>“d” dd</td>
<td></td>
</tr>
<tr>
<td>Write Data Memory</td>
<td>“D” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Chip Erase</td>
<td>“e”</td>
<td>13d</td>
</tr>
<tr>
<td>Write Lock Bits</td>
<td>“l” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Write Fuse Bits</td>
<td>“f” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Read Fuse Bits</td>
<td>“F” dd</td>
<td></td>
</tr>
<tr>
<td>Read High Fuse Bits</td>
<td>“N” dd</td>
<td></td>
</tr>
<tr>
<td>Leave Programming Mode</td>
<td>“L”</td>
<td>13d</td>
</tr>
<tr>
<td>Select Device Type</td>
<td>“T” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Read Signature Bytes</td>
<td>“s” 3*dd</td>
<td></td>
</tr>
<tr>
<td>Return Supported Device Codes</td>
<td>“t” n*dd dd</td>
<td>00d</td>
</tr>
<tr>
<td>Return Software Identifier</td>
<td>“S” s[7]</td>
<td></td>
</tr>
<tr>
<td>Return Software Version</td>
<td>“V” dd dd</td>
<td></td>
</tr>
<tr>
<td>Return Hardware Version</td>
<td>“v” dd dd</td>
<td></td>
</tr>
<tr>
<td>Return Programmer Type</td>
<td>“p” dd</td>
<td></td>
</tr>
<tr>
<td>Set LED</td>
<td>“x” dd</td>
<td>13d</td>
</tr>
<tr>
<td>Clear LED</td>
<td>“y” dd</td>
<td>13d</td>
</tr>
</tbody>
</table>
When AVRprog.exe is executed, it searches for any supported programmers on the available COM ports. It uses 19.2 Kbps 8N1 (8 databits, no parity bits and one stopbit) 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 :'AVRBI61' (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 2 shows a flowchart illustrating the operation.

Note: As shown in Figure 2, once the boot loader routine is entered, the only way to exit is through a reset of the device.

Figure 2. Main Program Execution
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 com-
mands 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 con-
trolling these register pairs and reduces overhead C code.

Calling the Assembly Routines

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

```c
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.

```asm
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.

```asm
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,R20 ;move address to z pointer (R31=ZH
R30=ZL)
OUT SPMCR,R20 ;argument 2 decides function

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 ATmega161 and 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,

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

and then execute the jump by calling this pointer:

```c
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.
<table>
<thead>
<tr>
<th>Atmel Headquarters</th>
<th>Atmel Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Corporate Headquarters</strong></td>
<td></td>
</tr>
<tr>
<td>2325 Orchard Parkway</td>
<td></td>
</tr>
<tr>
<td>San Jose, CA 95131</td>
<td></td>
</tr>
<tr>
<td>TEL (408) 441-0311</td>
<td></td>
</tr>
<tr>
<td>FAX (408) 487-2600</td>
<td></td>
</tr>
<tr>
<td><strong>Europe</strong></td>
<td></td>
</tr>
<tr>
<td>Atmel SarL</td>
<td></td>
</tr>
<tr>
<td>Route des Arsenaux 41</td>
<td></td>
</tr>
<tr>
<td>Casa Postale 80</td>
<td></td>
</tr>
<tr>
<td>CH-1705 Fribourg</td>
<td></td>
</tr>
<tr>
<td>Switzerland</td>
<td></td>
</tr>
<tr>
<td>TEL (41) 26-426-5555</td>
<td></td>
</tr>
<tr>
<td>FAX (41) 26-426-5500</td>
<td></td>
</tr>
<tr>
<td><strong>Asia</strong></td>
<td></td>
</tr>
<tr>
<td>Atmel Asia, Ltd.</td>
<td></td>
</tr>
<tr>
<td>Room 1219</td>
<td></td>
</tr>
<tr>
<td>Chinachem Golden Plaza</td>
<td></td>
</tr>
<tr>
<td>77 Mody Road Tsimhatsui</td>
<td></td>
</tr>
<tr>
<td>East Kowloon</td>
<td></td>
</tr>
<tr>
<td>Hong Kong</td>
<td></td>
</tr>
<tr>
<td>TEL (852) 2721-9778</td>
<td></td>
</tr>
<tr>
<td>FAX (852) 2722-1369</td>
<td></td>
</tr>
<tr>
<td><strong>Japan</strong></td>
<td></td>
</tr>
<tr>
<td>Atmel Japan K.K.</td>
<td></td>
</tr>
<tr>
<td>9F, Tonetsu Shinkawa Bldg.</td>
<td></td>
</tr>
<tr>
<td>1-24-8 Shinkawa</td>
<td></td>
</tr>
<tr>
<td>Chuo-ku, Tokyo 104-0033</td>
<td></td>
</tr>
<tr>
<td>Japan</td>
<td></td>
</tr>
<tr>
<td>TEL (81) 3-3523-3551</td>
<td></td>
</tr>
<tr>
<td>FAX (81) 3-3523-7581</td>
<td></td>
</tr>
<tr>
<td><strong>Atmel Colorado Springs</strong></td>
<td></td>
</tr>
<tr>
<td>1150 E. Cheyenne Min. Blvd.</td>
<td></td>
</tr>
<tr>
<td>Colorado Springs, CO 80906</td>
<td></td>
</tr>
<tr>
<td>TEL (719) 576-3300</td>
<td></td>
</tr>
<tr>
<td>FAX (719) 540-1759</td>
<td></td>
</tr>
<tr>
<td><strong>Atmel Rousset</strong></td>
<td></td>
</tr>
<tr>
<td>Zone Industrielle</td>
<td></td>
</tr>
<tr>
<td>13106 Rousset Cedex</td>
<td></td>
</tr>
<tr>
<td>France</td>
<td></td>
</tr>
<tr>
<td>TEL (33) 4-4253-6000</td>
<td></td>
</tr>
<tr>
<td>FAX (33) 4-4253-6001</td>
<td></td>
</tr>
<tr>
<td><strong>Atmel Smart Card ICs</strong></td>
<td></td>
</tr>
<tr>
<td>Scottish Enterprise Technology Park</td>
<td></td>
</tr>
<tr>
<td>East Kilbride, Scotland G75 0QR</td>
<td></td>
</tr>
<tr>
<td>TEL (44) 1355-357-000</td>
<td></td>
</tr>
<tr>
<td>FAX (44) 1355-242-743</td>
<td></td>
</tr>
<tr>
<td><strong>Atmel Grenoble</strong></td>
<td></td>
</tr>
<tr>
<td>Avenue de Rochepleine</td>
<td></td>
</tr>
<tr>
<td>BP 123</td>
<td></td>
</tr>
<tr>
<td>38521 Saint-Egreve Cedex</td>
<td></td>
</tr>
<tr>
<td>France</td>
<td></td>
</tr>
<tr>
<td>TEL (33) 4-7658-3000</td>
<td></td>
</tr>
<tr>
<td>FAX (33) 4-7658-3480</td>
<td></td>
</tr>
</tbody>
</table>

**Fax-on-Demand**

North America: 1-(800) 292-8635
International: 1-(408) 441-0732

**e-mail**
literature@atmel.com

**Web Site**
http://www.atmel.com

**BBS**
1-(408) 436-4309

Atmel Corporation makes no warranty for the use of its products, other than those expressly contained in the Company’s standard warranty which is detailed in Atmel’s Terms and Conditions located on the Company's web site. The Company assumes no responsibility for any errors which may appear in this document, reserves the right to change devices or specifications detailed herein at any time without notice, and does not make any commitment to update the information contained herein. No licenses to patents or other intellectual property of Atmel are granted by the Company in connection with the sale of Atmel products, expressly or by implication. Atmel’s products are not authorized for use as critical components in life support devices or systems.

AVR is a trademark of Atmel Corporation.

Terms and product names in this document may be trademarks of others.