Safe and Secure Bootloader Implementation

1. Introduction

This document describes various aspects of the implementation of a safe & secure bootloader, such as the one presented in the Atmel® application note Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers, literature no. 6253. The application note discusses several design considerations when developing this kind of software and the reader is thus expected to refer to it when consulting this document.

A correct bootloader implementation poses several challenges, such as correctly remapping the chip memory with the new loaded program. Those issues will be described and solved in the following chapters, with focus on the Atmel AT91SAM ARM® Thumb® based microcontroller family specificities. An example software is also provided along with this application note.

2. Related Documents

[1] Atmel Corp., 2006, Safe and Secure Firmware Upgrade for AT91SAM Microcontrollers, literature no. 6253.

AT91 ARM Thumb Microcontrollers

Application Note

6282A–ATARM–07-Dec-06

3. Bootloader Implementation

This section details several issues one may encounter while implementing a safe & secure bootloader, along with some insight on how to approach them. Example code from a working implementation is given to illustrate the solutions.

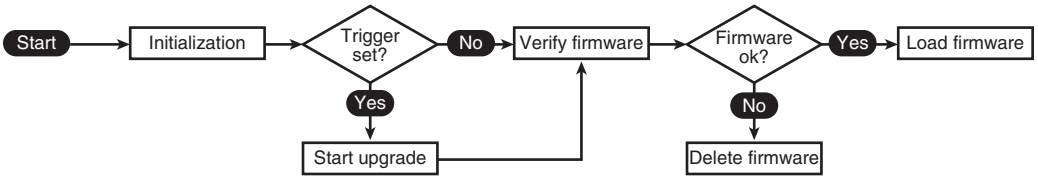
3.1 Bootloader Flow

3.1.1 Boot Sequence

The startup sequence of the bootloader is as follows:

- Initialization
- Trigger condition check
  - Firmware upgrade (if trigger condition is set)
- Firmware verification (optional)
- Firmware loading (if verification ok or disabled)

Figure 3-1. Boot Sequence Diagram



Here is the corresponding code in C:

```
// Bootloader initialization
trigger_init();
memory_init();
media_init();
communication_init();
encryption_init();

// Check trigger condition
if (trigger_poll()) {

  // Upgrade firmware
  bootloader_load(APP_START_ADDRESS);
}

// Verify firmware
if (integrity_check() != OK) {

  //
}

return APP_START_ADDRESS;
```

The next instruction uses the return value of this routine to load the firmware.

3.1.2 Upgrade Sequence

The basic upgrade flow starts with the host sending the firmware to the target, which then programs it in memory. Once the programming is done, the new application is loaded.

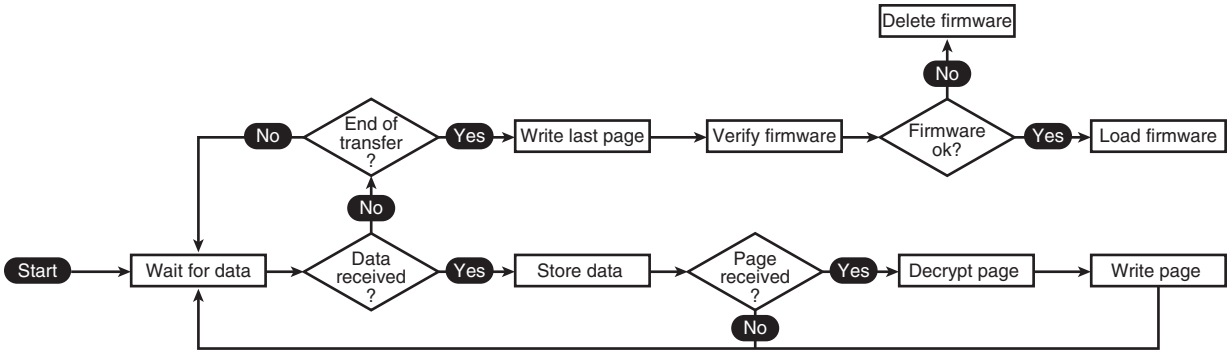
While, in theory, the “download” and “programming” steps are different, this is not the case in practice. Indeed, AT91SAM microcontrollers usually have 4x more Flash memory than RAM. Thus, they cannot store the whole firmware in RAM before writing it permanently to the Flash.

This means that the code must be written to the memory while it is received, and not after. Since a Flash write operation takes approximately 6 ms (with a page erase), a communication protocol is needed to halt the transfer when the memory is being written and resume it afterwards.

Note also that the Flash memory is split up into fixed-size blocks called pages. Depending on the quantity of Flash in the microcontroller, the page size can be between 64 and 512 bits. A memory write operation can upgrade one page (or less) at a time; thus it seems logical to send packets containing one full page.

Finally, there are several optional post-processing features to take into account. If code encryption is activated, then each page must be decrypted before being written. If a digital signature or a message authentication code is present, it must be verified once the download is complete.

Figure 3-2. Firmware Upgrading Process



Here is some sample C code implementing the upgrade flow:

```
// Receive and write firmware
pCurrent = APPLICATION_STARTING_ADDRESS;

do {
  bytes = communication_receive(page);

  // If at least one byte is received, write the page
  if (bytes > 0) {

    // Decrypt firmware
    encryption_decrypt(page, page, bytes);
```

```
    // Pad page data
    while (bytes < MEMORY_PAGE_SIZE) {
      page[bytes] = 0xFF;
      bytes++;
    }

    // Write page
    memory_write(pCurrent, page);
    pCurrent += MEMORY_PAGE_SIZE;
  }
} while (bytes > 0);
```

In this example, the *communication_receive* function waits for a whole page of data while detecting the transfer end. A returned value of 0 means that the transfer is finished.

3.1.3 Memory Partioning

Using memory partioning makes it possible to have at least one working version of the firmware in the device at any time. This is useful to avoid firmware corruption if a problem occurs during an upgrade, e.g. a power loss or a connection loss.

To do that, the Flash memory is divided into two distincts regions, A and B (excluding the bootloader region). The first one, A, is located right above the bootloader and contains the code which to be loaded. B acts as a buffer during an upgrade: the code is first downloaded to that region, verified, and finally, recopied to the first region if valid.

The boot and upgrade sequences are thus slightly modified. During the upgrade, the code is written to region B (the buffer zone). Several steps are then added to the boot sequence, regardless of whether or not a firmware upgrade has been requested. The bootloader first checks if the two codes present in regions A and B are identical. If they are, then the code in region A is loaded.

If two codes are not the same, this means that either an upgrade has just taken place, or that there has been an error during a previous upgrade. The validity of the code in region B is verified. If it is indeed valid, then it is copied over region A. If not, it is deleted.

Memory partitioning also makes it possible to use a slightly different bootloading strategy. Since there is always a working firmware embedded in the device, some functionalities of the bootloader can be moved to the user application. By doing this, they can be upgraded as well.

For example, the firmware transfer can be delegated to the user firmware. It simply downloads the new code in the buffer region and resets the chip. The bootloader then performs the remaining operations, i.e., verify region B and copy it over region A.

Figure 3-3. Boot Sequence with Memory Partitioning

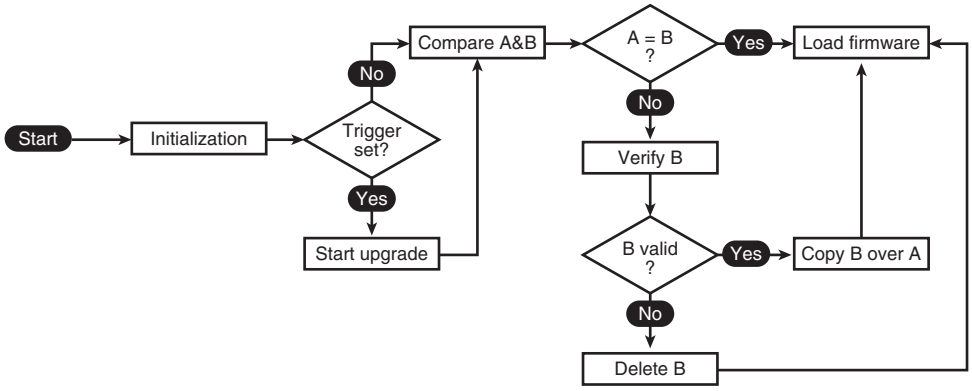
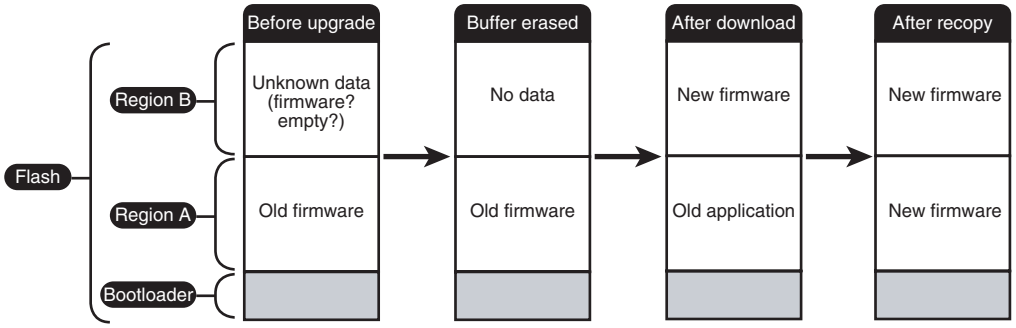


Figure 3-4. Memory Content during Upgrade Process



3.2 Bootloader Programming on AT91SAM Chips

3.2.1 Flash Programming

The Flash memory of AT91SAM microcontrollers cannot be read and written at the same time. This is because it is **single plane**. Therefore, a program executing from the Flash cannot perform a memory write. Since the bootloader is located inside the Flash, it must first be copied to the RAM prior to execution.

There are two ways to perform this step. The first one is available for certain toolchains such as IAR Embedded Workbench®. Using the *__ramfunc* attribute for a function will tell the compiler that it is supposed to run from the RAM, not the Flash. The C-initialization routine copies the function at runtime. Note that you must disable interrupts during a Flash write if this solution is used, since exception vectors will still be read from the Flash by the ARM core.

Example function performing a Flash command using the *__ramfunc* attribute:

```
__ramfunc void flash_cmd(unsigned int fcr) {

  // Start Flash operation and wait for completion
  // ITs are masked during this.
  while (!(AT91C_BASE_MC->MC_FSR & AT91C_MC_FRDY));
  mask = AT91C_BASE_AIC->AIC_IMR;
  AT91C_BASE_AIC->AIC_IDCR = 0xFFFFFFFF;

  AT91C_BASE_MC->MC_FCR = fcr | AT91C_MC_FCMD_START_PROG |
```

```
                                AT91C_MC_CORRECT_KEY;
  while (!(AT91C_BASE_MC->MC_FSR & AT91C_MC_FRDY));
}
```

The second solution is to recopy the above function into the RAM as well as exception vectors. We then use the memory remap feature of AT91SAM chips to map the RAM at virtual address 0x0, enabling proper execution of the code. This solution is slightly more complex to code compared to the first one, but has the benefit that interrupts can still work while writing the Flash.

The following code carries out the RAM copy and remap of the bootloader:

```
;-- Copy bootloader in RAM
  LDR r0, =__ramstart
  LDR r1, =__bootloaderend
  LDR r2, =0

copy:
  LDR r3, [r2]
  STR r3, [r2, r0]
  ADD r2, r2, #4
  CMP r2, r1
  BNE copy

;-- Perform a RAM remap
  LDR r0, =AT91_BASE_MC
  LDR r1, =AT91_MC_RCB
  STR r1, = [r0, #MC_RCR]
```

Do not forget to undo the remap before executing the firmware, using the same last 3 lines of code.

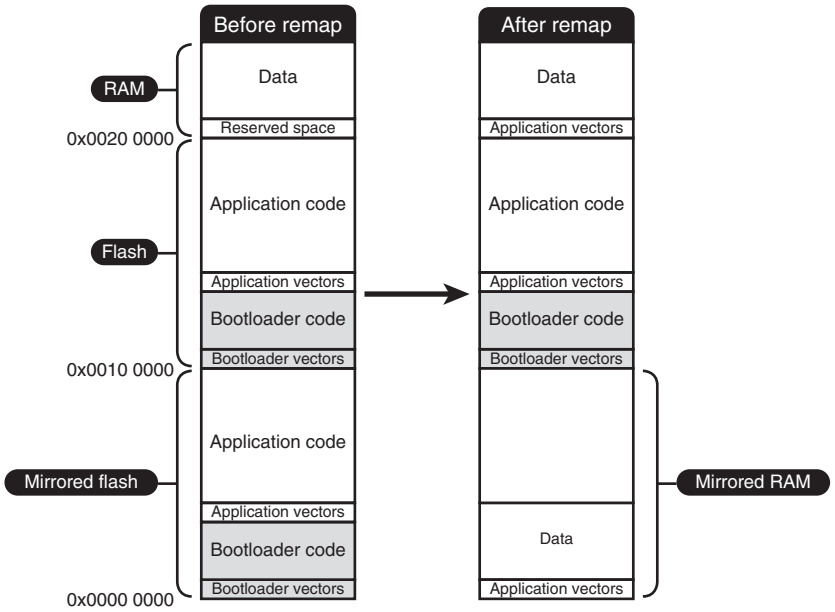
3.2.2 Exception Vectors Remapping

Atmel AT91SAM chips are based on an ARM core (either ARM7™ or ARM9™). When the core boots, it always starts to fetch code at address 0x0. Since at runtime the memory remap command is not active, the Flash is located both at adress 0x100000 and address 0x0. Therefore, the bootloader code must be located there to be executed.

In addition, the ARM core will also look for exception vectors starting at address 0x0 (the first one being the reset vector). Since the bootloader cannot be upgraded, its exception vectors will always be loaded at the beginning of the Flash. This can be a problem if the user firmware needs custom exception vectors.

The AT91SAM series offer a convenient mechanism to circumvent this problem. At all times, either the internal Flash (0x0010 0000 - 0x001F FFFF) or the internal SRAM (0x0020 0000 - 0x002F FFFF) is mirrored at address 0x0. A remap command makes it possible to toggle the memory which is currently mirrored. When the chip resets, the Flash is automatically mirrored (i.e., no remapping is done).

Figure 3-5. Memory Organization Before and After Exception Vectors Copy and Remap



Using this mechanism, the new exception vectors can be copied to the RAM. A memory remap is then performed to have the RAM available at both address 0x0020 0000 and address 0x0. This operation can be done by the firmware itself (after being loaded by the bootloader), during its initialization. Below is a code snippet to do that:

```
;-- Copy firmware exception vectors in RAM
  LDR r0, =INTERRUPT_VECTORS_START
  LDR r1, =INTERNAL_RAM_START
  LDR r2, =INTERRUPT_VECTORS_END

copy:
  LDR r3, [r0], #4
  STR r3, [r1], #4
  CMP r0, r2
  BNE copy

;-- Remap SRAM to have new exception vectors at address 0x0
  LDR r0, =AT91C_BASE_MC
  LDR r1, =AT91C_MC_RCB
  STR r1, [r0, #MC_RCR]
```

To be able to do that, some space needs to be reserved in the data segment. Otherwise, the compiler might link variables at the beginning of the RAM, which causes problems. This can be done by modifying the linker script used by the compiler. This operation is similar to the one described in the following section.

3.2.3 Application Code Linking

Ordinarily, the code segment of a user application is linked at the beginning of the Flash. This is because the ARM core of AT91SAM microcontrollers starts fetching program code at address 0x0. However, this cannot be the case when using a bootloader, since it will be itself located at address 0x0.

There are two possible solutions to this. The first one is to have the linker generate position-independent (PI) code. In a PI program, each reference to a function or variable is done relatively to the current executing instruction. This means that the code can be relocated anywhere in memory and still work perfectly.

ARM linkers can generate PI code, but only in ARM (32-bits) mode. This is because the **BX** instruction (used to switch between ARM and Thumb mode) is linked using the absolute address of the function. Therefore, PI code is only a solution if you do not use any Thumb code in your application.

The other way is to modify the address at which the application is linked, depending on the bootloader size. To do that, you have to modify the linker script of your project. Simply define a “bootloader_size” constant equals to the size of the compiled bootloader, and the first available ROM address to “flash_start+bootloader_size”. Here is an example using IAR Embedded Workbench:

```
//--
// Size of bootloader region
//--
-DBOOTSIZE=2000
...
//--
// Read-only segments
//--
-DROMSTART=(00000000+BOOTSIZE)
-DROMEND=0003FFFF
```

Code segments are then defined between ROMSTART and ROMEND.

3.2.4 Boot Region Locking

To avoid having the bootloader region accidentally erased by the user application, it is safe to lock it with the Flash controller. Any write to a locked Flash region will be aborted automatically.

A command is available in the Embedded Flash Controller (EFC) for locking a specified memory region. The problem is that a region is quite large: between 4K and 16K bytes. Since only the bootloader region must be locked, this means that the application will have to start at the beginning of the next region.

In practice, you have to set the bootloader segment size to a multiple of a region size. This effectively means that you waste the region space not used by the bootloader.

The security bit, used to protect the internal memory (Flash + SRAM) from external access, can also be set in the EFC.

4. Example Implementation

Here we describe a sample implementation of the safe & secure bootloader presented in this document. It is made up of three programs: the bootloader itself and two helper programs. One is used to transfer the firmware between a PC and the bootloader. The other one is necessary to encrypt the firmware before sending it. Those three pieces of software are detailed below.

4.1 Bootloader

4.1.1 Features

The following features have been implemented:

- Basic bootloading capabilities using a USART to transmit the firmware
- XON/XOFF protocol for flow control during download/Flash programming
- Boot region locking
- Code encryption using AES or Triple DES
- Memory partitioning

The software has been developed for IAR Embedded Workbench.

4.1.2 Configurability

The bootloader has been designed in a way which makes it easy to add new components to it, like a new media or a new communication protocol.

The */inc/config.h* file controls the configuration of both mandatory components (such as which media to use) and optional ones (security, safety). A particular component can be selected by defining the following constant:

```
#define USE_COMPONENTNAME
```

Where COMPONENTNAME is the name of the component (e.g.*USART0*). Note that only one component can be selected for each mandatory category, which are:

- Communication protocol
  - XON/XOFF (*USE_XON_XOFF*)
- Media layer
  - USART (*USE_USART0*)
- Debug
  - DBGU (*USE_DBGU*)
- Memory type
  - Flash (*USE_Flash*)
- Trigger condition
  - Dummy (*USE_DUMMY*)
  - Switches (*USE_SWITCHES*)
- Timing measurement
  - Timer0 (*USE_TIMER0*)

Both the debug and timing categories are not truly mandatory. If no debug driver is defined, for example, then the bootloader does not output debug messages. The timing module can be used to perform benchmarks on several features.

Three optional parameters are available:

- Boot region locking (*USE_BOOT_REGION_LOCKING*)
- Code encryption (*USE_ENCRYPTION*)
- Memory partitioning (*USE_MEMORY_PARTITIONING*)

4.1.3 Code Location

Main Bootloader Algorithm

The /src/main.c file contains the core bootloader algorithm. This includes the boot sequence (Section 3.1.1 on page 2), the upgrade sequence (Section 3.1.2 on page 3) as well as the modified boot & upgrade sequence for memory partitioning (Section 3.1.3 on page 4).

Memory locking of the bootloader region is also done during initialization of this bootloader.

Exception Vectors Remapping

The remapping of exception vectors is done directly during the startup of the user firmware. The code is thus located in the */resources/firmware_remap_startup_SAMxxxx.s79*.

Flash Programming

Algorithms for programming the Flash as well as locking/unlocking memory regions are located in the */src/media/flash.c* file. Note that the **__ramfunc** attribute is used for these functions to work properly. The bootloader can be modified to use RAM remapping to avoid using this compiler-specific attribute. To do that, replace the following files:

- */resources/bootloader_startup.s79*
  - */resources/bootloader_startup_remap.s79*
- */resources/bootloader_linkscript_XXX.xcl*
  - */resources/bootloader_linkscript_XXX_remap.xcl*
- */src/media/flash.c*
  - */src/media/flash_remap.c*

Linking and Startup Code

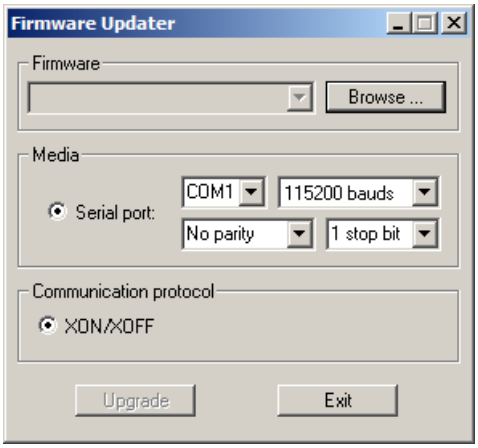
All the files used for both bootloader and user application linking and startup are located in the */resources* directory. It includes:

- Startup and linker file for the bootloader
- Startup and linker file for a firmware without remapped exception vectors
- Startup and linker file for a firmware with remapped exception vectors

4.2 Firmware Updater

This helper program is used to transmit the firmware between the host PC and the bootloader. Since only a RS232 (through a PC COM port) link is implemented at the moment, a standard terminal application (like HyperTerminal) could be used to perform the same operation. However, it will be necessary to develop a program if another media or communication protocol is implemented and not supported by standard programs.

Figure 4-1. Firmware Updater Main Window



The main window of the application enables the user to perform the following operations:

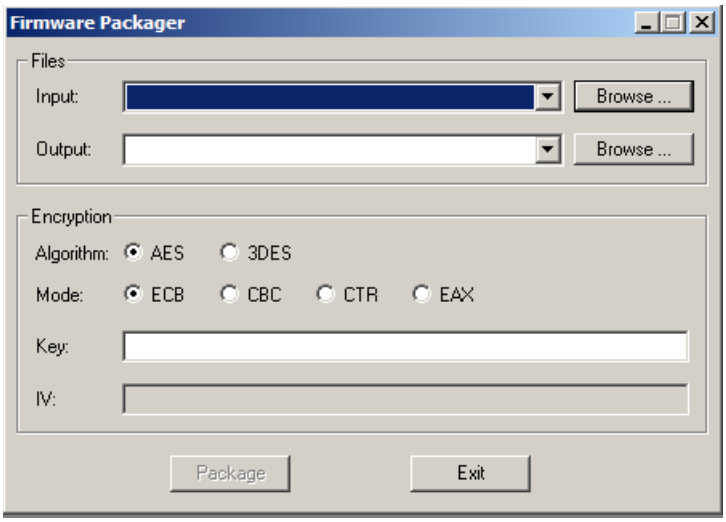
- Choose the firmware file to send to the bootloader
- Select the media and associated parameters
- Select the communication protocol and associated parameters
- Launch an upgrade

Note that you must select the same parameters in the Firmware Updater as the ones which have been selected for the bootloader. For example, if the bootloader is configured to connect using a USART configured at 115000 bps, no parity and 1 stop bit, select those parameters in the Firmware Updater.

4.3 Firmware Packager

The firmware packager is used to prepare the firmware prior to sending it to customers. This include encrypting it, generating a signature or MAC tag, etc.

Figure 4-2. Firmware Packager Main Window



Currently, the following functionalities are implemented:

- Select the firmware file to package
- Select the output file
- Select an encryption method (AES or Triple DES) with associated parameters
  - Encryption mode, key and initialization vector
- Launch the firmware packaging

Note that you must select the same parameters in the Firmware Packager as the ones selected for the bootloader. For example, if the bootloader is configured to accept a firmware encryption in AES-CBC with a particular key and IV, enter the same parameters in the Firmware packager.

5. Revision History

Table 5-1.

Document Ref.	Comments	Change Request Ref.
6282A	First issue.	

Atmel Corporation

2325 Orchard Parkway
San Jose, CA 95131, USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

Regional Headquarters

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 Tsimshatsui
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, USA
Tel: 1(408) 441-0311
Fax: 1(408) 436-4314

Microcontrollers
2325 Orchard Parkway
San Jose, CA 95131, USA
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, USA
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, USA
Tel: 1(719) 576-3300
Fax: 1(719) 540-1759

Biometrics
Avenue de Rochepleine
BP 123
38521 Saint-Egreve Cedex, France
Tel: (33) 4-76-58-47-50
Fax: (33) 4-76-58-47-60

Literature Requests
www.atmel.com/literature