Using the Memory Protection Unit (MPU) on AT91SAM7SE Microcontrollers

1. Introduction

Thanks to the **External Bus Interface (EBI)** present on the Atmel AT91SAM7SE chip, it is possible to connect external memories, such as SDRAM, NAND or NOR Flash chips. This feature can be used to store additional code which will be loaded by the main application (in internal Flash). This can have several uses:

- Increase the **code capacity** of the chip while still protecting critical parts.
- Enable the manufacturer or third parties to **add extensions** to the system without modifying the main program.

The **security bit** prevents external access to the chip's internal Flash, which guarantees that the code stored in it remains private and secure. However, loading a subprogram on an external memory poses a threat to this assumption. Indeed, the security bit does not block EBI peripherals from accessing internal memory. This means that it is easy to modify the code on the external memory to read and output the contents of the internal Flash, or use peripherals in an unwanted way.

This application note explains how to use the **Memory Protection Unit (MPU)** present on the AT91SAM7SE device in order to avoid this kind of problem and offers a safe way to execute code on external memories.

The AT91SAM7SE series is part of Atmel's family of ARM® Thumb®-based Microcontrollers.

2. References

Atmel, AT91SAM7SE Series Preliminary, lit° 6222

Atmel, Getting Started with the AT91SAM7SE Microcontroller", lit° 6295

and associated software: sam7se_getting_started_1.0.zip

ARM7TDMI® Technical Reference Manual

AT91 ARM Thumb-based Microcontrollers

Application Note

6306A–ATARM–05-Apr-07

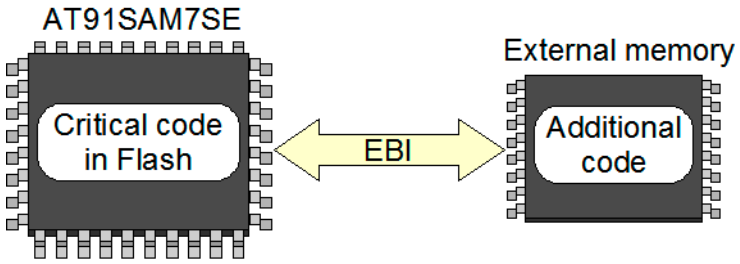
3. Case Study

3.1 Context

The AT91SAM7SE External Bus Interface (EBI) can be used as a way to increase the code capacity of a system by enabling the connection of external memories (e.g. SDRAM, NAND Flash, NOR Flash).

However, completely outsourcing the code would enable anybody to retrieve it, as external memories cannot be protected. A competitor could simply read the content of the memory chip and copy the program inside it. In order to keep a minimum level of privacy, critical sections of the code can be put in the internal Flash. This would effectively enable these critical sections to take advantage of the security bit feature of Atmel AT91SAM chips, which prevents any access from the ICE and FFPI interfaces to internal memories when set.

Figure 3-1. Using the AT91SAM7SE EBI to Extend Code Capacity



Using this architecture (depicted in the above figure) can be useful to:

- Increase the **code capacity** of the chip while still protecting critical parts.
- Enable the manufacturer or third-parties to **add extensions** to the system without modifying the main program.

In the rest of this document, the critical portion of the code located in internal Flash will be referred to as the “main application”. Code in an external memory chip will be named “external code”.

3.2 Issue

While the aforementioned strategy might look solid, there is one pitfall. The security bit does not prevent accessing internal memories through the EBI. This is convenient as it makes it easier to migrate code to external chips, as they will access peripherals and memories in the same way.

However, this poses a security problem. There is nothing to prevent an external memory from being reprogrammed, or unsoldered and replaced with another chip. It is thus easy for a potential attacker to replace the program on an external memory with another in order to, for example, dump the contents of the internal Flash, or take over critical peripherals.

3.3 Solution

The Memory Protection Unit (MPU) of the AT91SAM7SE makes it possible to monitor and regulate accesses to internal memories and peripherals. It can be used to provide a tight control over externally loaded code. The following section explains how to use the MPU.

4. Using the Memory Protection Unit

4.1 Generic Usage

The MPU can be used to protect **peripherals** and one or more **internal** (not external) **memory areas** from read/write accesses. Up to 16 user-defined regions can be specified, with size varying between 1 KB and 4 MB.

The protection offered by the MPU is configurable, and is different depending on the current mode of ARM core. If the core is in a privileged mode (any mode except USER), it will usually have greater rights. (See Section 4.2.2 "ARM® Modes of Operation") The following table shows the possible configurations:

Table 4-1. Possible MPU Configurations for Peripherals and Memory Areas

Mode #	Processor Mode	
	Privilege	User
1[1]	No access	No access
2	Read/Write	No access
3	Read/Write	Read-only
4	Read/Write	Read/Write

Note: 1. Only available for memory areas.

Whenever the MPU is enabled and an unauthorized access is made, an Abort exception is generated. This can either be a Prefetch Abort if the core tried to load code at a forbidden address, or a Data Abort if an instruction tried to read/write inside a protected region. When this occurs, two registers of the Memory Controller are updated with information about the source, type and address of abort.

4.2 Case Study Usage

4.2.1 Theory

Solving the issue presented in Section 3.2 on page 2 is done by using the fact that different rights can be given when the core runs in Privilege or User mode.

Basically, the main application (i.e. located inside the internal Flash) will run in a privileged mode. The MPU will be configured to allow read/write access to peripherals and memory areas in this case. That way, the program will be able to run normally. On the other hand, the external code (i.e. from the external memory) will run in User mode. This will make it possible to tightly control which accesses will or will not be possible.

If the external application performs an unauthorized action, the main program shall be able to continue without having the whole system crashing. This is important because otherwise, denial-of-service attacks would be possible by uploading a badly-behaved program in the memory.

4.2.2 ARM® Modes of Operation

The ARM7™ core can run in seven different modes of operation, as listed in the following table.

Table 4-2. ARM Processor Modes

Mode	Description
User	Normal program execution mode
FIQ	Used for fast interrupt handling
IRQ	Used for general-purpose interrupt handling
Supervisor	Protected mode for an Operating System
Abort	Memory protection and virtual memory management implementation
Undefined	Software emulation of hardware coprocessors
System	Runs privileged operating system tasks

All modes, except User, are referred to as **privileged** modes. When in these modes, a program has full access to all system resources without restrictions. Changing between modes can be done programmatically, but is most often the result of external interrupts or pending exceptions.

A program running in User mode cannot directly switch to a privileged mode. It must instead use the **Software Interrupt (SWI)** instruction. Whenever it is executed by the ARM core, this instruction changes the current mode to Supervisor mode and the processor jumps to the SWI vector (address 0x8). In addition, SWI takes an integer argument which can be retrieved by the exception handler routine.

4.2.3 Implementation

The solution to this case study combines the fact that the MPU can be configured with different rights in User and Privileged modes, and that a program running in User mode has to execute a SWI instruction to purposely change to a privileged mode.

Basically, the main application (in internal Flash) will define a **list of functions** for performing memory and peripheral accesses. Each of these functions will have a unique index. A program running in User mode will not call these methods directly. Instead, it will use the SWI instruction with the desired function index as a parameter. The SWI exception handler will then decode the argument, look for the corresponding function in the list, call it and finally, return to the program in User mode. This behavior is illustrated by the diagrams shown in Figure 4-1 and Figure 4-2 on page 5.

Figure 4-1. External Program Call Flow Using an SWI Function Table

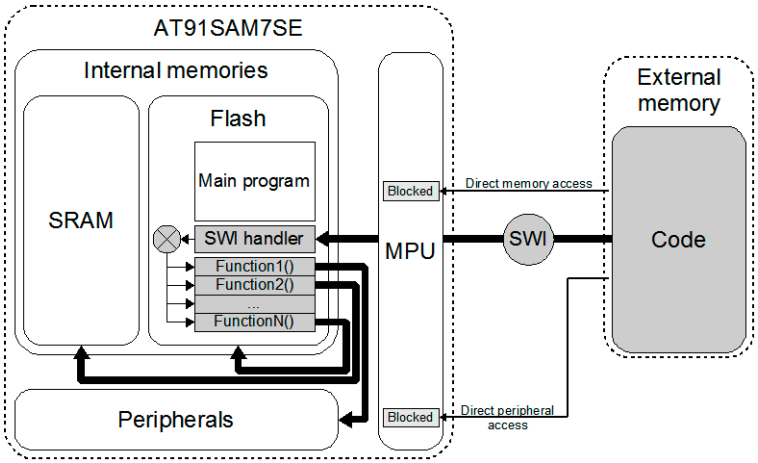
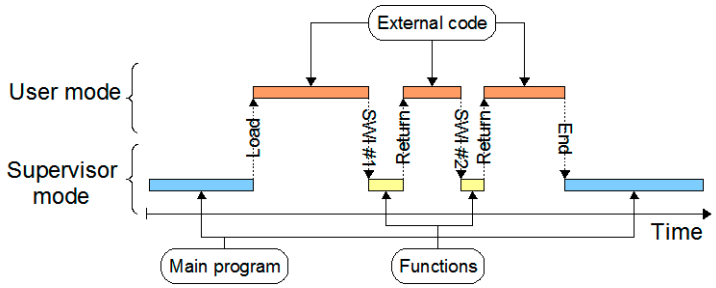


Figure 4-2. External Program Execution Timeline Using an SWI Function Table



For this to work, the MPU will have to be configured to deny any memory or peripheral access in User mode (except for a stack area). That way, an externally loaded program will not be able to go around the SWI mechanism, or it will cause an Abort.

5. Software Implementation

This section demonstrates a working implementation of an application using the MPU to protect peripherals and internal memory regions. It is based on the “Getting Started with the AT91SAM7SE Microcontroller” code example.

5.1 System

This software runs on the AT91SAM7SE-EK evaluation kit from Atmel. There are two external memory chip connected on this board: a 32-MB SDRAM and a 256-MB NAND Flash.

Since it would be too complex to add NAND Flash support to the example, the external code will run in SDRAM. It will have to be loaded using SAM-BA™ (Atmel In-Situ Programming tool). Compilation and usage of the example will be described in more detail in Section 6. ”Using the Example” on page 16.

5.2 Functionalities

This section defines the expected behavior of both the main and externally-loaded applications.

Since this example is focused on safe code loading from an external memory, the main program will not perform any particular operation (apart from loading the code on the external SDRAM). However, keep in mind that a practical implementation would include such code.

The external program will enable the user to select between four options from a DBGU-based interface:

- Try to access a memory address directly (should cause a Data Abort)
- Try to toggle a LED directly (should cause a Data Abort)
- Access a memory address through the predefined function table
- Toggle a LED through the predefined function table

This should demonstrate how the system reacts in a variety of cases.

5.3 Main Application

The main application can be divided into three sections:

- Initialization
- SWI function table
- External code loading

Implementation details for these items are given below.

5.3.1 Initialization

Apart from performing the standard chip initialization (as done in the “Getting Started” example), there are a few additional actions to perform, which are described in the following paragraphs. Note that the startup routine must eventually branch to the `main()` function in **Supervisor mode**.

5.3.1.1 MPU Initialization

The MPU shall be configured during the early steps of the chip initialization. A good place for that task is the beginning of the `main()` function. MPU configuration is a three-step operation: protected memory regions are defined first, then peripherals and finally, the MPU is enabled.

5.3.1.2 Protecting Memory Regions

The first step is to decide **which memory areas will be protected**, and the type of protection they will have. As stated in Section 4.2.3 on page 4, the MPU must be configured to deny any memory access from User mode without impeding Privilege-mode code. However, there is one exception to this statement.

All programs need access to read/write memory to store dynamic variables and set up a stack. This space is usually taken from the internal SRAM. An external piece of code has the same basic requirement, which means that it must be given access to the necessary space. This can be done in two different fashions.

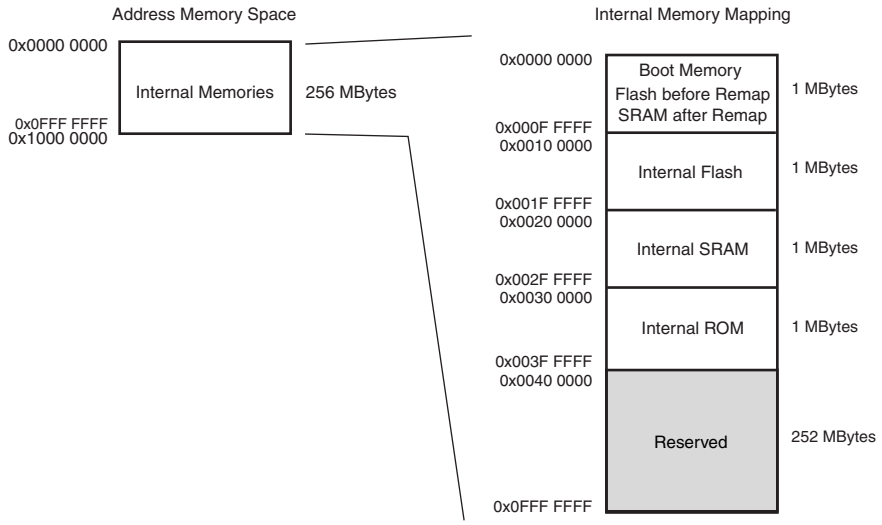
The first solution takes advantage of the fact that this example uses an SDRAM chip as its external memory. Since SDRAM supports read/write random accesses, it can be used to store variables and a stack. This is the easiest setup, as there are only four well defined regions to protect:

- Remap region (0x0000 0000 to 0x000F FFFF)
- Internal Flash (0x0010 0000 to 0x001F FFFF)

- Internal SRAM (0x0020 0000 to 0x002F FFFF)
- Internal ROM code (0x0030 0000 to 0x003F FFFF)

Note that the remap region must also be protected, as it mirrors one of the three previous areas. In addition, each region must be completely protected regardless of its real size, as it repeats itself along the allocated space. Finally, the ROM code must also be protected, as external code could try to exploit functions embedded in it otherwise.

Figure 5-1. AT91SAM7SE Internal Memory Map



For convenience, all four regions can be combined into a single 4-MB long region extending from 0x0000 0000 to 0x003F FFFF. This makes it easy to configure the MPU. Protection Unit Area 0 (MC_PUIA0) is simply set to the following values:

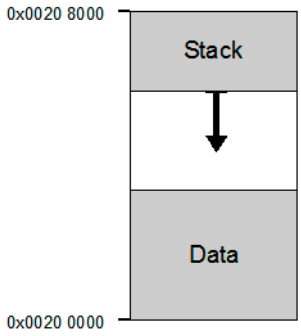
Table 5-1. MC_PUIA0 Register Values for Solution #1

Field	Value	Description
PROT	01b	Equivalent to mode #2 in Table 4-1 on page 3
SIZE	1101b	Size of protected region is 4MB
BA	0x000	Base address of area is 0x0000 0000

Note: The reset value of MC_PUIAx registers is 0, which blocks any access to the first 1 KB of memory starting at address 0. In addition, if two regions overlap, the MPU will always take the most strict settings. Therefore, **unused regions must be programmed to allow Read/Write access in both USR and SVC modes, to be 4 MB in size and to start at address 0**. Otherwise, the core will not be able to access exception vectors.

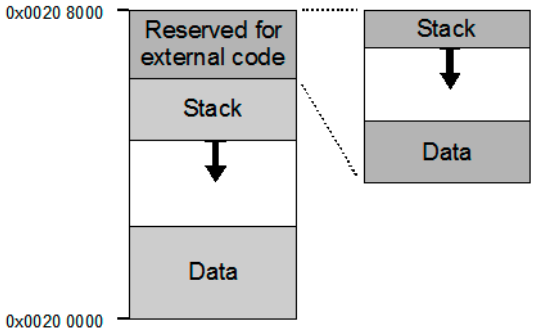
Whenever the external memory that holds the code is not random access, such as a NOR Flash chip, the memory protection strategy must be different. It is necessary to allow full access to a fixed-length SRAM area, so the external program may use it for stack and data storage. Figure 5-2 on page 8 shows a standard SRAM internal mapping:

Figure 5-2. Standard Mapping with a 32-KB SRAM



It would be unsafe to allow an external program to share data or stack space with the core application. Indeed, this could enable the former to peek on what the core is doing, or modify its data. As such, the best course of action is to create a third region dedicated to external execution. This area is most suitably placed at the top of memory, as shown below in Figure 5-3.

Figure 5-3. Reserving Stack and Data Space for an External Program with a 32-KB SRAM



This new fixed region must have an appropriate size for proper execution of external code. The size will depend on the application. Since it must be configured as unprotected in the MPU, it shall have a size between 1 KB and 4 MB, with power of two increments. Taking an 8 KB region as an example, here is the list of regions to protect:

- Remap area (0x0000 0000 - 0x000F FFFF)
- Internal Flash (0x0010 0000 - 0x001F FFFF)
- Start of the internal SRAM (0x0020 0000 - 0x0020 5FFF)
- End of the internal SRAM (0x0020 8000 - 0x002F FFFF)
- Internal ROM (0x0030 0000 - 0x003F FFFF)

Remember that the whole SRAM area (which is 1 MB long) must be considered, as a 32 KB SRAM repeats itself.

This time, the resulting configuration is more complex because of restrictions imposed by the MPU. There are only two of them:

- Region size must be a power of two between 1 KB and 4 MB
- Region base address must be a multiple of the region size

To better understand these constraints and how to produce a good configuration, the rest of this section is done step-by-step. First, consider a simple (but incorrect) two-region mapping:

Table 5-2. Simple Two-region Memory Protection Mapping for Solution #2 (**incorrect**)

#	Region	Base Address	Size
1	0x0000 0000 - 0x0020 5FFF	0x0000 0000	24KB
2	0x0020 8000 - 0x003F FFFF	0x0020 8000	2,016KB

The MPU does not support this configuration because the size of the two regions are not power of two. they are in the 1KB - 4MB range however, which is appropriate. Cutting the two areas to obtain valid sizes brings up a new (still incorrect) solution:

Table 5-3. Four-region Memory Protection Mapping for Solution #2 (**incorrect**)

#	Region	Base Address	Size
1	0x0000 0000 - 0x0020 3FFF	0x0000 0000	16KB
2	0x0020 4000 - 0x0020 5FFF	0x0020 4000	8KB
3	0x0020 8000 - 0x0030 7FFF	0x0020 8000	1MB
4	0x0030 8000 - 0x0037 7FFF	0x0030 8000	512KB
5	0x0038 8000 - 0x003C 7FFF	0x0038 8000	256KB
6	0x003C 8000 - 0x003E 7FFF	0x003C 8000	128KB
7	0x003E 8000 - 0x003F 7FFF	0x003E 8000	64KB
8	0x003F 8000 - 0x003F FFFF	0x003F 8000	32KB

While all sizes are correct this time, the second condition is not met for regions #3-8. Indeed, the base address of region #3 (0x0020 8000) is not a multiple of its size (1 MB). However, an appropriate configuration can be obtained by reversing the sizes of the last 6 regions:

Table 5-4. Correct Memory Protection Mapping for Solution #2

#	Region	Base Address	Size
1	0x0000 0000 - 0x0020 3FFF	0x0000 0000	16KB
2	0x0020 4000 - 0x0020 5FFF	0x0020 4000	8KB
3	0x0020 8000 - 0x0020 FFFF	0x0020 8000	32KB
4	0x0021 0000 - 0x0021 FFFF	0x0021 0000	64KB
5	0x0022 0000 - 0x0023 FFFF	0x0022 0000	128KB
6	0x0024 0000 - 0x0027 FFFF	0x0024 0000	256KB
7	0x0028 0000 - 0x002F FFFF	0x0028 0000	512KB
8	0x0030 0000 - 0x003F FFFF	0x0030 0000	1MB

This last try is successful, since all regions meet both the size and base address constraint. This configuration uses registers MC_PUIA0 - MC_PUIA7, which are set to the following values:

Table 5-5. MPU Register Field Values for Solution #2

#	Register	PROT	SIZE	BA
1	MC_PUIA0	01b	0100b	0x000
2	MC_PUIA1	01b	0011b	0x810
3	MC_PUIA2	01b	0101b	0x820
4	MC_PUIA3	01b	0110b	0x840
5	MC_PUIA4	01b	0111b	0x880
6	MC_PUIA5	01b	1000b	0x900
7	MC_PUIA6	01b	1001b	0xA00
8	MC_PUIA7	01b	1010b	0xC00

5.3.1.3 *Protecting Peripherals*

The MPU enables the selection between three different modes for protecting peripherals:

Table 5-6. MPU Peripheral Protection Modes

Mode #	Processor mode	
	Privilege	User
2	Read/Write	No access
3	Read/Write	Read-only
4	Read/Write	Read/Write

It is not possible to protect only a particular peripheral as they all share the same protection setting. Note that mode #1 (shown in Table 4-1 on page 3) is not available, as it would deny any peripheral access whatsoever, rendering the chip unable to interact with its external environment.

In this example, the PROT field of the Protection Unit Peripheral (MC_PUP) register should be set to 01b.

5.3.1.4 *Enabling the MPU*

Once both the memory regions and the peripherals are correctly protected, the MPU can be turned on. Setting the Protection Unit Enable Bit (PUEB) in the Protection Unit Enable Register (PUER) of the MPU enables it.

5.3.1.5 *DBGU Initialization*

The external program will use the DBGU has an user output and input medium, so it must be initialized. In addition, the main application will also output debug information through it.

Please refer to the “Getting Started with the AT91SAM7SE Microcontroller” for more information on how to setup the Debug Unit.

5.3.1.6 *PIO Controller Initialization*

As the external application shall be able to toggle LEDs, they have to be initialized first. This requires the corresponding PIO controller to be enabled and the pins associated with the LEDs to be programmed as outputs.

Please refer to the “Getting Started with the AT91SAM7SE Microcontroller” for more information about this step.

5.3.1.7 *Stack Setup*

As said before in Section 5.3.1.2 on page 6, the external program will need a data and stack area. Therefore, the stack pointer must be set correctly for User mode.

Once in User mode, it is not possible to switch back to a Privileged mode (except with the SWI instruction). This means that trying to initialize the stack in User mode would prevent the core from going back to Supervisor mode, which is necessary to execute the main application.

The workaround is to use the **System** mode instead. It shares the same registers as the User mode, but has the same rights as other Privileged modes. Once the stack is setup, the program will be able to switch back to Supervisor mode to continue its execution properly.

5.3.1.8 *Jumping to main()*

Once everything is setup appropriately, the startup routine can jump to the `main()` function. Just remember that the core must be in **Supervisor** mode at this point.

5.3.2 Setting Up the SWI Function Table

In order to allow an external program to perform certain actions in a controlled way, a mechanism based on the SWI instruction shall be implemented (see Section 4.2.3 on page 4). Three components must be setup for this:

- Definition and implementation of a set of authorized functions
- Implementation of a SWI exception handler function
- Implementation of an Abort exception handler function

5.3.2.1 *Authorized Functions*

The first step is to decide which functions will be made available to an external application. This entirely depends on the intended use of the system.

According to the description of the external program (see Section 5.2 on page 5), it should be able to do the following:

- Output text on the DBGU
- Read user input from the DBGU
- Toggle the state of a LED
- Read the content at a memory address

Initialization is taken care of during the chip startup, so there is no need to provide additional functions for doing that.

It may be desirable to keep the size of the internal Flash code minimal. Indeed, lack of space may be the exact reason why the external memory is used in the first place. Therefore, the API defined here must be the smallest possible. This can be done by providing very simple functions and having the User mode code build on top of them.

For example, it is not necessary to provide a full fledged `printf()` like function. Instead, only a simple `putc()`, which will send a single character on the DBGU, can be implemented. The external code can then use this simple method to recreate `printf()`. By using this approach, the following four functions will be provided:

- Send a single character on the DBGU
- Read a single character from the DBGU
- Toggle the state of a LED
- Read the content at a memory address

Note that the last two functions cannot be simplified much, so they will be provided as is.

Once the definitive set of exported functions has been chosen, it has to be implemented. In addition, the functions must be given a unique index (which will be used by the SWI handler). Simply numbering them from #0 to #3 is sufficient.

Finally, an array holding a pointer to each function must be declared. Note that the order of the functions must be chosen according to the previously defined indexes. The following piece of code illustrates the last two paragraphs:

```
// Function indexes
#define FUNCTION_PUTC 0
#define FUNCTION_GETC 1
#define FUNCTION_TOGGLE 2
#define FUNCTION_READ 3

// Function prototypes
extern void putc(char);
extern char getc(void);
extern void toggle();
extern unsigned int read(unsigned int);

// Function table
unsigned int pFunctions = {
  (unsigned int) putc,
  (unsigned int) getc,
  (unsigned int) toggle,
  (unsigned int) read
}
```

5.3.2.2 SWI Handler

The purpose of the SWI handler is to act as a router between the external program running in User mode and the internal Flash API. As it will be necessary to directly manipulate the ARM registers to do that, this function will be coded in assembly language. Note that registers r0-r11 cannot be modified without being saved first, as r0-r3 may hold the function parameters and r4-r11 are not to be altered according to the ARM Procedure Call Standard (APCS).

The first step is to retrieve the parameter which was specified by the external program when it executed the SWI instruction, as this should be the desired function index. This value is actually encoded in the instruction itself, in the lowest 24 bits. The only problem is located where the SWI is.

Since the link register (lr) contains the return address of the SWI call, its value can be used to retrieve the address of the SWI. Subtracting 4 (size of one ARM instruction) from lr should give the correct address, from which the SWI parameter can be extracted.

Once the function index has been retrieved, the SWI handler then checks if it is not out-of-bounds. For this, it is useful to define a constant value which holds the number of defined functions in the API.

If the index is valid, the handler can then retrieve the function address in the table and branch to it. It is necessary to save lr on the stack, as its value must be replaced with the return address of the function. Once the call returns, the last required operation is to return to the external code while switching back to User mode. This is done with the following instruction (assuming the return value is on the top of the stack):

```
ldr sp!, {pc}^
```

It modifies the program counter (pc) register to point right after the SWI call, while restoring the previous status register value (automatically saved in SPSR).

5.3.2.3 Abort Handler

If the external program performs an illegal action, the main application should be able to take back control and continue its execution normally. Whenever an unauthorized access is made, the core will enter either the Prefetch or Data Abort exception modes. The corresponding vectors will have to branch to the handler function.

The purpose of the handler is to retrieve the address right after the call to the external program and branch to it. It can be assumed that the lr register value of the Supervisor mode has been set to that address before loading the external code. In that case, the handler only has to switch back to Supervisor and branch to the address contained in lr.

Note that the handler should check whether the abort was triggered by the MPU or not, in order to treat it correctly. The Abort Status Register of the Memory Controller (MC_ASR) indicates the source of the last aborted access.

5.3.3 Loading External Code

In the protected environment that is being used in this case study, loading an external piece of code becomes more complicated than a simple branch to the correct address. The following subsections detail what actions must be performed before, after and during the actual jump to the code.

5.3.3.1 Pre-Loading

The major requirement at this step is to switch to User mode before executing unsafe external code. This cannot be done in a C function, as the language does not provide an operation for switching between processor modes. As such, the three steps of the loading process will be performed in a `load()` function written in **assembly language**.

First and foremost, it is necessary to set the value of lr to point to the post-loading stage. This is necessary because if an Abort occurs, the handler will resume the main application execution from there. The previous value should be pushed on the stack so it is not lost.

In addition, registers r0-r12 should be saved too because they are shared between the User and Supervisor mode. They should also be zero'ed to avoid leaking information about the core program execution.

Finally, it is necessary to set the value of lr in User mode. This must be done for the User program to return to the correct address. To do this, the program should change to System mode. While the latter shares the same registers as User mode, it is considered privileged and thus will not trigger a Prefetch Abort. The mode can then be changed back to Supervisor.

5.3.3.2 *Loading*

It is not possible at this point to switch in User mode and branch to the external code. Since this part of the memory is protected by the MPU, this would result in a Prefetch Abort exception. Thus, the branch and mode change operation must be done simultaneously.

It is possible to do this through the following instruction:

```
ldr sp!, {pc}^
```

This instruction pops the first word value from the stack into the program counter (pc). At the same time it restores the previous status register, which is saved in SPSR. The trick is to set SPSR to the desired value, i.e. User mode, with the MSR instruction, then execute the above line.

5.3.3.3 *Returning From User Mode*

Once in User mode, there is no way apart from the SWI instruction to purposely go back to a privileged mode. The solution used in the following paragraph does not work here because User mode does not have an SPSR register.

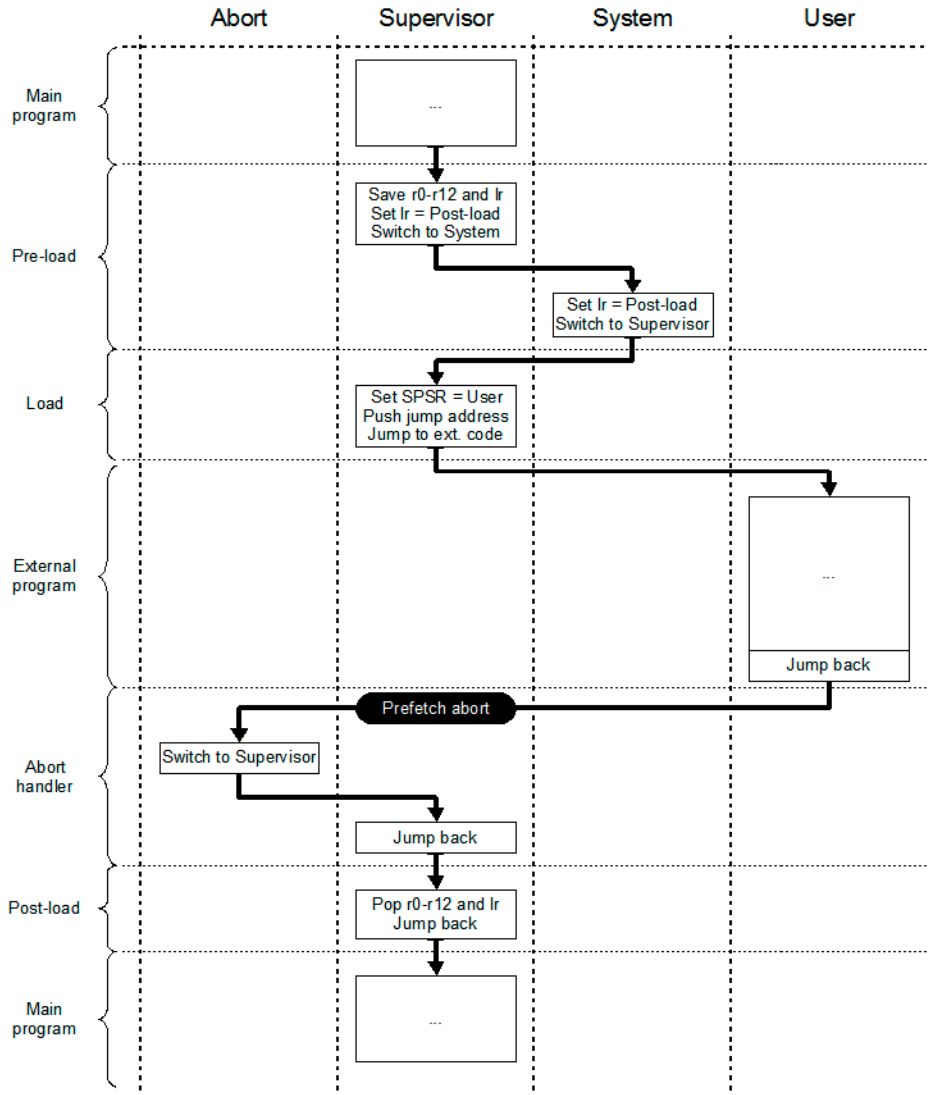
The solution is simple, as it has already been taken care of. Remember the Abort handler will allow the main application to resume its execution, by branching to the address in the link register. Whenever the external program will try to branch back to the post-loading stage, it will trigger a Prefetch Abort. The Abort handler will immediately resume execution at the same address, but in Supervisor mode. Thus, the desired result is obtained without further intervention.

5.3.3.4 *Post-Loading*

The purpose of the post-loading stage is to return the processor to the state it was in before the `load()` function was called. Basically, this means popping the r0-r12 as well as lr from the stack.

The execution of the main application is resumed by jumping to the address pointed by lr. The diagram below in Figure 5-4 sums up the whole loading process.

Figure 5-4. External Code Loading Execution Flow



5.4 External Application

Conversely to the main application, the external code requires few modifications to be functional in an MPU-based system. Basically, apart from actually coding the example, there is a single component to add, the functions defined by the main application must be re-implemented. This time however, they only have to call the SWI instruction with the correct index and return to the program after that.

Normally, a header file containing the index binding and the function prototypes have already been written (see Section 5.3.2.1 on page 11). Thus, it is only necessary to add an assembly language source file which will actually implement the functions. Below is how the `putc()` function is coded here:

```
putc:
    SWI FUNCTION_PUTC
    bx lr
```

Note that the function must probably be **exported** to allow C code to use it. This is done using a compiler-specific instruction, e.g. for GNU/GCC this is done as follows:

```
.globl putc
```

6. Using the Example

This section describes the code example provided with this application note, and explains how to use it.

6.1 Files

The root directory of the project contains two subdirectories, one for the main application and one for the external code. In addition, it also contains a script file and a launcher file; their use will be described later. A Makefile is also provided to build both the main application and the external code binaries.

6.1.1 Main Application

The main application is made up of the following files and directories:

- **bin**: directory holding executable files
- **obj**: directory holding object files
- **src**: directory holding source code files
  - board: AT91SAM7SE-EK related functions and definitions
  - chip: AT91SAM7SE512-related functions and definitions
  - dbgu: functions for using the DBGU peripheral
  - mpu: functions for using the MPU peripheral
  - mpu_appnote: implementation of this application note example software
- datavectors.lds: linker script (exception vectors are put in the .data segment)
- textvectors.lds: linker script (exception vectors are put in the .text segment)
- Makefile: makefile for building the project

6.1.2 External Code

The external code file architecture is similar to the previous one, except it only has one source code subdirectory which implements all the necessary features.

6.2 Compiling the Example

6.2.1 Requirements

In order to compile this software example, a **GNU toolchain** is necessary, as well as the **make** utility. Please refer to the "Getting Started with the AT91SAM7SE Microcontroller", application note for more information about these tools.

6.2.2 Compilation

The two code examples have their own makefiles. Thus, they can each be compiled individually by calling make while in the example root directory. This will create several binary and ELF files in the bin\ directory.

Alternatively, a top makefile is provided at the project root. It compiles both programs one after another.

6.3 Launching the Example

6.3.1 Requirements

Using the provided software example requires an AT91SAM7SE Evaluation Kit as well as Atmel AT91 In-System Programming tools available on http://www.Atmel.com.

6.3.2 Usage

As stated previously, the external code must be loaded in SDRAM with the SAM-BA tool. Since SAM-BA does not support loading and executing code in the internal SRAM for SAM7SE chips, the main application will have to be transferred to the internal Flash.

The following steps describe the loading process:

- Make sure the board is powered
- Make sure that SAM-BA Boot is running (GPNVM bit 2 cleared)
- Reset the AT91SAM7SE-EK board by connecting and disconnecting the appropriate jumper
- Plug the board to the computer using the USB cable
- Plug the DBGU serial port to one of host computer COM ports
- Launch a terminal application to view traces output by the example
  - Terminal configuration 115200 bauds, 8 bits, no parity, no flow control
- Launch SAM-BA, selecting “\usb\ARM0” as the connection and “AT91SAM7SE512-EK” for the board
- Under the flash tab:
  - In the “Send file name” text box, enter the path to the “flash_remap.bin” main application binary
  - Click on “Send file”
  - Optionally, when the transfer is finished, flash regions can be locked
  - In the “Scripts” combobox below, select the “Boot from Flash” item
  - Click on the “Execute” button
- Under the SDRAM tab:
  - In the “Scripts” combobox below, select the “Enable SDRAM” item
  - Click on the “Execute” button
  - In the “Send file name” text box, enter the path to the “sdram.bin” external code binary
  - Click on “Send file”
- In the console, enter “go 0” and press enter

A script performing most of these actions is provided in the project root directory. To use it, do the following:

- Make sure the board is powered
- Reset the AT91SAM7SE-EK board by connecting and disconnecting the appropriate jumper
- Plug the board to the computer using the USB cable
- Plug the DBGU serial port to one of host computer COM ports
- Launch a terminal application to view traces output by the example
- Execute the “Launch.bat” file located at the project root directory

Revision History

Doc. Rev	Comments	Change Request Ref.
6306A	First issue	

Headquarters

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

International

Atmel Asia
Room 1219
Chinachem Golden Plaza
77 Mody Road Tsimshatsui
East Kowloon
Hong Kong
Tel: (852) 2721-9778
Fax: (852) 2722-1369

Atmel Europe
Atmel Europe
Le Krebs
8, rue Jean-Pierre Timbaud
BP 309
78054 St Quentin-en-Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel 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

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