AVR32825: Executing code from external SDRAM

Features

- **Execute application binary from external SDRAM**
- **Place and execute single function in external SDRAM**
- **Place and execute ISR in external SDRAM**
- **Place variables in external memory**
- **Timing analysis for code execution in on-chip flash and external SDRAM**

1 Introduction

Many embedded applications incorporate external random access memory (RAM) for storing large amount of data, such as bitmap files. Similarly, many operating system-based applications need to execute the code from RAM where on-chip static random access memory (SRAM) may not be sufficient enough. Such requirements demand code execution from external RAM. To execute code from external memory, the microcontroller requires architectural support. Atmel® AVR®32 UC3A and UC3C series devices provide this feature.

This application note provides a way to execute an application binary from external synchronous dynamic random access memory (SDRAM) interfaced to UC3 devices over the external bus interface (EBI). Also, it illustrates how a single function or an interrupt service routine (ISR) can be placed and executed in external SDRAM. The document explains how to generate the binary files and copy them into external SDRAM, as well as make linker script modifications, and details the execution time comparison.

32-bit AVR® Microcontrollers

Application Note

Rev. 32160A-AVR-02/11

2 Main concepts

Executing the application binary from external SDRAM requires:

- Configuring the SDRAM controller and the target SDRAM
- Generating an executable file (.bin) using AVR32 toolchain and IAR Embedded Workbench®
    - Controlling the linker script (for the GNU linker) or linker command file (for the IAR XLINK Linker) to generate the code linked to external SDRAM addresses
- Copying the binary to SDRAM, either from an SD/MMC card or external on-board Atmel DataFlash® memory

NOTE

In the subsequent section, SDRAM would refer to external SDRAM, external to the UC3 device, unless otherwise specified.

2.1 Configuring the SDRAM controller and the target SDRAM

Before any access to the SDRAM is performed, the SDRAM controller must be configured according to the SDRAM it will be interfaced to. The SDRAM used on all Atmel evaluation kits is the Micron MT48LC16M16A2TG-7E.

Please refer to [1] for an overview of how to configure and initialize the SDRAM controller.

2.2 Generating an executable file

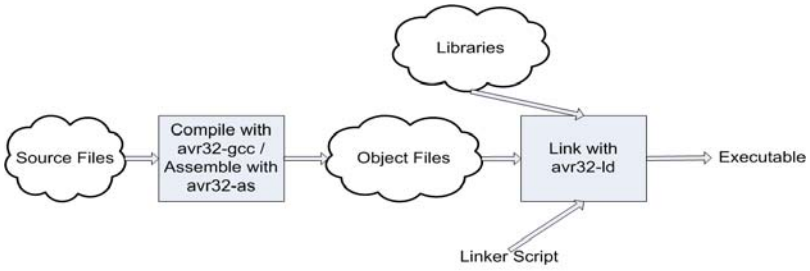
Once the SDRAM controller is configured, the SDRAM can be accessed as a normal memory-mapped device. This makes it possible for the CPU to access the instructions in SDRAM as part of the AVR32 UC3 physical memory map.

To be able to execute the code from SDRAM, the binary (executable) needs to be linked to SDRAM addresses. The linking addresses are controlled using the linker script file, in the case of the GNU linker, or linker command file, for the IAR XLINK Linker.

2.2.1 Linking process

Before explaining how the linker can be used to control the memory map of the execution binary, this section takes a brief overview of the linking process and linker file concepts. Figure 2-1 shows the basic steps of compilation.

Figure 2-1. AVR GNU toolchain build steps.



Once the compiler generates the object files, they need to be correctly linked as per the memory map of the corresponding target device. All the object files use relative addressing, and final address mapping is performed at link time. The GNU and IAR linkers make use of a linker script/command file to place different code and data sections into appropriate memory.

A linker combines input files (object file format) into a single output file (executable).

Each object file has, among other things, a list of sections. We refer to a section in an input file as an input section. Similarly, a section in the output file is an output section.

Each section in an object file has a name and a size.

Every output section has two addresses. The first is the virtual memory address (VMA). This is the address the section will have when the output file is run.

The second is the load memory address (LMA). This is the address at which the section will be loaded into memory. In most cases, the two addresses will be the same.

An example of when the LMA and VMA might be different is when a data section is loaded into read-only memory (ROM), and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM-based system). In this case, the ROM address would be the LMA, and the RAM address would be the VMA.

Some of the commonly used sections are:

.text: usually contains the program code, and is usually loaded to a nonvolatile memory, such as the internal flash.

.data: initialized data; usually contains initialized variables.

.bss: usually contains non-initialized data.

Below is an example of how to add a section that will be placed in external SDRAM.

```
.text_sdram :
{
  *(.text_sdram)
} >SDRAM :SDRAM_AT_FLASH
```

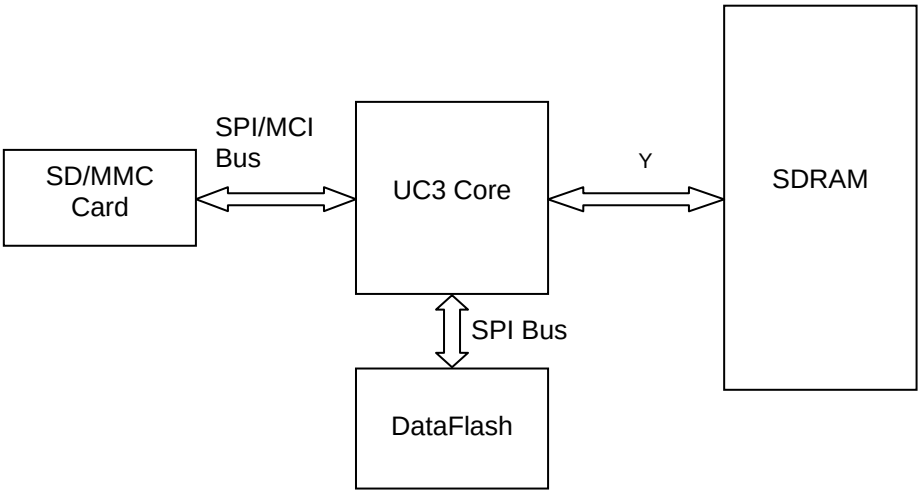
The ELF object file format uses *program headers* (PHDR), which are read by the system loader and describe how the program should be loaded into memory. After defining a section, it then has to be placed into the correct PHDR. The text_sdram section definition above, for example, would be placed in the SDRAM_AT_FLASH program header.

NOTE For detailed information about linker scripts and command files, check [2], [3] and [4].

2.3 Loading and executing a binary image

To execute code from SDRAM, the executable needs to be placed in external memory. SDRAM is a volatile memory that can retain its contents only as long as power is available. Due to the volatile nature of the SDRAM, the executable cannot be stored in SDRAM, and has to be loaded on every power-on reset. It could be loaded from either external DataFlash memory or an external SD/MMC card. Figure 2-2 shows a summary of the two different approaches.

Figure 2-2. Block diagram of the system.



2.3.1 External DataFlash

An external DataFlash memory can be interfaced to the UC3 MCU over the SPI bus. The executable has to be transferred to the onboard DataFlash memory (present on the EVK1100, for example). This can be done using an external utility. Atmel AVR32 Studio® [13] provides a software framework with built-in drivers for UC3 devices, as well as various examples. One such example is mass storage [5], which, when executed, allows the onboard DataFlash memory to be seen as one of the file system partitions on a host PC. This example can be run to copy the executable file onto this partition and eventually onto the DataFlash memory. Once this is done, the user will have to program the actual application/boot loader that will read the code from DataFlash memory and copy it into the SDRAM for execution. The overhead is in terms of a dependency on a USB mass storage device. Also, the boot loader needs to embed the FAT file system so as to read the binary from the DataFlash memory.

2.3.2 SD/MMC card

User can store the executable image on an SD/MMC card from which the boot loader will copy the executable to SDRAM using the mass storage example described in above section. The boot loader will have all the code for the MCI/SPI driver and FAT file system. The advantage is that the size of the executable can be as large as the SDRAM.

3 Execution from SDRAM

This section provides step-by-step details on how to execute a single function, an ISR, or a complete application binary from SDRAM.

3.1 AVR32 GNU toolchain

To be able to use the SDRAM in an application, it needs to be defined as a new memory segment in the linker script file with the correct address and length. For example,

```
MEMORY
{
  SDRAM (wxai): ORIGIN = 0xD0000000, LENGTH = 0x02000000
}
```

As mentioned in Section 2.2.1, a program header corresponding to new memory shall be added.

```
PHDRS
{
  SDRAM_AT_FLASH PT_LOAD;
}
```

3.1.1 Execute a function/ISR in SDRAM

Similarly, a new section should be defined in the linker script file as shown in Section 2.2.1. To place the particular function/ISR in this newly defined section, the GCC compiler defines a preprocessor directive “section,” as follows:

```
__attribute__((__section__(".text_sdram"))) void test(){…}
```

This will place the function definition test() in the .text_sdram section, which is part of SDRAM memory. It is possible to define multiple functions with this attribute so as to place them all in the SDRAM memory. When the code is compiled, the .text_sdram section will have an LMA of flash memory and VMA of SDRAM memory. The crt0.S file (startup code) will copy these functions from flash to SDRAM during system initialization so that they will be executed from SDRAM when called.

3.1.2 Execute a binary from SDRAM

By default, the CODE/TEXT, RESET, and INTERRUPT/EXCEPTION sections in the linker script file are mapped to on-chip flash memory. In order to execute the application from SDRAM, these sections have to be remapped to SDRAM addresses. As everything will be moved to SDRAM, no new section needs to be defined. However, if the user wants to retain some of the code/variables in on-chip flash, the above mentioned procedure should be followed for mapping the section to flash instead of SDRAM.

On reset, the CPU jumps to address 0x80000000 in flash, and, hence, a boot loader kind of application is required that executes in flash, copies the binary (from DataFlash/SD) into SDRAM, and transfers the execution control to the start address of SDRAM. The binary can be copied from external DataFlash memory connected over the SPI bus or SD/MMC card connected over the SPI/MCI bus.

3.2 IAR EWAVR32 environment

The linker command file has to be modified to add the SDRAM memory segment with its starting address and size.

Example

Start	Stop	Name	Type
0xD0000000	0xD2000000	SDRAM	External RAM

3.2.1 Execute a function/ISR in SDRAM

The IAR environment provides the following preprocessor directive that will place the function in flash, but then get copied to RAM during initialization and also run from RAM:

```
__ramfunc void test() {…}
```

The corresponding change in the linker command file is

```
-Z(CODE)RAMCODE32=0xD0000000-0xD1000000
```

3.2.2 Execute a binary from SDRAM

Similar to the GNU linker script file, the CODE/TEXT, RESET, and INTERRUPT/EXCEPTION sections in the linker command file are mapped to SDRAM. All the segments have to be mapped to addresses starting from 0xD0000000 instead of 0x80000000.

4 Basic application package

A standalone package with source code for a basic C application illustrating the topics addressed in this document is bundled with this application note. This section describes the contents of this package.

4.1 Requirements

The package is provided as a standalone zip file. To compile the example application it contains, the user needs to have installed at least one of the following tools:

- AVR32 Studio and the AVR32 GNU toolchain
- IAR EWAVR32

To program and execute the application on the target device, the following tools are required:

- PC with access to a serial port configured for 57600bps, 8 data bits, no parity,1 stop bit, and no flow control, or a PC with a USB port
- EVK1100, EVK1104, EVK1105, or UC3C_EK evaluation kit
- Atmel AVR JTAGICE mkII or AVR ONE!

For the UC3A0512 (EVK1100/EVK1105) device, only rev. K and higher revisions are supported.

4.2 Description of the application

The application goal is to illustrate the topics addressed in Chapter 3 of this document.

4.2.1 Setup

The project is set up so that the user can select to either execute a specific function in SDRAM or load an external binary into SDRAM based on macros.

This application note illustrates the following features:

- Interrupt execution in SDRAM
- Placing a variable in internal SRAM while the code is in SDRAM
- Placing a variable and code in SDRAM
- Calling a function in flash while the rest of the code is in SDRAM, allowing the code to jump to flash and return control back to SDRAM

There are two sets of code.

4.2.1.1 SDRAM boot loader code

This code executes in on-chip flash, and loads another binary from an external DataFlash memory or SD/MMC card into SDRAM. The project is set up to use a custom GCC linker script or a custom XLINK Linker command file.

The main file of the application is sdram_bootloader.c. The following software drivers are also used:

- INTC (to catch eventual exceptions, should an error occur)
- PM (to switch the main clock to 12MHz, and to switch the part into IDLE mode at the end of the application)

- GPIO (for pin configuration of the SPI, MCI, and external SDRAM connections)
- SDRAMC (to configure the external SDRAM before access)
- FLASHC (flash controller)
- SPI (for interfacing DataFlash memory as well as an SD/MMC card)
- MCI (for interfacing an SD/MMC card)

In addition to this file system component, AT45BDX memory, SD_MMC_MCI, and SD_MMC_SPI are also needed.

4.2.1.2 SDRAM application code

The binary file generated after compiling this code will be loaded into SDRAM either from DataFlash memory or an SD/MMC card using SDRAM boot loader code. The project is set up to use a GCC custom linker script or a custom XLINK Linker command.

The main file of the application is sdram_application.c. The following software drivers are also used:

- INTC (to catch eventual exceptions, should an error occur)
- PM (to switch the main clock to 12MHz, and to switch the part into IDLE mode at the end of the application)
- GPIO (for pin configuration of the USART)
- FLASHC (flash controller)
- USART (for interfacing DataFlash memory as well as an SD/MMC card)

4.2.2 Execution steps

4.2.2.1 SDRAM boot loader execution steps

The SDRAM boot loader code performs the following steps (only the most relevant steps for this application note are mentioned):

- C startup sequence (driven by crt0.S) for GCC or by the IAR default startup code
- Clock configuration, SDRAM initialization, SDRAM verification
- DataFlash, SD/MMC MCI, or SD/MMC SPI initialization

The following macros are defined in the conf_sdram.h file:

To execute a single function or ISR in SDRAM, use the following values:

```
#define     EXECUTE_FUNCTION_IN_SDRAM     1
#define     EXECUTE_CODE_IN_SDRAM         0
```

To load and execute the complete binary in SDRAM, use the following values:

```
#define     EXECUTE_FUNCTION_IN_SDRAM     0
#define     EXECUTE_CODE_IN_SDRAM         1
```

When binary execution from SDRAM is enabled, to load the binary from DataFlash memory or an SD/MMC card, select the appropriate BOOT_OPTION:

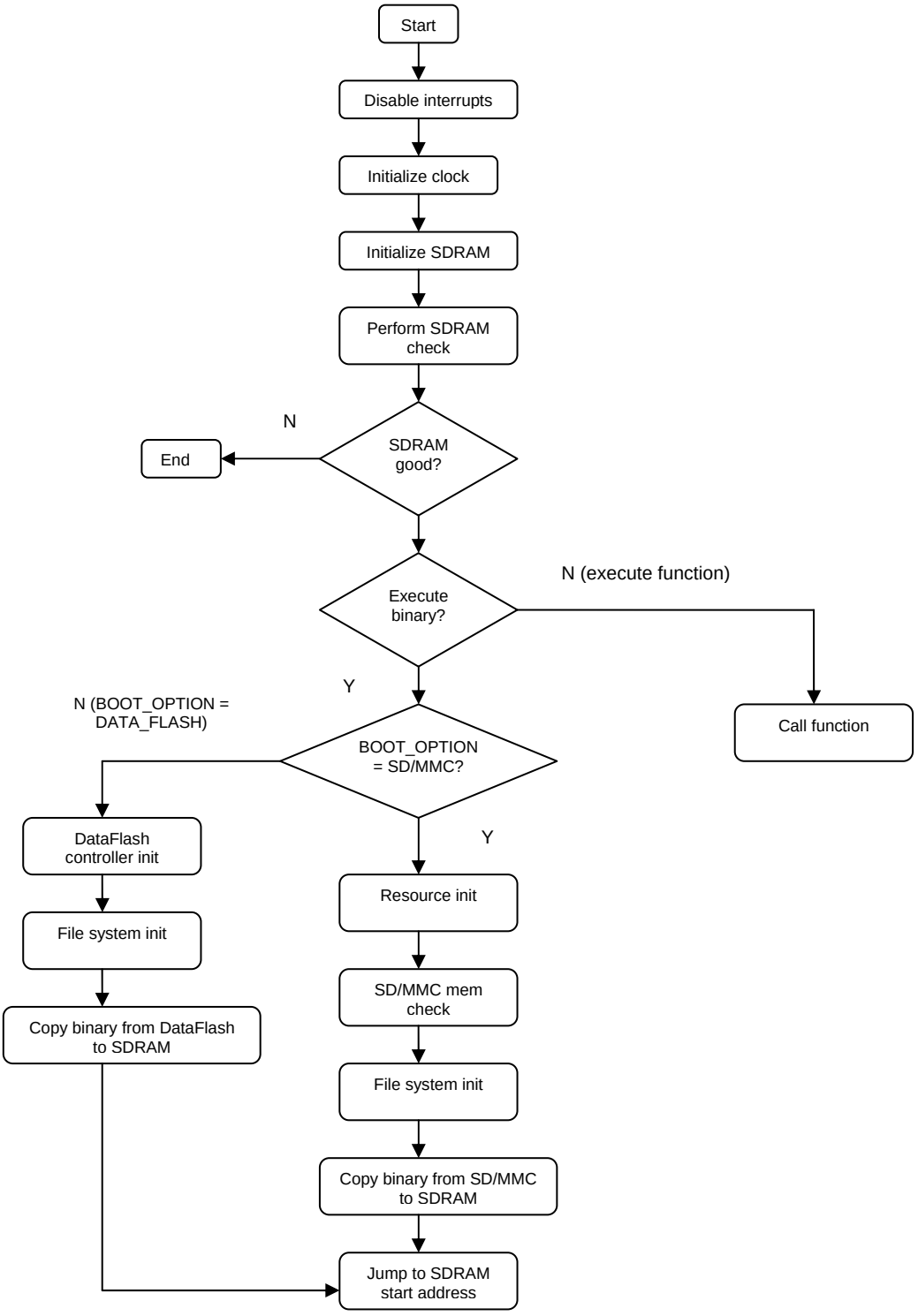
```
#define     DATA_FLASH          1
#define     SD_MMC_CARD         2

#define     BOOT_OPTION         SD_MMC_CARD
```

Depending on the BOOT_OPTION (DataFlash or SD), the binary will be read and copied to SDRAM, and control will be transferred to the start of SDRAM address, 0xD0000000.

The flowchart in Figure 4-1 shows how various conditions are handled in the SDRAM boot loader code.

Figure 4-1. SDRAM boot loader execution.



4.2.2.2 SDRAM application execution steps

The SDRAM application code performs the following steps (for this application note, only the most relevant steps are mentioned):

- Configure clock, USART, and interrupts, and then go to sleep
- If the user types any character on the serial terminal, the application will come out of sleep and run an LED chaser sequence
- Based on the macro EXT_FLASH_FUNCTION value, the function can be executed from flash or SDRAM

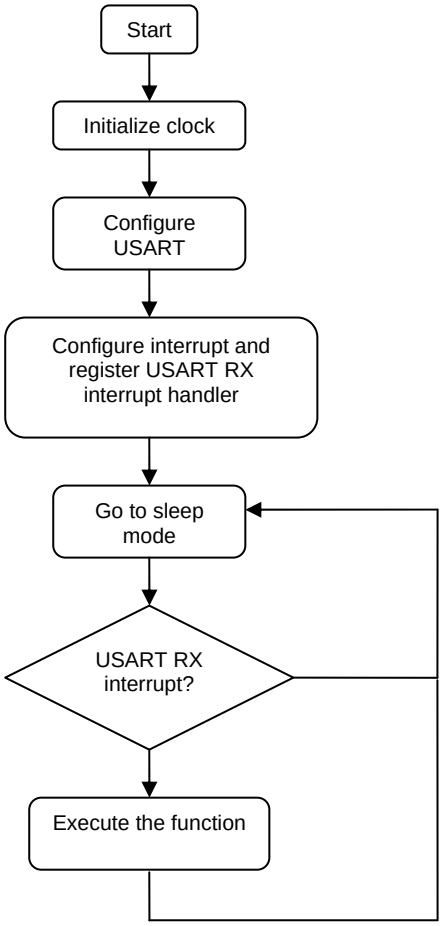
To place the LED chaser function in on-chip flash, use the following value:

```
#define       EXT_FLASH_FUNCTION          1
```

A “flag” variable is set in the USART RX interrupt handler. This variable is placed in SDRAM based on the use of the preprocessor directive “section.” If the attribute is not used, the variable is placed in the internal SRAM of the UC3 device.

The flowchart in Figure 4-2 shows how SDRAM application code is executed.

Figure 4-2. SDRAM application execution.



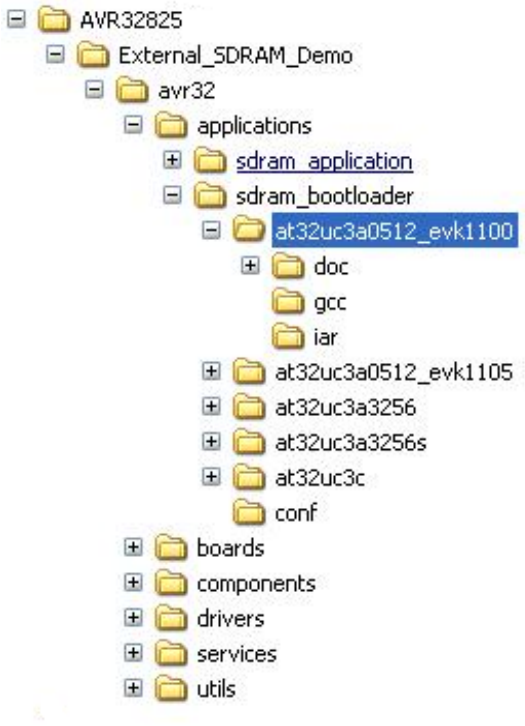
4.3 Package contents

The zip package that accompanies this application note is named AVR32825.zip. There are two folders: one for the SDRAM boot loader and another for the SDRAM application. Each folder has five subfolders (for EVK1100, EVK1104 (UC3A3256 and UC3A3256S), EVK1105, and UC3C_EK), each holding two projects:

- A makefile/config.mk project
- An IAR project

Check Section 4.4 for step-by-step instructions on how to build each project.

The makefile/config.mk projects are stored under their respective device/board category in the GCC folder, while the IAR projects are stored under their respective device/board category in the IAR folder, as shown



The main files related to this application note are:

- AVR32825/External_SDRAM_DEMO/sdram_bootloader.c:
  holds the main() function and the peripheral initialization functions for SPI, MCI, and SDRAM
- AVR32825/External_SDRAM_DEMO/startup_uc3.S:
  startup file used for system initialization and copying code to SDRAM
- AVR32825/External_SDRAM_DEMO/sdram_application.c:
  holds the main() function, USART initialization, and interrupt configuration
- linker script file (.lds) [for GCC only]:
  the GCC linker script, stored under respective device/board category in GCC folder
- linker command file(.xcl) [for IAR only]:
  the IAR XLINK Linker command file, stored under respective device/board category in IAR folder

The boards/ folder holds the abstraction layer for different boards.

The components/ folder holds the abstraction layer for the SDRAM, SD/MMC, and DataFlash.

The services/ folder holds the abstraction layer for the file system.

The drivers/ folder holds the software drivers for the EBI/SDRAMC, GPIO, INTC, PM, and USART modules.

4.4 Building and running the example on the target device

Before running the SDRAM application, connect the EVK1104 USB VPC to a PC serial port configured for 57600bps, 8 data bits, no parity, 1 stop bit and no flow control. The test results performed by the application are displayed via the USART (cf. the description of the application in Section 4.2.2). For the EVK1100, use the USART1 communication port.

In order to execute the binary from SDRAM, the binary has to be generated first. The SDRAM application needs to be built.

4.4.1 With the bare AVR32 GNU toolchain only

(The Atmel AT32UC3A3256 and EVK1104 taken as an example.)

- Connect a programmer to the EVK1104
- Turn the EVK1104 on
- For the sdram_application, open a shell, go to the AVR32825/External_SDRAM_DEMO/sdram_application/at32uc3a3256/gcc directory, and type: `make rebuild`

For GCC, the final executable is in the ELF format, which can be converted to binary using the following command:

```
avr32-objcopy –Ielf32-avr32 “elf file name” – Obinary “binary file name”
```

4.4.2 With IAR EWAVR32

(The AT32UC3A3256 and EVK1104 taken as an example.)

- Connect a programmer to the EVK1104
- Turn the EVK1104 on
- Open IAR, and load the associated IAR project
- Update the IAR header files (default location is under C:/Program Files/IAR Systems/Embedded Workbench x.x/avr32/inc/) with the content of avr32-headers.zip (located under src/SOFTWARE_FRAMEWORK/UTILS/AVR32_HEADER_FILES/). This is not required for IAR EWAVR32 version 3.31.0 and higher
- Select Project->Rebuild All

For IAR, the final executable is in hex format, which can be converted to binary using the following command:

```
avr32-objcopy –Iihex “hex file name” –Obinary “binary file name”
```

4.4.3 Copying the binary to SD/MMC or external DataFlash

The following procedure is used to copy the SDRAM application binary to an SD/MMC card or external DataFlash memory:

- Rename the binary generated above to external_sdram_binary.bin
- In AVR32 Studio, open the USB mass storage example for the corresponding device/board. (AVR32 Studio -> File -> New -> AVR Example Project)
- Compile and run the mass storage example with an SD/MMC card inserted
- In My Computer on the host PC there will be two additional partitions for the DataFlash memory and SD/MMC card
- Copy the external_sdram_binary.bin file to DataFlash or SD/MMC, whichever is required

Now the SDRAM boot loader can be executed. For sdram_bootloader, open a shell, go to the AVR32825/External_SDRAM_DEMO/sdram_bootloader/at32uc3a3256/gcc directory, and type: `make rebuild program reset run` (this command builds the application, programs it to the target, and then executes the code).

5 Timing analysis

The UC3 devices mentioned in this application note support interfacing external SDRAM over EBI. Data transfers are performed through a 16-bit data bus and an address bus of up to 24 bits, compared to the 32-bit internal data bus used to access the on-chip flash memory. Due to the reduced data bus width as well as the additional cycles required to access SDRAM, the execution time of code in SDRAM decreases by more than 50% as compared to on-chip flash.

In the case of an interrupt service routine, the ISR latency is increased as well due to the additional cycles required to access SDRAM. The actual ISR execution time would depend upon the length of the ISR.

6 References

[1] AVR32102: Using the AVR32 SDRAM controller:
http://www.atmel.com/dyn/resources/prod_documents/doc32013.pdf

[2] AVR32795: Using the GNU Linker Scripts on AVR UC3 Devices:
http://www.atmel.com/dyn/resources/prod_documents/doc32158.pdf

[3] The GNU linker ld online documentation:
http://sourceware.org/binutils/docs-2.18/ld

[4] The IAR XLINK Linker and compiler reference documentation: the EWAVR32_CompilerReference.pdf document under the IAR installation directory/IAR Systems/Embedded Workbench 5.6/avr32/doc/

[5] USB Mass Storage Class example:
http://asf.atmel.no/selector/show.php?device=uc3&store=serv

[6] The UC3A datasheet:
http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf

[7] The ATEVK1100 AVR32 UC3A0512 evaluation kit:
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4114

[8] The ATEVK1105 AVR32 UC3A0512 evaluation kit:
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4428

[9] The ATEVK1104 AVR32 UC3A3256 evaluation kit:
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4427

[10] The UC3C-EK AVR32 UC3C0512 evaluation kit:
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4822

[11] The IAR Embedded Workbench for AVR32:
http://www.iar.com/website1/1.0.1.0/124/1/index.php

[12] The AVR32 GNU toolchain:
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4118

[13] AVR32 Studio:
http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4116

7 Table of contents

Features *1*
1 Introduction *1*
2 Main concepts *2*
2.1 Configuring the SDRAM controller and the target SDRAM 2
2.2 Generating an executable file 2
2.2.1 Linking process 2
2.3 Loading and executing a binary image 3
2.3.1 External DataFlash 4
2.3.2 SD/MMC card 4
3 Execution from SDRAM *5*
3.1 AVR32 GNU toolchain 5
3.1.1 Execute a function/ISR in SDRAM 5
3.1.2 Execute a binary from SDRAM 5
3.2 IAR EWAVR32 environment 6
3.2.1 Execute a function/ISR in SDRAM 6
3.2.2 Execute a binary from SDRAM 6
4 Basic application package *7*
4.1 Requirements 7
4.2 Description of the application 7
4.2.1 Setup 7
4.2.2 Execution steps 8
4.3 Package contents 12
4.4 Building and running the example on the target device 13
4.4.1 With the bare AVR32 GNU toolchain only 13
4.4.2 With IAR EWAVR32 13
4.4.3 Copying the binary to SD/MMC or external DataFlash 14
5 Timing analysis *15*
6 References *16*
7 Table of contents *17*

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

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Milennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chou-ku, Tokyo 104-0033
JAPAN
Tel: (+81) 3523-3551
Fax: (+81) 3523-7581