AVR32733: Placing data and the heap in external SDRAM.

Features

- **Place the heap in external SDRAM**
- **Place variables in external SDRAM**
- **SDRAMC Software Framework driver usage**
- **GNU linker script overview and IAR™ linker command file overview**
- **Startup customization APIs of the AVR32 GNU Toolchain and IAR EWAVR32**

1 Introduction

This application note provides a way to place the heap and the variables of a C application in external SDRAM. It first introduces the main concepts involved before proposing the steps to follow to implement a solution. A basic C application implementing the techniques described in this document is provided as a stand-alone zip package; its content is also presented.

This document only applies to AVR32 UC3 products that have a SDRAMC module.

32-bit AVR® Microcontrollers

Application Note

Rev. 32121B-AVR32-02/10

2 Main concepts

The operation of placing the heap or variables to the external SDRAM involves:

- the SDRAM controller and the target SDRAM,
- the GCC and IAR startup customization APIs,
- the linker and especially the way it can be controlled by a linker script (for the GNU linker) or by a linker command file (for the IAR XLINK linker™),
- the heap and the dynamic memory allocation scheme.

2.1 The SDRAM Controller and the Target SDRAM

Check reference 1 together with reference 2 for a good overview of the AVR32 SDRAM Controller.

Once the SDRAM controller is correctly configured, the external SDRAM can then be accessed as a normal memory-mapped device. This feature is essential because this is what is making possible to place the heap and variables in external SDRAM and access them as part of the AVR32 UC3 physical memory map.

Refer to section 3.1.1 (for GCC) or section 3.2.1 (for IAR) for an explanation on how to initialize the SDRAM controller using the Software Framework SDRAM software driver and the GCC (or IAR) startup customization APIs.

2.2 The GCC and IAR Startup Customization API

Before the execution reaches the `main()` function, the startup code is executed. This startup code is usually part of the C Standard library (often in the object file crt0.o), is platform-dependent and is usually written in assembly code. The default crt0.o file of the newlib is linked in to the application code. CRT stands for “C Runtime”.

Note: a very basic example of a crt0.S file can be found in the AVR32 UC3 Software Framework under UTILS/STARTUP_FILES/GCC/.

The startup code is responsible for:

1. Initializing registers and modules that need it depending on the platform and environment,
2. Eventually calling the customization code `__low_level_init()` for system initialization **[IAR EWAVR32 only]**,
3. Loading initialized data having a global lifetime from the data LMA,
4. Zeroing the bss section (i.e. clearing un-initialized data having a global lifetime in the Blank Static Storage section),
5. Eventually calling the customization code `_init_startup()` for system initialization **[AVR32 GNU toolchain only]**,
6. Calling or jumping to the `main()` function.

The startup customization API `_init_startup()` (for the AVR32 GNU Toolchain) and `__low_level_init()` (for the IAR EWAVR32) are defined as weak functions: if a symbol is defined with the same name by the application, the startup code will call the customization code defined by the application.

This mechanism will be useful for initializing the SDRAMC as soon as possible (i.e. before the heap or variables placed in external SDRAM are accessed) [as performed in section 3.1.1 and in section 3.2.1].

2.2.1 The AVR32 GNU Toolchain Startup Customization API

The AVR32 GNU Toolchain startup customization function has the following prototype:

```
void _init_startup(void)
```

As mentioned in the startup code duty list earlier, this function is called after the steps 3 and 4 of the startup are performed: if variables are placed in external SDRAM, step 3 and step 4 will fail since the SDRAM controller cannot be initialized before. The solution to counter that is to write a custom startup process to:

- call the _init_startup() function before steps 3 and 4,
- initialize the sections placed in external SDRAM.

2.2.2 The IAR EWAVR32 Startup Customization API

The IAR startup customization function has the following prototype:

```
int __low_level_init(void)
```

As mentioned in the startup code duty list earlier, this function is called before the steps 3 and 4 of the startup are performed. This implies that `__low_level_init()` must not use static initialized variables, as variable initialization has not been performed at this point.

The limitation imposed by IAR is that the variables placed in external SDRAM must be declared either as **__no_init** or as **const** (see also section 3.2.4).

The value returned by `__low_level_init()` determines whether or not data segments should be initialized by the system startup code. If the function returns 0, the data segments will not be initialized.

Note 1: check reference 8 for more details on the `__low_level_init()` function.

2.3 The Linker and the Linker Command File

The linker combines a number of object files and archive files, relocates their data and resolves symbol references. Usually the last step in compiling a program is to run the linker.

The GNU linker is avr32-ld (check reference 7), the IAR linker is XLINK (check reference 8).

Both linkers accept Linker Command Language files to provide explicit and total control over the linking process. The GNU linker command file (aka linker script) is written in a superset of AT&T®'s Link Editor Command Language syntax while the IAR linker command file language is different and specific to the IAR XLINK linker.

By customizing a linker command file, it is possible to:

- define the location and size of a new memory (e.g. the external SDRAM) [this is later described in section 3.1.2 (for GCC) or section 3.2.2 (for IAR) of this document]
- specify another memory location and size for the heap (e.g. in external SDRAM) [this is described in section 3.1.3 (for GCC) or section 3.2.3 (for IAR) of this document]
- create specific sections in a given memory (e.g. create a .mydata section in the external SDRAM where global variables in that memory will be stored) [this is described in section 3.1.4 of this document (for GCC only, because it is unnecessary for IAR)].

2.4 The Heap and the Dynamic Memory Allocation Scheme

Dynamic memory allocation is the allocation of memory storage for use in a program during the runtime of that program. The functions in standard C dealing with dynamic memory allocation are: `malloc`, `free`, `calloc`, `realloc`.

These library functions allocate (or de-allocate) blocks of memory on the heap: the heap is an area of memory structured for this purpose. The location and size of this area of memory are customizable: this is later described in section 3.1.3 (for GCC) or section 3.2.3 (for IAR) of this document.

3 Placing Variables and the Heap in External SDRAM

This section provides step-by-step details on how to place variables and the heap in external SDRAM. It is based on a basic C example running on an EVK1100 which uses drivers and components source code files from the Software Framework. The basic C application stand-alone package (described in section 4 of this document) contains an example where the described techniques are applied.

Since the AVR32 GNU Toolchain and IAR EWAVR32 environments are not compatible, this chapter has been split in two sections, one for each.

3.1 Using an Environment based on the AVR32 GNU Toolchain

This section describes how to place the heap and/or variables to external SDRAM when working with the AVR32 GNU Toolchain environment.

3.1.1 Initialization of the SDRAM Controller

3.1.1.1 What: the Initialization itself

Before any access to the external SDRAM is performed, the SDRAM controller must be configured according to the external SDRAM it should drive.

The SDRAM on the EVK1100 is the Micron SDRAM MT48LC16M16A2TG-7E. The file *mt48lc16m16a2tg7e.h* is a header file describing the features (mostly the timings) of this SDRAM. It is located under *src / SOFTWARE_FRAMEWORK / COMPONENTS / MEMORY / SDRAM / MT48LC16M16A2TG7E/*.

The SDRAM controller software driver *sdramc.c/.h* (located under *src/SOFTWARE_FRAMEWORK/DRIVERS/EBI/SDRAMC/*) is a one-function interface driver:

```
void sdramc_init(unsigned long hsb_hz)
```

Since the SDRAM controller is part of the External Bus Interface (EBI) which is connected to the High Speed Bus(HSB), the `sdramc_init()` function requires the HSB frequency as an input parameter for appropriate configuration.

Example: if the application is designed to run on OSC0

```
// Switch to external oscillator 0.
pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);


// Initialize the SDRAM Controller and the external SDRAM chip.
sdramc_init(FOSC0);
// From that point on, the external SDRAM can be accessed as a
// memory being part of the AVR32 UC3 memory map.
```

3.1.1.2 How: using the Startup Customization API

This SDRAM initialization must occur before any access to the variables or the heap are done. Thus this code is placed in the startup customization function

`_init_startup()` (see section 2.2 and section 2.2.1 for more details on this feature). This function can be found in the *extsdram_example.c* main file. It is called by the startup code.

3.1.1.3 When: during the Startup Process

Since the default AVR32 GNU Toolchain startup code calls the `_init_startup()` function too late (cf. section 2.2 and section 2.2.1), an application defining variables in external SDRAM must not use the default *crt0.o* from the AVR32 GNU Toolchain and must thus use a custom *crt0.S (or crt0.x under AVR32 Studio® 2.0)* instead. This custom startup code (written in assembly) performs the basic startup steps as described in section 2.2 and most essentially calls the `_init_startup()` function before performing variables initialization in external SDRAM and in internal RAM.

Since we're using our own crt0.S file, we must tell the linker to not use its default startup files: this is done by giving avr32-gcc the option *–nostartfiles*.

Note: The SDRAM initialization code relies on the current (i.e. as set by the initialization code) HSB bus frequency. If the application later changes that frequency, the SDRAM won't operate anymore. So, care should be taken not to change the HSB frequency later in the application.

3.1.2 Adding the External SDRAM to the Linker Memory Mapping

If it has to place variables or the heap to external SDRAM, the linker must know about the external SDRAM, mostly its location and size. This is done by using the MEMORY command. The 'SDRAM' memory block is described with a start address of 0xD0000000 and a size of 32MB (this is not the maximum size supported by the UC3A, but the size of the SDRAM soldered on the EVK1100 evaluation kit).

```
MEMORY
{
  FLASH (rxai!w) : ORIGIN = 0x80000000, LENGTH = 0x00080000
  INTRAM (wxa!ri) : ORIGIN = 0x00000004, LENGTH = 0x0000FFFC
  SDRAM (wxa!ri) : ORIGIN = 0xD0000000, LENGTH = 0x02000000
  USERPAGE : ORIGIN = 0x80800000, LENGTH = 0x00000200
}
```

This command is written in the linker script *link_uc3a0512_extsdram.lds* found under *src/SOFTWARE_FRAMEWORK/APPLICATIONS/AVR32733/AT32UC3A0512_EVK1100/GCC/*.

3.1.3 Specifying the Size and Location of the Heap to External SDRAM

Again this is achieved using a linker script.

Specifying the location of the heap to SDRAM is done in the *link_uc3a0512_extsdram.lds* linker script:

```
.heap :
  {
    __heap_start__ = .;
```

```
    *(.heap)
    . = __heap_size__;
    __heap_end__ = .;
  } >SDRAM AT>SDRAM :SDRAM
```

This is assigning the .heap output section to the defined memory region SDRAM.

`__heap_size__` is a linker symbol predefined early in the linker script to:

```
__heap_size__ = DEFINED(__heap_size__) ? __heap_size__ :
LENGTH(SDRAM);
```

If the symbol `__heap_size__` is already defined, it will be used as the required heap size, else the heap size will be the total length of the SDRAM region as it was defined with the MEMORY command. This symbol can be defined when invoking the linker using the option *–Wl,--defsym,__heap_size__=value* when calling the linker through avr32-gcc (which is expressed as *–defsym __heap_size__=value* when calling avr32-ld directly).

In the stand-alone package coming with this application note, the heap size is set to 0x200000 (i.e. 2MB).

3.1.4 Placing Data in External SDRAM

The purpose of this section is to show the steps to follow to place variables in external SDRAM. This is basically done in three steps:

- defining two output sections assigned to external SDRAM in the linker script (.data_sdram (for the initialized data), .bss_sdram (for the uninitialized data)),
- adding a .data_sdram initialization step and a .bss_sdram initialization step to the startup process,
- setting, for each variable destined to be in external SDRAM, the attribute section to one of .data_sdram (if the variable has initialization data) or .bss_sdram (if the variable has no initialization value).

3.1.4.1 Defining Output Sections in External SDRAM

This is done in the linker script. Since variables can be initialized or not at declaration time (e.g. in the declaration “int i = 0;” i is an initialized variable, while in the case “int j;” j is an un-initialized variable), two sections were added: one for initialized data in external SDRAM named .data_sdram and another for un-initialized data internal SDRAM named .bss_sdram. These sections are respectively defined in the *link_uc3a0512_extsdram.lds* linker script as .data_sdram and .bss_sdram.

3.1.4.2 Adding Initialization Steps of the Sections in External SDRAM to the Startup Sequence

The .data_sdram section (holding the initialized data) must be initialized to the initialization values and the .bss_sdram section (holding the non-initialized data) must be cleared. This is done in the *crt0.S* (or *crt0.x* for AVR32 Studio) file and must be done after the SDRAMC is configured, thus after calling the `_init_startup()` function.

3.1.4.3 Assigning a Variable to External SDRAM

Last but not least, each variable that is required to be placed in external SDRAM should be declared using the GCC-specific syntax `__attribute__((__section__("section-name")))`. Each variable that has

initialization values should be placed in the *.data_sdram* section, while others should be placed in the *.bss_sdram* section.

Example (from *extsdram_example.c*):

```
// Place this variable in the .data_sdram output section.
__attribute__((__section__(".data_sdram")))
static int *au32StoreMallocPtr[EXTSDRAM_EXAMPLE_NB_MALLOC] = {(int
*)NULL,(int *)NULL,(int *)NULL,(int *)NULL};


// Place this variable in the .bss_sdram output section.
__attribute__((__section__(".bss_sdram")))
static int aHugeBuffer[EXTSDRAM_EXAMPLE_HUGEBUFF_SIZE];
```

3.2 Using the IAR EWAVR32 Environment

This section describes how to place the heap and/or data to external SDRAM when working with the IAR EWAVR32 environment.

3.2.1 Initialization of the SDRAM Controller

3.2.1.1 What: the Initialization itself

This operation is the same as for the AVR32 Gnu toolchain environment: refer to section 3.1.1.1.

3.2.1.2 How: using the Startup Customization API

This SDRAM initialization must occur before any access to the variables or the heap are done. Thus this code is placed in the startup customization function `__low_level_init()` (see section 2.2 and section 2.2.2 for more details on this feature). This function can be found in the *extsdram_example.c* main file. It is called by the startup code.

3.2.1.3 When: during the Startup Process

Since the default IAR startup code calls the `__low_level_init()` function before the variables initialization steps are performed (cf. section 2.2 and section 2.2.1), it is not necessary to provide a startup code other than the IAR's default.

Note: The SDRAM initialization code relies on the current (i.e. as set by the initialization code) HSB bus frequency. If the application later changes that frequency, the SDRAM won't operate anymore. So, care should be taken not to change the HSB frequency later in the application.

3.2.2 Adding the External SDRAM to the Linker Memory Mapping

If it has to place variables or the heap to external SDRAM, the linker must know about the external SDRAM, mostly its location and size. This is done by specifying, in the linker command file, the available address range in external SDRAM for each output section (as shown in section 3.2.3 and section 3.2.4).

The linker command file *lnkuc3a0512_extsdram.xcl* is found under *src/SOFTWARE_FRAMEWORK/APPLICATIONS/AVR32733/AT32UC3A0512_EVK1100/IAR/*.

3.2.3 Specifying the Size and Location of the Heap to External SDRAM

This is achievable using the linker command file.

Specifying the location of the heap to external SDRAM is done in the *lnkuc3a0512_extsdram.xcl* linker command file:

```
-Z(DATA)HEAP+_HEAP_SIZE=D0050000-D1FFFFFF
```

This is used to specify the range of the external SDRAM available for the heap, not the size of the heap (i.e. here the heap can be placed in the address range [0xD0050000, 0xD1FFFFFF]).

To specify the size of the heap, select **Project>Options**; in the **General Options** category, click the **Runtime** tab. Add the required heap size in the **Heap** text box. Note that this is actually setting the _HEAP_SIZE constant referred to in the linker command file.

In the stand-alone package coming with this application note, the heap size is set to 0x200000 (i.e. 2MB).

3.2.4 Placing Data in External SDRAM

The purpose of this section is to place variables in external SDRAM instead of internal RAM: using a custom modified linker command file, this will be done automatically by IAR if a variable doesn't fit in internal RAM (whether because it is too big to fit in or because other variables already use the internal RAM space) or this can be done manually using an IAR-specific syntax.

3.2.4.1 Specifying the Location and Range of the Data Segment in External SDRAM

This is achievable using the *lnkuc3a0512_extsdram.xcl* linker command file with the command:

```
-Z(DATA)DATA32_I,DATA32_Z,DATA32_N=00000004-0000FFFF,D0000000-
D004FFFF
```

This is telling IAR to place data in the memory range [0x00000004, 0x0000FFFF] (i.e. the internal RAM for a UC3 chip with 64kB of internal RAM) or in the memory range [0xD0000000, 0xD004FFFF] (i.e. part of the external SDRAM). Note that the external SDRAM range for the data segment ends at address 0xD004FFFF because the range [0xD0050000, 0xD1FFFFFF] is dedicated to the heap (cf. section 3.2.3).

Since the system and application stacks (respectively SSTACK & CSTACK) are placed in internal RAM too, the commands controlling the placement of these stacks must be defined before the command placing the DATA32 segment. This is so in the *lnkuc3a0512_extsdram.xcl* linker command file; the commands:

```
-Z(DATA)SSTACK+_SSTACK_SIZE#00000004-0000FFFF
-Z(DATA)CSTACK+_CSTACK_SIZE#00000004-0000FFFF
```

occur before the command placing the DATA32 segment.

3.2.4.2 The IAR XLINK Linker chooses the Placement of the Data

Using the linker command file with the modifications listed above, the IAR XLINK linker will choose the correct placement of variables itself:

- If a 16385*4 Bytes buffer is declared, it will automatically place it in external SDRAM (because it doesn't fit in the 64kB of internal RAM (e.g. for a UC3 part with 64kB of internal RAM)) [this is what is done with the `aHugeBuffer[]` array in the *extsdram_example.c* file of the stand-alone software package bundled with this application note],
- If the total sum of data is bigger than the internal RAM, some data will be inevitably placed in external SDRAM.

In both cases, the variables will be initialized to 0 (if they're not explicitly initialized by the application source code) or initialized to the values set by the application source code.

3.2.4.3 Controlling the Placement of Data Objects

It could sometimes be relevant to rather place a variable in internal RAM instead of external SDRAM (mostly for speed access reasons).

Absolute placement of a variable can be achieved with IAR using the @ operator or the #pragma location directive. However such variables must be declared either as **__no_init** or as **const**.

Example:

```
#pragma location = 0xD0000000
__no_init static int *au32StoreMallocPtr[4];
```

This is done in *extsdram_example.c* in the stand-alone software package bundled with this application note.

Note: The `@` operator, alternatively to the `#pragma location` directive, can be used for placing individual variables or individual functions in named segments. Refer to the IAR documentation (reference 8).

4 Basic C Application Stand-Alone Package

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

4.1 Requirements

The package is provided as a stand-alone zip file. To compile the example application, the user needs to have installed at least one of these tools:

- AVR32 Studio 2.1 and the AVR32 GNU toolchain,
- The IAR Embedded Workbench® for AVR32.

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

- a PC with access to a serial port configured as 57600bps / data bits:8 / Parity: none / Stop Bits: 1 / Flow control: none, or a PC with a USB port,
- an EVK1100 evaluation kit with an AT32UC3A0512 chip, or an EVK1104 evaluation kit with an AT32UC3A3256 chip,
- a JTAGICE mkII or an AVR ONE!.

4.2 Description of the Application

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

4.2.1 Setup

This setup is using the methods provided in section 3 of this document.

The project is setup so that the heap is placed in external SDRAM at offset 0xD0050000. The range [0xD0000000, 0xD0050000[is dedicated to variables.

The array `au32StoreMallocPtr[]` is assigned to external SDRAM and its cells are initialized to the value NULL. This array is used by the application to store the addresses of the dynamically allocated buffers.

The array `aHugeBuffer[]` is assigned to external SDRAM and it is not initialized. This array is used by the application to perform write/read checks.

The project is setup to use:

- a custom crt0.S for GCC only,
- a custom linker script (the file *link_uc3a0512_extsdram.lds*) for GCC or a custom XLINK linker command file (the file *lnkuc3a0512_extsdram.xcl*) for IAR.

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

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

- the GPIO software driver (for pins configuration of the USART and the external SDRAM connections),
- the USART software driver (to print the tests messages of the application)
- the SDRAMC software driver (to configure the external SDRAM before access).

4.2.2 Execution Steps

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

- C startup sequence (driven by *crt0.S* (or *crt0.x* for AVR32 Studio) for GCC or by the IAR default startup code, as described in section 3 of this document):
    - Call `_init_startup()` (for GCC) or `__low_level_init()` (for IAR) (these functions share the same code and are placed in the *extsdram_example.c* file):
        - Initialize the SDRAMC
    - Initialize the *.data_sdram* section (GCC only)
    - Initialize the *.bss_sdram* section (GCC only)
- Check the startup code initialization sequence (referenced by the title "*I. Startup code initialization tests.*" in *extsdram_example.c*).
- Check the dynamic allocation of memory buffer on the heap in external SDRAM (referenced by the title "*II. Dynamic allocation in external SDRAM tests.*" in *extsdram_example.c*)
- Check the statically allocated variables in external SDRAM (referenced by the title "*III. Static allocation of variables in external SDRAM tests.*" in *extsdram_example.c*)

At the end of the tests, the part is switched to the IDLE sleep mode.

4.3 Package Content

The zip package is named avr32733.zip. It contains two archive zip files (one for UC3A devices and one for UC3A3 devices), each holding three projects:

- An AVR32 Studio project,
- A makefile/config.mk project,
- An IAR project.

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

The root files *.cproject* and *.project* and the root directory *.settings* are specific to the AVR32 Studio project.

The *src* root folder is organized in subfolders and contains the source code, one makefile/config.mk project and one IAR project. The makefile/config.mk project is stored under *src/SOFTWARE_FRAMEWORK/APPLICATIONS/AVR32733/AT32UC3A0512_EVK1100/GCC/*, while the IAR project is stored under *src/SOFTWARE_FRAMEWORK/APPLICATIONS/AVR32733/AT32UC3A0512_EVK1100/IAR/*.

The main application is stored under *src/ SOFTWARE_FRAMEWORK /APPLICATIONS/ AVR32733/*. The main files related to this application note are:

- *extsdram_example.c*: holds the main() function, the `_init_startup()` and the `__low_level_init()` functions, and the declaration of the variables assigned to the external SDRAM
- *crt0.S* and *crt0.x* [for GCC only]: holds the startup sequence assembly code
- *link_uc3a0512_extsdram.lds* [for GCC only]: the GCC linker script, stored under *src/SOFTWARE_FRAMEWORK/APPLICATIONS/AVR32733/AT32UC3A0512_EVK1100/GCC/*.
- *lnkuc3a0512_extsdram.xcl* [for IAR only]: the IAR XLINK linker command file, stored under *src/SOFTWARE_FRAMEWORK/APPLICATIONS/AVR32733/AT32UC3A0512_EVK1100/IAR/*.

The *src/SOFTWARE_FRAMEWORK/BOARDS/* folder holds the EVK110x abstraction layer.

The *src/SOFTWARE_FRAMEWORK/COMPONENTS/* folder holds the abstraction layer for the SDRAM soldered on the EVK1100.

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

The src/*SOFTWARE_FRAMEWORK*/UTILS/ folder holds the basic DEBUG software module (used to print messages to UART) and various header files defining useful C macros.

4.4 Building and running the Application on Target

Before running the application, connect the EVK1100 UART_1 to a PC serial port configured as 57600bps / data bits:8 / Parity: none / Stop Bits: 1 / Flow control: none: the tests results performed by the application are displayed to UART_1 (cf. the description of the application in section 4.2.2). For the EVK1104, use the USB VPC communication port.

4.4.1 With AVR32 Studio and the AVR32 GNU Toolchain

- Launch AVR32 Studio
- In the "Project Explorer" view, right click and select the "Import" item
- Under the "General" item, select "Existing Project into Workspace" and click "Next"
- Click on ‘Select archive file’, browse to the Archive file location (select the avr32733_UC3A.zip archive or the avr32733_UC3A3.zip archive) and click "Open"
- Select the project(s) you want to import - Click "Finish" to import the selected project(s) (***Note:*** *if a project with same name already exists in your workspace, you will not be able to import it. In this case rename the project present in your workspace.)*
- Press the build button
- Load the Code: refer to the application note AVR32015: AVR32 Studio getting started

4.4.2 With the bare AVR32 GNU Toolchain only

(AT32UC3A0512 and EVK1100 taken as an example)

- Connect a programmer to the EVK1100
- Turn the EVK1100 on
- Open a shell, go to the src/SOFTWARE_FRAMEWORK/APPLICATIONS/ AVR32733/ AT32UC3A0512_EVK1100/GCC/ directory and type: *make rebuild program reset run* (this command builds the application, then programs it to the target, and then resets the target).

4.4.3 With IAR EW for AVR32

(AT32UC3A0512 and EVK1100 taken as an example)

- Connect a programmer to the EVK1100
- Turn the EVK1100 on
- Open IAR and load the associated IAR project of this application (located in the directory src/SOFTWARE_FRAMEWORK/APPLICATIONS/ AVR32733/AT32UC3A0512_EVK1100/IAR/).
- 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/).
- Press the “Debug” button at the top right of the IAR interface: the project is compiled then the generated binary file is downloaded to the microcontroller to finally switch to the debug mode.
- Click on the “Go” button in the “Debug” menu or press F5.

5 Glossary

- LMA: Load Memory Address
- SDRAM: Synchronous Dynamic RAM
- BSS: Blank Static Storage
- EBI: External Bus Interface
- HSB: High Speed Bus

6 References

1. The application note AVR32102: Using the AVR32 SDRAM controller http://www.atmel.com/dyn/resources/prod_documents/doc32013.pdf
2. The UC3A datasheet: http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf
3. The ATEVK1100 AVR32 UC3A0512 evaluation kit: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4114
4. The ATEVK1105 AVR32 UC3A0512 evaluation kit: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4428
5. The ATEVK1104 AVR32 UC3A3256 evaluation kit: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4427
6. The IAR Embedded Workbench for AVR32: http://www.iar.com/website1/1.0.1.0/124/1/index.php
7. The AVR32 GNU toolchain: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4118
8. AVR32 Studio 2.0: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4116
9. The GNU linker ld online documentation: http://sourceware.org/binutils/docs-2.18/ld
10. The IAR XLINK linker & compiler reference documentation: the EWAVR32_CompilerReference.pdf document under the IAR installation directory/IAR Systems/Embedded Workbench 4.0/avr32/doc/.
11. The AVR32 UC3 Software Framework: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4192
12. Wikipedia, The Free Encyclopedia: http://en.wikipedia.org
13. JTAGICE mkII: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3353
14. AVR ONE!: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4279

15. AVR32015: AVR32 Studio getting started:
http://www.atmel.com/dyn/resources/prod_documents/doc32086.pdf

Headquarters

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

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-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

Product Contact

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

Technical Support
avr@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. **EXCEPT AS SET FORTH IN ATMEL'S TERMS AND CONDITIONS OF SALE LOCATED ON ATMEL'S WEB SITE, ATMEL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.** Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel's products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

© 2010 Atmel Corporation. All rights reserved. Atmel®, Atmel logo and combinations thereof, AVR®, AVR® logo, AVR studio®, and others, are the registered trademarks or trademarks of Atmel Corporation or its subsidiaries. Other terms and product names may be trademarks of others.