AVR32119: Getting Started with 32-bit AVR UC3 A0/A1/A3/A4 series Flash Microcontroller

Features

- **Time Counter, Interrupt Controller and General Purpose Input/Output management on 32-bit AVR UC3 A0/A1/A3/A4 series**
- **Flash controller and Clock initialization**
- **Project Compiling and loading**
- **Associated peripherals on evaluation kits (LEDs and buttons)**

1. Introduction

This application note is aimed at helping the reader become familiar with the Atmel® 32-bit AVR® UC3 A0/A1/A3/A4 series Flash Microcontroller.

It describes in detail a simple project that uses several important features present on the UC3 A0/A1/A3/A4 series. This includes how to setup the microcontroller prior to executing the application, as well as how to add the functionalities themselves. After going through this guide, the reader should be able to successfully start a new project from scratch.

This document also explains how to setup and use a AVR32 GNU toolchain in order to compile and run a software project.

For more information about the AVR UC3 architecture, please refer to the appropriate documents available from http://www.atmel.com/.

32-bit AVR UC3 Microcontrollers

Application Note

32078C–AVR32–03/10

2. Requirements

The software provided with this application note requires several components:

- **AVR32Studio Development Tools**: AVR32 Studio is a free Integrated Development Environment (IDE) for 32-bit AVR that enables you to write, build, deploy and debug your C/C++ and assembler code. AVR32 Studio integrates with the AVR32 GNU Toolchain including GCC for building applications for 32-bit AVR. http://www.atmel.com/dyn/products/tools_card_mcu.asp?tool_id=4116
- **GNU Toolchai**n: AVR32 GNU Toolchain is a set of standalone command line programs used to create applications for 32-bit AVR microcontrollers (compiler, assembler, linker). http://www.atmel.com/dyn/products/tools_card_mcu.asp?tool_id=4118
- **EVK1100**: The EVK1100 is an evaluation kit and development system for the AT32UC3A0512 microcontroller. http://www.atmel.com/dyn/products/tools_card_mcu.asp?tool_id=4114
- **EVK1104**: The EVK1104 is an evaluation kit and development system for the AT32UC3A3256 microcontroller. http://www.atmel.com/dyn/products/tools_card_mcu.asp?tool_id=4427
- **EVK1105**: The EVK1105 is an evaluation kit and development system for the AT32UC3A0512 microcontroller. http://www.atmel.com/dyn/products/tools_card_mcu.asp?tool_id=4428
- **AT32UC3A0512 and AT32UC3A3256 Datasheet**: http://www.atmel.com/dyn/products/datasheets_mcu.asp?family_id=607
- **AVR32 UC3 Software Framework**: This framework provides software drivers, libraries and application examples to build any application for 32-bit AVR UC3 Flash Microcontroller familly. http://www.atmel.com/dyn/products/datasheets_mcu.asp?family_id=607
- **AVR UC3 Architecture Manual**: http://www.atmel.com/dyn/resources/prod_documents/doc32000.pdf
- **AVR32 Introduction to Header Files Application Note**: introduction to header files, I/O register, bit-names and module type definitions http://www.atmel.com/dyn/resources/prod_documents/doc32005.pdf

3. Getting Started with a Software Example

This section describes how to program a basic application that helps you to become familiar with UC3 A0/A1/A3/A4 microcontroller series. It is divided into two main sections: the first one covers the specification of the example (what it does, which peripherals are used); the second one details the implementation aspect.

3.1 Specification

3.1.1 Features

The demonstration program makes two LEDs on the board blink at a fixed rate. This rate is generated by using a timer for the first LED; the second one uses a Wait function based on a 1 ms tick. The blinking can be stopped using one button.

While this software may look simple, it uses several peripherals which make up the basis of an operating system. As such, it makes a good starting point for someone wanting to become familiar with the UC3 A0/A1/A3/A4 microcontroller series before looking deeper in the 32-bit AVR UC3 Software Framework.

3.1.2 Peripherals

In order to perform the operations described in the previous section, the software example uses the following set of peripherals:

- General Purpose Input/Output (GPIO) controller

- Timer Counter (TC)
- Interrupt Controller (INTC)

LEDs and buttons on the board are connected to standard input/output pins of the chip; those are managed by a GPIO controller. In addition, it is possible to have the controller generating an interrupt when the status of one of its pin changes; buttons are configured to have this behavior.

The TC is used to generate a time base, in order to obtain the LED blinking rate. It is used in interrupt mode: the TC triggers an interrupt every millisecond, incrementing a variable by one tick; the main function monitors this variable to provide an accurate delay for toggling the second LED state.

Using the INTC is required to manage interrupts. It allows the configuration of a separate vector for each source; two different functions are used to handle GPIO and TC interrupts.

3.1.3 Evaluation Kits

3.1.3.1 EVK1100

3.1.3.1.1 Booting

The AT32UC3A0512 found on EVK1100 evaluation boards features two internal memories: a 512 KB Flash and a 64KB SRAM. The Getting Started example software is to be compiled and loaded in flash.

3.1.3.1.2 Buttons

The EVK1100 evaluation kit features 3 push buttons, connected to pins PX16 (GPIO88), PX19 (GPIO85), PX22 (GPIO82). When pressed, they force a logical low level on the corresponding GPIO line.

The Getting Started example uses the push button 0 (GPIO88 - PX16).

3.1.3.1.3 LEDs

There are four general-purpose green LEDs on the EVK1100; they are wired to pins PB27, PB28, PB29 and PB30. Setting a logical low level on one of these GPIO lines turns the corresponding LED on. There are also two bi-colors LEDs, one connected on PB19 (GPIO51), PB20 (GPIO52), the other connected to PB21 (GPIO53), PB22 (GPIO54).

The example application uses the two bi-color LEDs: PB19 (GPIO51)and PB22 (GPIO54).

3.1.3.2 EVK1104

3.1.3.2.4 Booting

The AT32UC3A3256 found on EVK1104 evaluation boards features two internal memories: a 256 KB Flash and a 128KB SRAM. The Getting Started example software is to be compiled and loaded in flash.

3.1.3.2.5 Buttons

The EVK1104 Evaluation Kit features 1 push button, connected to pin PB10 (GPIO42). When pressed, they force a logical low level on the corresponding GPIO line.

The Getting Started example uses the push button SW2 (GPIO42 - PB10).

3.1.3.2.6 LEDs

There are four general-purpose green LEDs on the EVK1104; they are wired to pins PX16, PX50, PX54 and PX57. Setting a logical low level on one of these GPIO lines turns the corresponding LED on.

The example application uses LEDs: PX16 (GPIO67)and PX50 (GPIO101).

3.1.3.3 EVK1105

3.1.3.3.7 Booting

The AT32UC3A0512 found on EVK1105 evaluation boards features two internal memories: a 512 KB Flash and a 64KB SRAM. The Getting Started example software is to be compiled and loaded in flash.

3.1.3.3.8 Buttons

The EVK1105 Evaluation Kit has been developped around Qtouch features, therefore no push button has been implemented. Nevertheless, some GPIOs are free and can be manualy forced to a low level (wired to ground).

The Getting Started example uses the PA13 (GPIO13) available on the J16 free connections area.

3.1.3.3.9 LEDs

There are four general-purpose green LEDs on the EVK1105; they are wired to pins PB27, PB28, PA5 and PA6. Setting a logical low level on one of these GPIO lines turns the corresponding LED on.

The example application uses the two LEDs: PB27 (GPI59)and PB28 (GPIO60).

3.2 Implementation

As stated previously, the example defined above requires the use of several peripherals. It must also provide the necessary code for starting up the microcontroller. Both aspects are described in detail in this section, with commented source code when appropriate.

3.2.1 C-Startup

Most of the code of an embedded application is written in C. This makes the program easier to understand, more portable and modular. However, using the C language requires the initialization of several components. These initialization procedures must be performed using assembly language, and are grouped into a file referred to as **C-startup**. The C-startup code must:

- Initialize the exception vector base address and the exception vectors
- Initialize critical peripherals
- Initialize stacks
- Initialize memory segments

These steps are described in the following paragraphs.

The C-startup code will be used from the Newlib C-library. Newlib is a C standard library implementation intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses that make them easily usable on embedded products.

The Newlib is bundled with the AVR32 GNU toolchain.

3.2.1.1 *Exception*

All exceptions routines starts at the address EVBA (Exception Vector Base Address). EVBA is a register that contains a pointer to the exceptions routines table.

If the program does not need to handle an exception, then the corresponding instruction can simply be set to an infinite loop, i.e. a branch to the same address. For vectors which are to be handled, a branch instruction to a function must be provided.

In this example, the only relevant vector is the one for interrupts. It must simply branch to the interrupt handler, which is described in Section 3.2.1.2 on page 5.

Note: Refer also to the AVR UC3 Architecture Manual, section Event Processing.

3.2.1.2 *Exception: Interrupt Handler*

The main purpose of the interrupt handler is to fetch the correct jump address for the pending interrupt. This information is held in the Interrupt Vector Register (IPRn) of the INTC (see Section 3.2.5 on page 6 for more information about the INTC). Once the address is loaded, the handler just branches to it.

3.2.2 Low-Level Initialization: Flash Controller (FLASHC)

Whenever the microcontroller core runs too fast for the internal Flash, it uses one **wait state**, i.e. cycles during which it does nothing but wait for the memory. The number of wait states can be configured in the FLASHC.

After reset, the chip uses its internal slow clock (cadenced at 115 kHz), so there is no need for any wait state. However, before switching to the main oscillator or to the PLL, the correct number of wait states must be set if the frequency is above 33MHz on UC3 A0/A1/A3/A4 series. If not, the core may no longer be able to read the code from the Flash and code execution may be unpredictable.

Configuring the number of wait states is done in the Flash Control Register (FCR) of the FLASHC. For example, a 48 MHz operation requires the use of one wait state:

```
AVR32_FLASHC.fcr |= 1<<AVR32_FLASHC_FCR_FWS_OFFSET;
```

Note: Refer to the AVR UC3 Introduction to Header Files Application Note for more details about the I/O register, bit-names and module type definitions.

For more information about the required number of wait states depending on the operating frequency of a microcontroller, please refer to the AC Electrical Characteristics section of the corresponding datasheet.

In this example, the device will run at 12MHz so the flash controller does not require to use the wait state feature.

3.2.3 Low-Level Initialization: Main Oscillator

After reset, the chip runs using a slow clock (internal RC oscillator), which is cadenced at 115.2 kHz. The main oscillator must be configured in order to run at 12MHz. It can be configured in the Power Manager controller (PM).

The first step is to enable the main oscillator and wait for it to stabilize. The following piece of code performs these two operations:

- To configure the oscillator 0 in crystal mode:

```
AVR32_PM.oscctrl0=AVR32_PM_OSCCTRL0_MODE_CRYSTAL_G3<<AVR32_PM_OSCCTRL0_MODE
_OFFSET | 3<<AVR32_PM_OSCCTRL0_STARTUP_OFFSET;
```

- Then enable the oscillator 0:

```
AVR32_PM.mcctrl |= AVR32_PM_MCCTRL_OSC0EN_MASK;
```

- Wait for oscillator 0 to be ready:

```
while (!(AVR32_PM.poscsr & AVR32_PM_POSCSR_OSC0RDY_MASK));
```

- Switch main clock from internal RC oscillator to oscillator 0. On the EVK1100, EVK1104 and EVK1105, a 12MHz crystal is connected to OSC0:

```
AVR32_PM.mcctrl |=  AVR32_PM_MCCTRL_MCSEL_OSC0;
```

At this point, the chip is configured to run on the main clock with the oscillator, at the desired frequency 12MHz.

Note: For more details refer to the UC3 A0/A1 and UC3 A3/A4 series datasheet, section Power Manager.

3.2.3.1 *Low-Level Initialization: Interrupt Controller*

How to set up the INTC properly is described in Section 3.2.5 on page 6.

3.2.4 Generic Peripheral Usage

3.2.4.1 *Initialization*

Most peripherals are initialized by performing three actions

- Enabling the peripheral clock in the PM: this is already the default PM configuration.
- Enabling the control of the peripheral on GPIO pins
- Configuring the interrupt handler of the peripheral in the INTC if required
- Enabling the interrupt source at the peripheral level

Finally, if an interrupt is to be generated by the peripheral, then the source must be configured properly in the Interrupt Controller. Please refer to Section 3.2.5 on page 6 for more information.

3.2.5 Using the Interrupt Controller (INTC)

3.2.5.1 *Purpose*

The INTC manages all internal and external interrupts of the system. It enables the definition of one handler for each interrupt source, i.e., a function which is called whenever the corresponding event occurs. Interrupts can also be individually enabled or masked, and have several different priority levels.

3.2.5.2 *Initialization*

The only mandatory action to perform. This is done using the INTC library (int.c, intc.h files) with the instructions:

- Setup the interrupt vectors

```
INTC_init_interrupts();
```

- Register the interrupt handlers for TimerCounter (AVR32_TC_IRQ0 is the IRQ of the interrupt handler to register, INT0 is the interrupt priority level to assign to the group of this IRQ) and GPIO (In every port there are four interrupt lines connected to the interrupt controller. Every eight interrupts in the port are stored together to form an interrupt line. That is why we use the formula "AVR32_GPIO_IRQ_0 + (GPIO to be registered/8)".

EVK1100: AVR32_GPIO_IRQ_0 is used as the base interrupt line and we add '(88/8)' to register the corresponding interrupt line.

```
INTC_register_interrupt(&tc_irq, AVR32_TC_IRQ0, INT0);
INTC_register_interrupt(&gpio_irq, (AVR32_GPIO_IRQ_0+88/8), INT1);
```

EVK1104: AVR32_GPIO_IRQ_0 is used as the base interrupt line and we add '(42/8)' to register the corresponding interrupt line.

```
INTC_register_interrupt(&tc_irq, AVR32_TC_IRQ0, INT0);
INTC_register_interrupt(&gpio_irq, (AVR32_GPIO_IRQ_0+42/8), INT1);
```

EVK1105: AVR32_GPIO_IRQ_0 is used as the base interrupt line and we add '(13/8)' to register the corresponding interrupt line.

```
INTC_register_interrupt(&tc_irq, AVR32_TC_IRQ0, INT0);
INTC_register_interrupt(&gpio_irq, (AVR32_GPIO_IRQ_0+13/8), INT1);
```

3.2.6 Using the Timer Counter

3.2.6.1 Purpose

Timer Counters on 32-bit AVR UC3 series can perform several functions, e.g., frequency measurement, pulse generation, delay timing, Pulse Width Modulation (PWM), etc.

In this example, the primary goal of the Timer Counter (TC) is to generate periodic interrupts. This is most often used to provide the base tick of an operating system. The TC uses PBA divided by x as its input clock (x=2, 8, 32 or 128 on revision G and higher, x=4, 8, 1 6, 32 on revision E).

The getting started example uses the TC to provide a 1 ms time base. Each time the TC interrupt is triggered, a 32-bit counter is incremented.

3.2.6.2 Initialization

The first step is to configure the Channel Mode Register (CMR). TC channels can operate in two different modes. The first one, which is referred to as the Capture mode, is normally used for performing measurements on input signals. The second one, the Waveform mode, enables the generation of pulses. In the example, the purpose of the TC is to generate an interrupt at a fixed rate. Actually, such an operation is possible in both the Capture and Waveform mode. Since no signal is being sampled or generated, there is no reason to choose one mode over the other. However, setting the TC in Waveform mode and outputting the tick on TIOA or TIOB can be helpful for debugging purpose.

Setting the CPCTRG bit of the CMR resets the timer and restarts its clock every time the counter reaches the value programmed in the TC Register C. Generating a specific delay is thus done by choosing the correct value for RC. It is also possible to choose between several different input clocks for the channel, which in practice makes it possible to prescale MCK. Since the timer resolution is 16 bits, using a high prescale factor may be necessary for bigger delays.

Consider the following example: the timer must generate a 1 ms trigger with a 12 MHz main clock frequency. RC must be equal to the number of clock cycles generated during the delay.

The last initialization step is to configure the interrupt whenever the counter reaches the value programmed in RC. At the TC level, is easily done by setting the CPCS bit of the Interrupt Enable Register.

3.2.6.3 *Interrupt Handler*

The first action to do in the handler is to acknowledge the pending interrupt from the peripheral. Otherwise, the latter continues to assert the IRQ line. In the case of a Timer Counter channel, acknowledging is done by reading the corresponding Status Register (SR).

Special care must be taken to avoid having the compiler optimize away a dummy read to this register. In C, this is done by declaring a *volatile* local variable and setting it to the register content. The *volatile* keyword tells the compiler to never optimize accesses (read/write) to a variable.

The rest of the interrupt handler is straightforward. A global variable is incremented with the number of ticks read.

```
__attribute__((__interrupt__)) static void tc_irq( void )
{
  // Increment the ms seconds counter
  tc_tick++;
// clear the interrupt flag of TC channel 0
  AVR32_TC.channel[0].sr;
// specify that an interrupt has been raised
  print_sec = 1;
}
```

3.2.7 Using the General Purpose Input/Output (GPIO) controller

3.2.7.1 *Purpose*

Most pins on 32-bit AVR UC3 microcontroller series can either be used by a peripheral function (e.g. USART, SPI, etc.) or used as generic input/outputs. All these pins are managed by the **General Purpose Input/Output (GPIO)** controller.

A GPIO controller enables the programmer to configure each pin as used by the associated peripheral or as a generic IO. In the second case, the level of the pin can be read/written using several registers of the GPIO controller. Each pin can also have an internal pull-up activated individually.

In addition, the GPIO controller can detect a status change on one or more pins, optionally triggering an interrupt whenever this event occurs.

In the EVK1100 and EVK1104 examples, the GPIO controller manages two LEDs and one button. The buttons are configured to trigger an interrupt when pressed (as defined in Section 3.1.1 on page 2).

In the EVK1105 example, the GPIO controller manages two LEDs and a free J16 slot. The PA13 free slot is configured to trigger an interrupt when manually forced to low level.

3.2.7.2 *Configuring LEDs*

The two GPIOs connected to the LEDs must be configured as outputs, in order to turn them on or off. First, the GPIO control must be enabled in GPIO Enable Register (GPER) by writing the value corresponding the two LED IDs.

GPIO output direction is controlled using the registers Output Driver Enable Register (ODER).

EVK1100:

- Init GPIO51 (PB19)

```
AVR32_GPIO.port[1].oders = 1 << (19 & 0x1F); // The GPIO output driver is
enabled for that pin.
AVR32_GPIO.port[1].gpers = 1 << (19 & 0x1F); // The GPIO module controls
that pin.
```

- Init GPIO54 (PB22)

```
AVR32_GPIO.port[1].oders = 1 << (22 & 0x1F); // The GPIO output driver is
enabled for that pin.
AVR32_GPIO.port[1].gpers = 1 << (22 & 0x1F); // The GPIO module controls
that pin.
```

- Initialize GPIO88 (PX16) as interrupt (pin level change). GPIO88 bit control can be found in gpio port 2 (88/32=>2), bit 24 (88%32=24). First enable the glitch filter on GPIO88.

```
AVR32_GPIO.port[2].gfers = 1 << (24 & 0x1F);
```

- Configure the edge detector on pin change on GPIO88

```
AVR32_GPIO.port[2].imr0c = 1 << (24 & 0x1F);
AVR32_GPIO.port[2].imr1c = 1 << (24 & 0x1F);
```

- Enable interrupt on GPIO88

```
AVR32_GPIO.port[2].iers = 1 << (24 & 0x1F);
```

EVK1104:

- Init GPIO67 (PX16)

```
AVR32_GPIO.port[2].oders = 1 << (3 & 0x1F); // The GPIO output driver is
enabled for that pin.
AVR32_GPIO.port[2].gpers = 1 << (3 & 0x1F); // The GPIO module controls
that pin.
```

- Init GPIO101 (PX50)

```
AVR32_GPIO.port[3].oders = 1 << (5 & 0x1F); // The GPIO output driver is
enabled for that pin.
AVR32_GPIO.port[3].gpers = 1 << (5 & 0x1F); // The GPIO module controls
that pin.
```

- Initialize GPIO42 (PB10) as interrupt (pin level change). GPIO42 bit control can be found in gpio port 1 (42/32=>1), bit 10 (42%32=10). First enable the glitch filter on GPIO42.

```
AVR32_GPIO.port[1].gfers = 1 << (10 & 0x1F);
```

- Configure the edge detector on pin change on GPIO42

```
AVR32_GPIO.port[1].imr0c = 1 << (10 & 0x1F);
AVR32_GPIO.port[1].imr1c = 1 << (10 & 0x1F);
```

- Enable interrupt on GPIO42

```
AVR32_GPIO.port[1].iers = 1 << (10 & 0x1F);
```

EVK1105:

- Init GPIO59 (PB27)

```
AVR32_GPIO.port[1].oders = 1 << (27 & 0x1F); // The GPIO output driver is
enabled for that pin.
AVR32_GPIO.port[1].gpers = 1 << (27 & 0x1F); // The GPIO module controls
that pin.
```

- Init GPIO60 (PB28)

```
  AVR32_GPIO.port[1].oders = 1 << (28 & 0x1F); // The GPIO output driver is
enabled for that pin.
  AVR32_GPIO.port[1].gpers = 1 << (28 & 0x1F); // The GPIO module controls
that pin.
```

- Initialize GPIO13 (PA13) as interrupt (pin level change). GPIO13 bit control can be found in gpio port 0 (13/32=>0), bit 13 (13%32=13). First enable the glitch filter and pull up on GPIO13.

```
AVR32_GPIO.port[0].gfers = 1 << (13 & 0x1F);
AVR32_GPIO.port[0].puers = 1 << (13 & 0x1F);
```

- Configure the edge detector on pin change on GPIO13

```
AVR32_GPIO.port[0].imr0c = 1 << (13 & 0x1F);
AVR32_GPIO.port[0].imr1c = 1 << (13 & 0x1F);
```

- Enable interrupt on GPIO13

```
AVR32_GPIO.port[0].iers = 1 << (13 & 0x1F);
```

3.2.7.3 *Controlling LEDs*

LEDs are turned on or off by changing the level on the GPIOs to which they are connected. After those GPIOs have been configured, their output values can be changed by writing the pin IDs in the Output Value Register Toggle (OVRT) of the GPIO controller.

EVK1100:

Toggle the GPIO51 (PB19);

```
AVR32_GPIO.port[1].ovrt  = 1 << (19 & 0x1F);
```

EVK1104:

Toggle the GPIO67 (PX16);

```
AVR32_GPIO.port[2].ovrt  = 1 << (3 & 0x1F);
```

EVK1105:

Toggle the GPIO59 (PB27);

```
AVR32_GPIO.port[1].ovrt  = 1 << (27 & 0x1F);
```

3.2.7.4 *Configuring Buttons*

As stated previously, the GPIO connected to the push button on the board shall be input. Also, a “state change” interrupt is configured. This triggers an interrupt when a button is pressed or released.

After the GPIO control has been enabled on the GPIOs (by writing GPER), it is configured as inputs by writing its IDs in ODER.

Enabling interrupts on the pins is simply done in the Interrupt Enable Register (IER). However, the GPIO controller interrupt must be configured as described in Section 3.2.6.3 on page 8.

3.2.7.5 *Interrupt Handler*

The interrupt handler for the GPIO controller detect a state change level on the pin. This corresponds to either the press or the release action on the button or in case of the EVK1105 example to wire manually GPIO13 to the ground.

Note that the interrupt must be acknowledged in the GPIO controller. This is done implicitly when IFR is read by the software.

EVK1100:

```
__attribute__((__interrupt__)) static void gpio_irq( void )
{
  // GPIO88 (PX16) is connected to push button 0.
  // GPIO88 bit control can be found in gpio port 2 (88/32=>2), bit 24
(88%32=24).
  AVR32_GPIO.port[2].ifrc = 1<<24;

  // Toggle the I/O line GPIO54 (PB22)
  // GPIO54 bit control can be found in gpio port 1 (54/32=>1), bit 22
(54%32=22).
  AVR32_GPIO.port[1].ovrt  = 1 << (22 & 0x1F);
}
```

EVK1104:

```
__attribute__((__interrupt__)) static void gpio_irq( void )
{
  // GPIO42 (PB10) is connected to push button SW2.
  // GPIO42 bit control can be found in gpio port 1 (42/32=>1), bit 10
(42%32=10).
  AVR32_GPIO.port[1].ifrc = 1<<10;

  // Toggle the I/O line GPIO101 (PX50)
  // GPIO101 bit control can be found in gpio port 3 (101/32=>3), bit 5
(101%32=5).
  AVR32_GPIO.port[3].ovrt  = 1 << (5 & 0x1F);
}
```

EVK1105:

```
__attribute__((__interrupt__)) static void gpio_irq( void )
{
  // GPIO13 (PA13) J16 right hole.
  // GPIO13 bit control can be found in gpio port 0 (13/32=>0), bit 13
(13%32=13).
  AVR32_GPIO.port[0].ifrc = 1<<13;

  // Toggle the I/O line GPIO60 (PB28)
  // GPIO60 bit control can be found in gpio port 1 (60/32=>1), bit 28
(60%32=28).
  AVR32_GPIO.port[1].ovrt  = 1 << (28 & 0x1F);
}
```

4. Loading the Project

The development environment for this getting started is a PC running Microsoft Windows OS or Linux.

The required software tools for building the project and loading the binary file is the AVR32 GNU toolchain (available at www.atmel.com).

The connection between the PC and the board is achieved with a USB cable and a JTAGICE mkII debugger.

4.1 Building the Project

The AVR32 GNU toolchain provides assembler, compiler, linker and flash programing tools. Useful programs for debug are also included. AVR32Studio is a free Integrated Development Environment (IDE) for 32-bit AVR UC3 series that enables you to write, build, deploy and debug your C/C++ and assembler code.

4.1.1 AVR32Studio

Refer to the application note AVR32105: AVR32Studio Getting Started, and in particular the section Creating a new AVR32 project, then Adding files to the project

4.1.2 Standalone Makefile

The Makefile contains rules indicating how to assemble, compile and link the project source files to create a binary file ready to be downloaded on the target.

A config.mk file contains the variables settings, the other for rules implementation are contained in the Makefile.

4.1.2.1 Variables

The config.mk file contains variables (uppercase), used to set up some environment parameters, such as the compiler toolchain prefix and program names, and options to be used with the compiler.

EVK1100 & EVK1105:

- GCC Architecture and parts: the AVR UC3 architecture and part number.
  - ARCH = ucr2
  - PART = uc3a0512
- Flash memories: [{cfi|internal}@address,size]...
  - FLASH = internal@0x80000000,512Kb
- Clock source to use when programming: [{xtal|extclk|int}]
  - PROG_CLOCK = xtal
- Target name: {*.a|*.elf}
  - TARGET = $(PART)-getting_started.elf
- C and Assembler source files
  - ./main.c \
  - ./intc.c
- Optimizations:
  - OPTIMIZATION = -O0 -ffunction-sections -fdata-sections

- Extra flags to use when linking
  - LD_EXTRA_FLAGS = -Wl,--gc-sections

EVK1104:

- GCC Architecture and parts: the AVR UC3 architecture and part number.
  - ARCH = ucr2
  - PART = uc3a0256
- Flash memories: [{cfi|internal}@address,size]...
  - FLASH = internal@0x80000000,256Kb
- Clock source to use when programming: [{xtal|extclk|int}]
  - PROG_CLOCK = xtal
- Target name: {*.a|*.elf}
  - TARGET = $(PART)-getting_started.elf
- C and Assembler source files
  - ./main.c \
  - ./intc.c
- Optimizations:
  - OPTIMIZATION = -O0 -ffunction-sections -fdata-sections
- Extra flags to use when linking
  - LD_EXTRA_FLAGS = -Wl,--gc-sections

For more detailed information about gcc options, please refer to gcc documentation (gcc.gnu.org).

4.1.2.2 Rules

The Makefile contains rules. Each rule is composed on the same line by a target name, and the files needed to create this target.

The first rule, ‘all’, is the default rule used by the make command if none is specified in command line.

Table 4-1. Make Goal List

Make Goal	Description
[all]	Default goal: build the project
clean	Clean up the project
rebuild	Rebuild the project
ccversion	Display CC version information
cppfiles file.i	Generate preprocessed files from C source files.
asfiles file.x	Generate preprocessed assembler files from C and assembler source files.
objfiles file.o	Generate object files from C and assembler source files.
a file.a	Archive: create A output file from object files
elf file.elf	Link: create ELF output file from object files

Table 4-1. Make Goal List (Continued)

Make Goal	Description
lss file.lss	Create extended listing from target output file
sym file.sym	Create symbol table from target output file
sizes	Display target size information
isp	Use ISP instead of JTAGICE mkII when programming.
cpuinfo	Get CPU information.
halt	Stop CPU execution.
chiperase	Perform a JTAG Chip Erase command
erase	Perform a flash chip erase.
program	Program MCU memory from ELF output file
secureflash	Protect chip by setting security bit
reset	Reset MCU.
debug	Open a debug connection with the MCU
run	Start CPU execution
readregs	Read CPU registers
doc	Build the documentation
cleandoc	Clean up the documentation
rebuilddoc	Rebuild the documentation
verbose	Display main executed commands

To build the project, type:

```
make all
```

It compiles source files and links object files together to generate one binary file (program running in Flash). It describes how to compile source files and link object files together. This generates an elf format file, which is converted to a binary file without any debug information by using the objcopy program.

4.2 Loading the Code

Once the build step is completed, one .elf file is available and ready to be loaded into the board.

The AVR UC3 ISP solution offers an easy way to download files into 32-bit AVR products on Atmel Evaluation Kits through a USB or the JTAG link. Target programming is done here via the JTAGICE mkII debugger tools.

EVK1100:

Follow the steps below to load and run the code:

- Shut down the board
- Plug the USB cable between the PC and the EVK1100
- Plug the JTAGICE mkII between the PC and the EVK1100
- Power on the board
- To load with the standalone Makefile, open a shell and type:

```
make program run
```

- To load from AVR32Studio, refer to application note AVR32015.

Note: The AT32UC3A0512 is pre programmed with a USB bootloader protected with the BOOTPROT fuse. The only to program again the flash through JTAG is to send a chiperase command. To do this, type in a shell: “avr32program chiperase”.

The code then starts running, LED5 blinks in red every 1sec and the LED6 green is controlled by the push button 0.

EVK1104:

Follow the steps below to load and run the code:

- Shut down the board
- Plug the USB cable between the PC and the EVK1104
- Plug the JTAGICE mkII between the PC and the EVK1104
- Power on the board
- To load with the standalone Makefile, open a shell and type:

```
make program run
```

- To load from AVR32Studio, refer to application note AVR32015.

Note: The AT32UC3A3256 is pre programmed with a USB bootloader protected with the BOOTPROT fuse. The only to program again the flash through JTAG is to send a chiperase command. To do this, type in a shell: “avr32program chiperase”.

The code then starts running, LED0 blinks every 1sec and the LED1 is controlled by the push button SW2.

EVK1105:

Follow the steps below to load and run the code:

- Shut down the board
- Plug the USB cable between the PC and the EVK1105
- Plug the JTAGICE mkII between the PC and the EVK1105
- Power on the board
- To load with the standalone Makefile, open a shell and type:

```
make program run
```

- To load from AVR32Studio, refer to application note AVR32015.

Note: The AT32UC3A0512 is pre programmed with a USB bootloader protected with the BOOTPROT fuse. The only to program again the flash through JTAG is to send a chiperase command. To do this, type in a shell: “avr32program chiperase”.

The code then starts running, LED0 blinks every 1sec and the LED1 is controlled by wiring to ground the PA13 (J16).

4.3 Debug Support

When debugging the Getting Started example with GDB, it is best to disable compiler optimizations. Otherwise, the source code will not correctly match the actual execution of the program. To do that, simply comment out (with a ‘#’) the “OPTIM = ” line of the makefile and rebuild the project.

For more information on debugging with GDB, please refer to the “GNU-Based Software Development” application note and to the GDB manual available on gcc.gnu.org.

5. Revision History

Table 5-1.

Document Ref.	Comments	Change Request Ref.
32078A	First issue.	
32078B	Fixed GPIO LED Control.	
32078C	Updated and extended to EVK1104 and EVK1105.	

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
www.atmel.com

Technical Support
Enter Product Line E-mail

Sales Contact
www.atmel.com/contacts

Literature Requests
www.atmel.com/literature