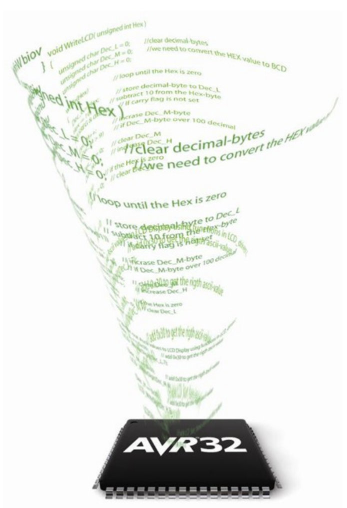
AVR32 UC3B Hands-on 01: Using Low-Level Drivers of the UC3B Software Framework

32-bit AVR32 Microcontrollers

Prerequisites

- **Hands-On**
  - **AVR32 Studio Hands-On 01**
- **Knowledge Requirements**
  - **Basic understanding of microcontrollers**
  - **Basic knowledge of the C language**
  - **Basic knowledge of Integrated Development Tools**
- **PC Platform**
  - **Windows® 2000, Windows® XP, Windows® Vista**
- **Software Requirements**
  - **AVR32 Studio V2**
  - **AVR32 GNU Tool-chain**
- **Hardware**
  - **UC3B Evaluation Kit EVK1101**
  - **JTAGICE mkII**
- **Estimated Completion Time**
  - **90 minutes**



Introduction

The purpose of this hands-on is to use the AVR32 Studio and the Software Framework to create a custom application.

The following items will be reviewed:

- AVR32 Studio & GNU tool chain
  - Adding peripheral driver from the Software Driver
  - Compile
  - Debug
- Software Framework
  - Power Management driver
  - GPIO driver
  - Timer/Counter driver
  - Interrupt driver
  - PWM driver

Rev. Training UC3-0002

Document Overview

The hands-on is split into several different assignments. Each assignment is further divided into individual sections to simplify understanding.

Throughout this document you will find some special icons. These icons are used to identify different sections of assignments and ease complexity.

- **Information**
  Delivers contextual information about a specific topic
- **Tip**
  Highlights useful tips and techniques
- ***To do***
  Highlights objectives to be completed in *italicized text*
- Result
  Highlights the expected result of an assignment step
- **Warning**
  Indicates important information

Abbreviations

- HID Human Interface Device
- ISP In System Programming

Table of Contents

Prerequisites .... **_1_**

Introduction .... **_1_**

Document Overview .... **_2_**

Abbreviations .... **_2_**

Table of Contents .... **_3_**

Software Requirement .... **_5_**

Development tools 5

Software Framework 5

Hardware Requirement .... **_6_**

Evaluation Kit 6

Emulator 6

Hands-On .... **_7_**

Objectives 7

The UC3 Software Framework 7

Setup 7

Hands-on prerequisite 7

AVR32 Studio 8

Hands-On - Assignment 1 .... **_12_**

Objectives 12

UC3B General Purpose Input/Output Controller (GPIO) 12

Software Framework GPIO Driver 12

Exercises 13

A1 – Step 0 14

A1 – Step 1 15

A1 – Step 2 15

A1 – Step 3 15

A1 – Step 4 16

Summary 16

Hands-On - Assignment 2 .... **_17_**

Objectives 17

UC3B Power Manager (PM) 17

Software Framework PM Driver 17

Exercises 18

A2 – Step 0 18

A2 – Step 1 18

Summary 18

Hands-On - Assignment 3 .... **_19_**

Objectives 19

UC3B Timer/Counter....19

Software Framework TC Driver....20

UC3B Interrupt Controller....20

Software Framework INTC Driver....21

Exercises....21
- A3 – Step 0....21
- A3 – Step 1....22
- A3 – Step 2....23
- A3 – Step 3....24

Summary....25

Hands-On - Assignment 4....26

Objectives....26

UC3B Pulse Width Modulation....26

Software Framework PWM Driver....26

Exercises....27
- A4 – Step 0....27
- A4 – Step 1....27
- A4 – Step 2....28

Summary....29

Hands-On Summary....29

Resources....30

Atmel Technical Support Resources....30

Software Requirement

Development tools

- **AVR32 Studio**
  As stated in the Prerequisites section, the installation of the AVR32 Studio as well as the GNU tool-chain must be done prior to the beginning of this hands-on session.

Software Framework

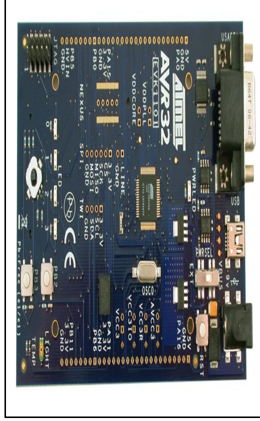
- **AVR32 Software Framework**
  The UC3 Software Framework is part of the AVR32 Studio installation and does not require any specific installation.

Hardware Requirement

Evaluation Kit

- EVK1101

  The EVK1101 is the AVR32 UC3B product series evaluation kit.



- This hands-on will utilize the UC3B EVK1101 board exclusively. However, it can be easily modified to support the UC3A EVK1100 board.

Emulator

- JTAGICE mkII

  The JTAGICE mkII is the programmer and emulator tool for all AVR Microcontrollers, including the UC3 product family.

Hands-On

Objectives

The goal of this hands-on exercise is to develop a step-by-step application that makes LEDs blink using different methods.

In this hands-on you will:

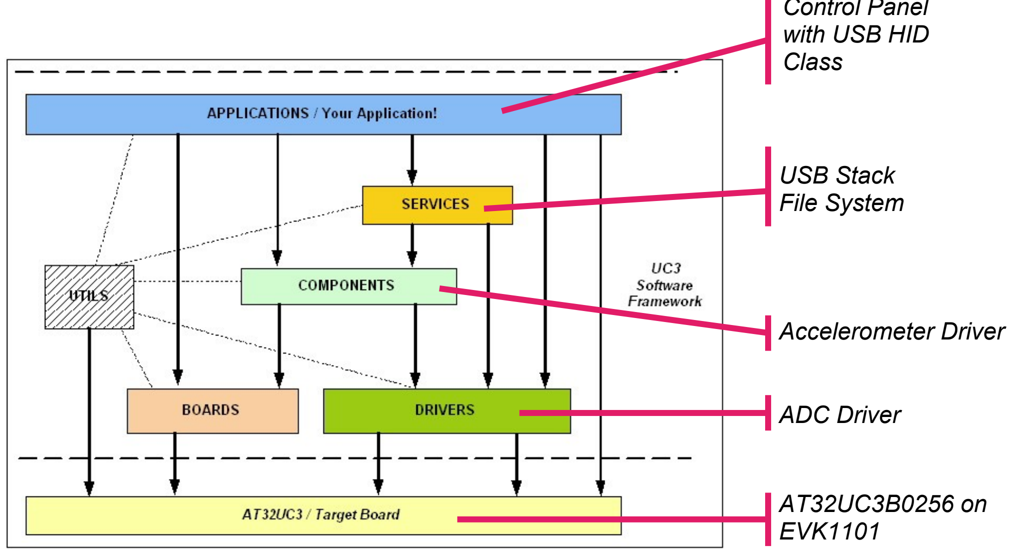
- Use the Drivers modules of the Software Framework
- Develop and debug the application with the AVR32 Studio

The UC3 Software Framework

The Software Framework consists of AVR32 UC3 microcontroller drivers, software services and demonstration applications.

Each software module is provided with full source code, examples, rich html documentation and ready-to-use projects for the IAR EWAVR32 and GNU GCC compilers.

The below diagram shows the Software Framework architecture along with an example based on the EVK1101 control panel application.



Setup

Hands-on prerequisite

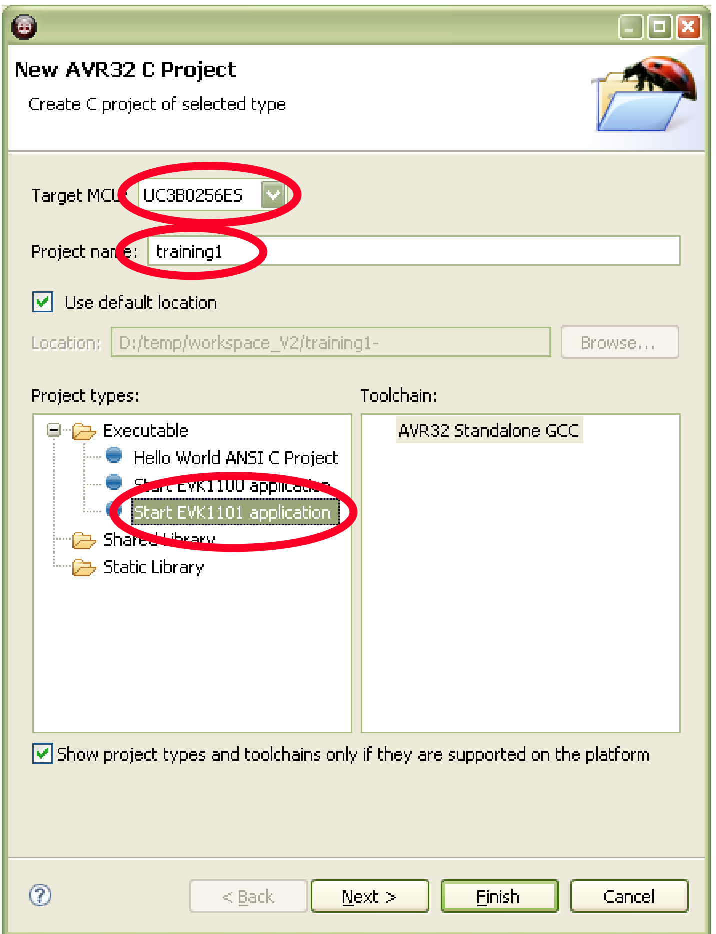
By completing the “AVR32 Studio Hands-On 01” you have learned how to:

- Create a workspace
- Create a project
- Create the target
- Setup the evaluation Kit
- Setup the JTAGICE mkII
- Compile and Debug

Creation

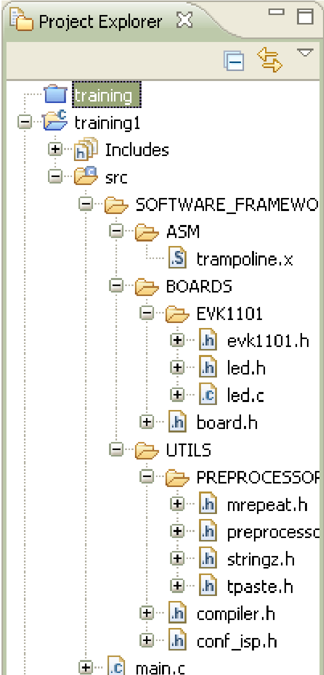
You will now create a new project using the “Project Creation From Template” wizard and import a new main file that contains the skeleton code for this hands-on.

- To avoid confusion with multiple projects within the same workspace, it is recommended to close all the open projects by doing the following:
  - right click on the project name
  - select the “Close Project” item
  - ***Create the project:***
  - *In the “Project Explorer” view click right to select the “New / AVR32 C Project From Template” item*
- The “New AVR32 C Project” wizard opens
  - *Fill in the Project Name as “training1”*
    - *Select the target MCU: UC3B0256ES*
  - *Expand the “Executable” folder and select the project template depending on the selected MCU:*
    - *Start EVK1101 application*



  - *Click “Finish”*

- Your project with expanded folders looks like this:



Below is some information about the Software Framework files included in the project:

trampoline.x
this file contains assembly code that allows to bypass the boot loader location and place the user's code above the boot loader location.

evk1101.h
this file contains definitions and services related to the features of the EVK1101 board

led.h
led.c
these files are the LED component driver of the EVK1101

board.h
this file includes the appropriate board header file according to the defined board

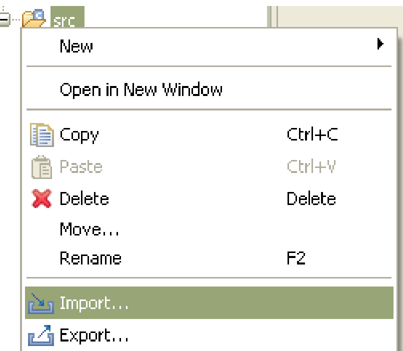
mrepeat.h
preprocessor.h
stringz.h
tpaste.h
these files contain some preprocessor macro-function utilities

compiler.h
this file defines commonly used types and macros

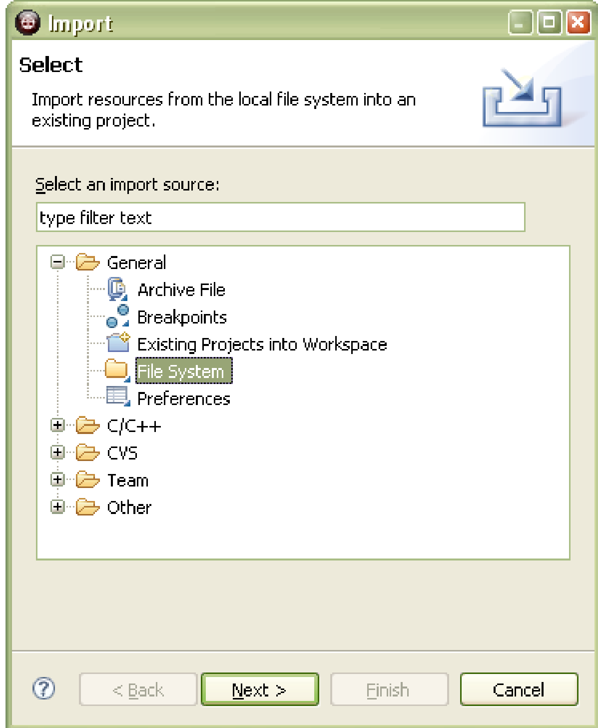
conf_isp.h
this file contains the external configuration of the ISP

- **AVR32 Studio – Software Framework**
  Note that the Software Framework source and header files are all located inside the SOFTWARE_FRAMEWORK folder.

- ***Add the new main file:***
- *In the "Project Explorer" view, right click on the "src" folder and select the "Import…" item:*



- The "Import" wizard opens:



- *Expand the "General Folder" and select the "File System" Item*
- *Click "Next"*
- *Browse to the location where you unzipped the training package and select the "*`Atmel\AVR32 UC3 Training\Hands-On\AVR32 UC3B\01\setup`*" folder*
- *Select the "main.c" file*
- *Click "Finish"*
- *Click "Yes" to overwrite the existing "main.c" file*

- Your project is now ready to run.

- **AVR32 Studio – Project Explorer View**
  If you mistakenly imported the file in the wrong project folder, you can move it to the "src" folder using drag and drop.

Tasks View

You will now open the "Tasks" view in order to have access to the TODO hands-on bookmarks. Double clicking on a TODO bookmark will take you to the corresponding entry in the main file.

- ***Open the Tasks view:***
- *Select the "Show Views/Other…" item of the Windows menu.*
- *Expand the General folder.*
- *Select the "Tasks" item and click OK.*

- You are now ready to run the first assignment of this hands-on.

- **AVR32 Studio – Quick Access**
  Another way to open the "Tasks" view is to use the AVR32 Studio "Quick Access Navigation" pop-up by doing the following:
  - Press CTRL+3 to open the pop-up and type "`tasks`" to see all the related items linked to this keyword.
  - Click on the "Views – Tasks" item to open the "Tasks" view.

Hands-On - Assignment 1

Objectives

The goal of this assignment is to set-up some I/O ports in output and input modes to respectively drive the board LEDS and monitor the board push buttons.

In this Assignment you will:

- Use the GPIO driver module of the Software Framework
- Compile and debug the application with the AVR32 Studio

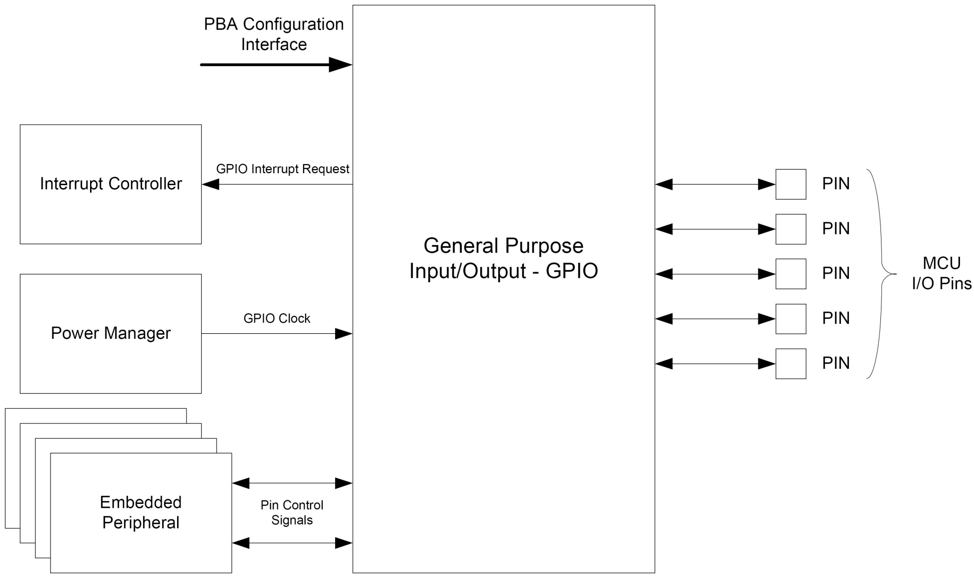
UC3B General Purpose Input/Output Controller (GPIO)

The General Purpose Input/Output manages the I/O pins of the microcontroller. Each I/O line may be dedicated as a general-purpose I/O or be assigned to a function of an embedded peripheral. This assures effective optimization of the pins of a product.

Each I/O line of the GPIO features:

- Configurable pin-change, rising-edge or falling-edge interrupts on any I/O line
- A glitch filter providing rejection of pulses shorter than one clock cycle
- Open Drain mode enabling sharing of an I/O line between the MCU and external components
- Programmable internal pull-up resistor
- Input visibility and output control
- Multiplexing of up to three peripheral functions per I/O line

The following figure illustrates the GPIO block diagram.



Software Framework GPIO Driver

The GPIO driver is split between two files that define a useful set of functions for the GPIO controller:

Below is an abstract of some functions of the Software Framework used during this hands-on:

- `gpio_enable_module_pin(…)`
  enables an alternate function of a GPIO pin
- `gpio_enable_gpio_pin(…)`
  enables the GPIO module to control the pin
- `gpio_set_gpio_pin(…)`
  drives a GPIO pin value to 1
- `gpio_clr_gpio_pin(…)`
  drives a GPIO pin value to 0
- `gpio_tgl_gpio_pin(…)`
  toggles a GPIO pin value
- `gpio_get_pin_value(…)`
  returns the pin value

- **AVR32 Studio – AVR32 Header files**
  Expanding any header files inside the Project Explorer view (by clicking the + widget in front of its name) shows all the definitions and declarations of the module.

Exercises

- **Software Framework**
  The GPIOs of the UC3B are defined in the UC3B header files as follow:
  - Port A: `AVR32_PIN_PAxx` with xx from 00 to 31
  - Port B: `AVR32_PIN_PBxx` with xx from 00 to 11

  This corresponds to a total of 44 GPIO pins.

- **EVK1101**
  The LEDs are connected as:
  - LED0: `PA07`
  - LED1: `PA08`
  - LED2: `PA21`
  - LED3: `PA22`

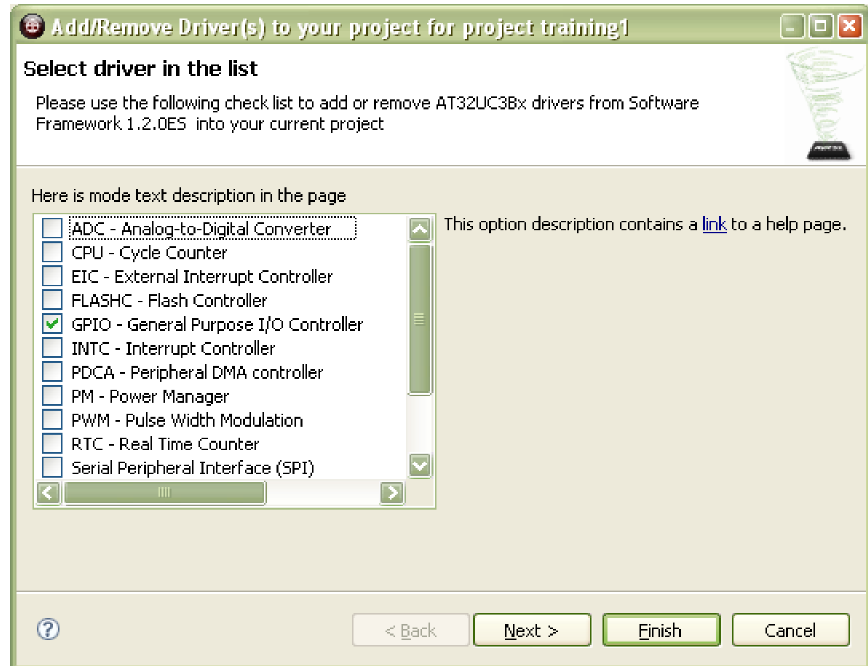
  The Push Buttons are connected as:
  - BP0: `PB02`
  - PB1: `PB03`
  - Joystick Center: `PA13`
  - Joystick Up: `PB07`
  - Joystick Down: `PB08`
  - Joystick Right: `PB06`
  - Joystick Left: `PB09`

A1 – Step 0

Before compiling you need to add the GPIO driver to the project. This is easily done by using the Software Framework "Add/Remove Driver" wizard.

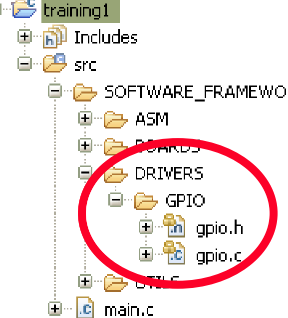
- ***Add the GPIO driver:***
- *In the "Framework" Menu, select the "Add/Remove Driver(s)" item*

- The Driver Import wizard opens:
  - *Select the "General Purpose I/O Controller" driver*



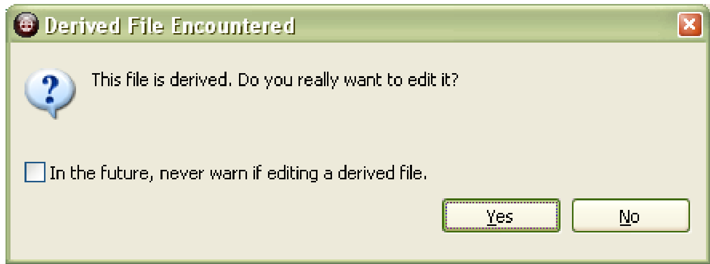
  - *Click "Finish"*

- The GPIO files "gpio.c" and "gpio.h" are now part of the project inside the SOFTWARE_FRAMEWORK/DRIVERS/GPIO folder:



- **AVR32 Studio – Software Framework plug-in**

  Note the padlock present on the imported files icon: . These files are locked as derived from the Software Framework. However you still have the ability to modify them by acknowledging the warning pop-up when trying to edit:



- **Software Framework – Driver Header File**
  The "gpio.h" header file must then be included in the main file to declare the function prototypes.
  - *Add the following line to the include section of "main.c":*

```
#include "gpio.h"
```

A1 – Step 1

You will now turn on the LED1.

- **EVK1101 – LED Driving**
  The LEDs are turned on by driving a low level on the output pin.
  - ***Clear the LED1 GPIO:***
  - *Go to the `TODO A1-Step 1` bookmark*
  - *Replace the `A1 Step 1.1` comment by:*
    - ***`gpio_clr_gpio_pin(…);`***
      This function takes one argument:
      - *the GPIO pin number: **`AVR32_PIN_P`**….*
  - *Compile and debug this step*
- The LED1 turns on.

A1 – Step 2

You will now turn off the LED0.

- **EVK1101 – LED Driving**
  The LEDs are turned off by driving a high level on the output pin.
  - ***Set the LED0 GPIO:***
  - *Go to the `TODO A1-Step 2` bookmark*
  - *Replace the `A1 Step 2.1` comment by:*
    - ***`gpio_set_gpio_pin(…);`***
      This function takes one argument:
      - *the GPIO pin number: **`AVR32_PIN_P`**….*
  - *Compile and debug this step*
- The LED0 turns off after 1s due to the software delay.

A1 – Step 3

You will now make LED2 blink using a software loop.

- ***Toggle the LED2 GPIO.***
- *Go to the `TODO A1-Step 3` bookmark*

- *Replace the `A1 Step 3.1` comment by:*
  - ***gpio_tgl_gpio_pin(…);***
    This function takes one argument:
    - *the GPIO pin number:* ***AVR32_PIN_P****….*
- *Compile and debug this step*
- *Press now PB0*
- *What happens to LED2? Why?*

- The LED2 blinks at a period of 6s based on the software loop.

A1 – Step 4

You will now turn the LED0 on or off depending on the push-button state PB0:

- PB0 pressed: LED on
- PB0 released: LED off

- **EVK1101 – Push-button State**
  Push buttons drive a low level on the pin when pressed.

- ***Read PB0 and reflect its state on LED0 GPIO:***
- *Go to the `TODO A1-Step 4` bookmark*
- *Fill in the `A1 Step 4.1 and A1 Step 4.3` comment of the test operators* ***if*** *() and* ***while*** *() using:*
  - ***gpio_get_pin_value(…);***
    This function takes one argument:
    - *the GPIO pin number:* ***AVR32_PIN_P****….*
- *Replace the `A1 Step 4.2 and A1 Step 4.4` comment by some previously used GPIO functions*
- *Compile and debug this step*

- The LED0 turns on when PB0 is pressed and turns off when PB0 is released.

- **EVK1101**
  All GPIOs connected to push buttons and joystick have external 10KΩ pull-up resistor. These pull-ups could be replaced by the GPIO internal pull-up using:

- **`gpio_enable_pin_pull_up(AVR32_PIN_PY_xx);`**

Summary

The above exercise illustrates how to:

- Use a pin in GPIO mode
- Control the GPIO in output mode
- Read a GPIO pin in input mode

Hands-On - Assignment 2

Objectives

The goal of this assignment is to set-up the crystal oscillator in order to speed-up the CPU execution as well as the peripheral clocks.

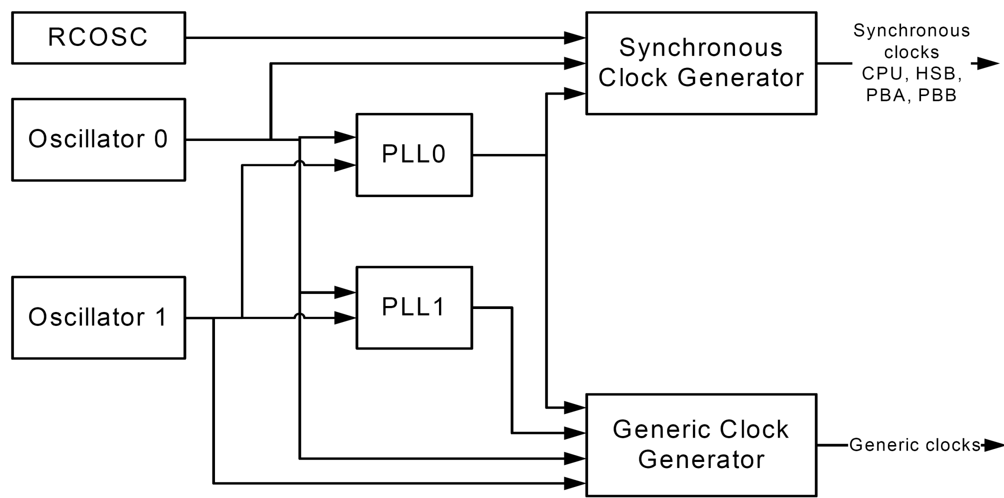
In this Assignment you will:

- Use the PM driver module of the Software Framework
- Compile and debug the application with the AVR32 Studio

UC3B Power Manager (PM)

The Power Manager (PM) controls the oscillators, PLLs, clock generation, BOD, and reset circuitry. The PM also contains advanced power-saving features, allowing the user to optimize the power consumption for an application.

The following figure illustrates the TC block diagram.



Clock generation is divided into two groups: synchronous and generic clocks. The generic clocks are highly tunable asynchronous clocks suitable for peripherals that required specific frequencies, such as timers and communication modules.

The synchronous clocks are divided into three domains: one for the CPU and High Speed Bus (HSB), one for modules on the Peripheral Bus A (PBA), and one for modules on the Peripheral Bus B (PBB). The three clock domains can run at different speeds, so the user can save power by running peripherals at a relatively low clock, while maintaining a high CPU performance.

Software Framework PM Driver

The PM driver is split between two files that define a useful set of functions for the PM controller:

pm.c
pm.h

Below is an abstract of a function of the Software Framework used during this hands-on exercise:

- `pm_switch_to_osc0(…)`
  enables the crystal oscillator 0 and switches the main clock to oscillator 0.

Exercises

A2 – Step 0

Before starting this assignment, you need to add the PM driver to the project using the Software Framework “Driver Import” wizard as done in step 0 of assignment 1.

- ***Add the PM driver:***
- *Add the PM driver using the Software Framework “Driver Import” wizard*

- Do not forget to include the PM header file in “main.c” by adding the following line:

```
#include "pm.h"
```

A2 – Step 1

You will now switch the main clock to the external oscillator 0.

- **AVR32 UC3B**
  By default the UC3 devices start on the internal 115KHz RC oscillator. In order to run at higher frequency, the 12MHz external crystal oscillator must be selected by software.

  - ***Switch to oscillator 0:***
  - *Go to the TODO A2-Step 1 bookmark*
  - *Replace the A2 Step 1 comment by:*
    - ***`pm_switch_to_osc0(…,…,…);`***
      This function takes three arguments:
      - ***`&AVR32_PM`*** *is the address of the Power Manager peripheral module*
      - ***`FOSC0`*** *is the oscillator frequency in Hz*
      - ***`OSC0_STARTUP`*** *is the startup time in RC oscillator periods*
  - *Compile and debug this step*

- The LED2 now blinks 10 times faster than previously; exactly 12,000/115.

- **AVR32 Studio – C Indexer**
  In order to go to the declaration of any C objects (e.g. constant, function, variable, files…) just put the cursor on the word and press F3.

- **AVR32 UC3B**
  Note that running above 33 MHz requires a wait state on the Flash memory access. This is done using the call to the function `flashc_set_wait_state(…)` of the FLASHC driver.

Summary

The above exercise illustrates how to:

- Start and select a new clock source as master clock

Hands-On - Assignment 3

Objectives

The goal of this assignment is to set-up the Timer/Counter with interrupt generation to toggle a LED.

In this Assignment you will:

- Use the Timer/Counter (TC) driver module of the Software Framework
- Use the Interrupt Controller (INTC) driver module of the Software Framework
- Compile and debug the application with the AVR32 Studio

UC3B Timer/Counter

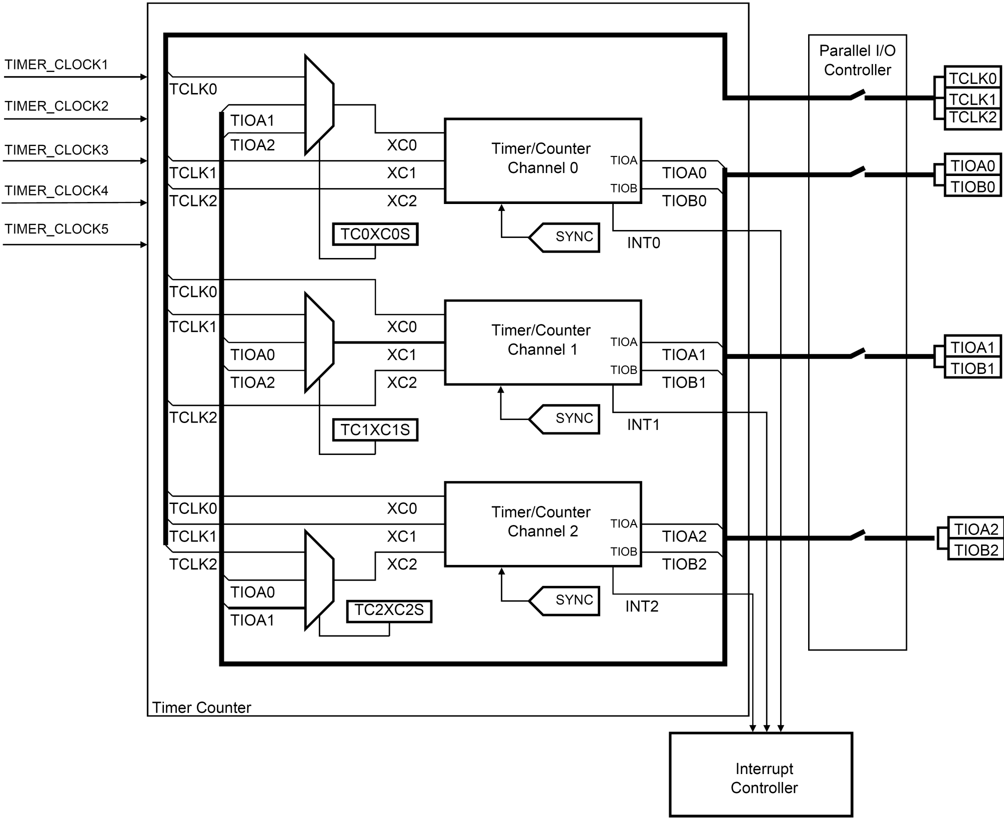
The Timer Counter (TC) includes three identical 16-bit Timer Counter channels.

Each channel has three external clock inputs, five internal clock inputs and two multi-purpose input/output signals which can be configured by the user. Each channel drives an internal interrupt signal which can be programmed to generate processor interrupts.

The Timer/Counter block contains a wide range of functions including:

- Frequency Measurement
- Event Counting
- Interval Measurement
- Pulse Generation
- Delay Timing
- Pulse Width Modulation
- Up/down Capabilities

The following figure illustrates the TC block diagram.



Software Framework TC Driver

The TC driver is split between two files that define a useful set of functions for the Timer/Counter:

tc.c

tc.h

Below is an abstract of some functions of the Software Framework used during this hands-on exercise:

- `tc_init_waveform(…)`
  initializes the timer in waveform mode using the configuration structure
- `tc_write_rc(…)`
  sets the Register C (RC) of the Timer/Counter
- `tc_start(…)`
  starts the Timer/Counter
- `tc_configure_interrupts(…)`
  configures the timer interrupt sources

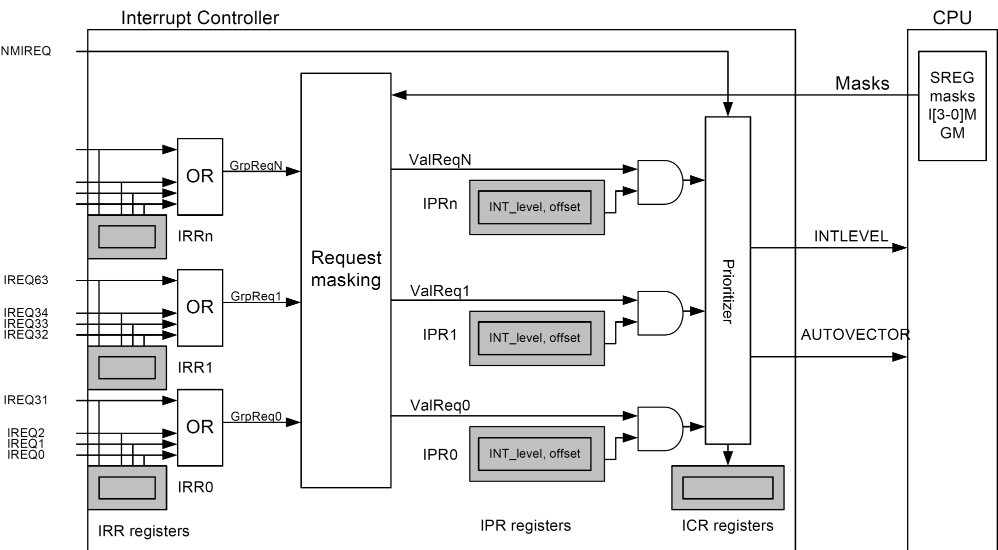
UC3B Interrupt Controller

The Interrupt Controller (INTC) collects interrupt requests from peripherals, prioritizes them, and delivers an interrupt request and an autovector to the CPU. The AVR32 architecture supports 4 priority levels for regular, maskable interrupts, and a Non-Maskable Interrupt (NMI).

When the CPU receives an interrupt request it checks if any other exceptions are pending. If no exceptions of higher priority are pending, interrupt handling is initiated.

Clearing of the interrupt request is done by writing to registers in the corresponding peripheral module, which then clears the corresponding NMIREQ/IREQ signal.

The following figure illustrates the INTC block diagram.



Software Framework INTC Driver

The INTC driver is split between three files that define a useful set of functions for the Interrupt Controller:

- intc.c
- intc.h
- exception.x: this file defines the UC3 vector table.

Below is an abstract of some functions of the Software Framework used during this hands-on exercise:

- `INTC_init_interrupts()`
  Initializes the hardware interrupt controller driver
- `INTC_register_interrupt(…)`
  registers a peripheral interrupt handler and priority to the interrupt driver.

- **GNU Tool Chain – Assembly files**
  Inside the AVR32 Software Framework, there are two kinds of assembly files:

  - file.s
    This file is a pure assembly file that does not request preprocessing.
  - file.x
    This file is an assembly file that requests preprocessing. For instance it includes a C header file. This is the case of exception.x.

Exercises

A3 – Step 0

Before starting this assignment, you need to add the TC and INTC drivers to the project.

- ***Add the TC and INTC driver:***
- *Add the TC and INTC drivers using the Software Framework "Driver Import" wizard*

- Do not forget to include the TC and INTC header files in "main.c" by adding the following lines:

```
#include "tc.h"
#include "intc.h"
```

- ***Uncomment the TC configuration variables:***
- *Go to the TODO A3-Step 0 bookmark*
- *Uncomment the commented block of code as follow:*
  - *Highlight some code inside the commented block*
  - *Click right and select the "Source" ->"Remove Block Comment" item*

A3 – Step 1

You will now initialize the timer as a counter in waveform mode. You will use the channel 0 of the TC in waveform mode 2 without trigger: counter cleared by compare match with the register C.

- ***Configure the timer:***
- *Go to the TODO A3-Step 1 bookmark*
- *Replace the A3 Step 1.1 comment by:*
  - ***`tc_init_waveform(…,…);`***
    This function takes two arguments:
    - ***`&AVR32_TC`*** *is the address of the Timer Counter peripheral module*
    - ***`&WAVEFORM_OPT`*** *is the address of the variable that contains the initialization structure*

- **Software Framework – TC configuration structure**
  The structure is named `tc_waveform_opt_t` and is defined in the "tc.h" file. It contains all the timer configuration fields in waveform mode. The `WAVEFORM_OPT` variable is declared and initialized as follows:

```
// Options for timer waveform generation
static const tc_waveform_opt_t WAVEFORM_OPT =
{
  .channel  = TC_CHANNEL0,           // Channel selection.

  .bswtrg   = TC_EVT_EFFECT_NOOP,    // Software trigger effect on TIOB
  .beevt    = TC_EVT_EFFECT_NOOP,    // External event effect on TIOB
  .bcpc     = TC_EVT_EFFECT_NOOP,    // RC compare effect on TIOB
  .bcpb     = TC_EVT_EFFECT_NOOP,    // RB compare effect on TIOB

  .aswtrg   = TC_EVT_EFFECT_NOOP,    // Software trigger effect on TIOA
  .aeevt    = TC_EVT_EFFECT_NOOP,    // External event effect on TIOA
  .acpc     = TC_EVT_EFFECT_NOOP,    // RC compare effect on TIOA
  .acpa     = TC_EVT_EFFECT_NOOP,    // RA compare effect on TIOA

  .wavsel   = TC_WAVEFORM_SEL_UP_MODE_RC_TRIGGER,// Waveform mode
  .enetrg   = FALSE,                 // External event trigger enable
  .eevt     = 0,                     // External event selection
  .eevtedg  = TC_SEL_NO_EDGE,        // External event edge selection
  .cpcdis   = FALSE,                 // Counter disable when RC compare.
  .cpcstop  = FALSE,                 // Counter clock stopped with RC compare

  .burst    = FALSE,                 // Burst signal selection
  .clki     = FALSE,                 // Clock inversion
  .tcclks   = TC_CLOCK_SOURCE_TC5    // Source clock
```

```
};
```

All the initialization value options of the structure members are detailed in “tc.h”. For instance, the `TC_CLOCK_SOURCE_TC5` corresponds to the master clock divided by 128.

- ***Set the compare register value:***
- *Replace the A3 Step 1.2 comment by:*
    - ***`tc_write_rc(…,…,…);`***
      This function takes three arguments:
        - ***`&AVR32_TC`*** *is the address of the Timer Counter peripheral module*
        - ***`TC_CHANNEL0`*** *is the timer channel to set*
        - ***`0xF000`*** *is the compare value to load in Register C (RC)*

          The period calculation is performed as follows:

          T= Tin x RC = (128 / 12E6) x 61440 = 1.5s
- ***Start the counter:***
- *Replace the A3 Step 1.3 comment by:*
    - ***`tc_start(…,…);`***
      This function takes two arguments:
        - ***`&AVR32_TC`*** *is the address of the Timer Counter peripheral module*
        - ***`TC_CHANNEL0`*** *is the timer channel to start*
- *Compile and debug this step*

- The TC channel 0 is now initialized.
- **UC3 – TC status**
  In order to see if the timer is running, monitor the RC Compare Status (CPCS) bit of the TC/SR0 Status register. It will be set when a compare has occurred since the last read of SR0:

Note that reading SR clears all the register bits.

A3 – Step 2

You will now code the timer Interrupt Service Routine. The LED3 will toggle at each interrupt.

- ***Acknowledge the interrupt:***
- *Go to the TODO A3-Step 2 bookmark*
- *Replace the A3 Step 2.1 comment by:*
  - ***`AVR32_TC.channel[TC_CHANNEL0].sr`***

- The interrupt source of the TC is cleared by reading the SR register. In this hands-on we are using only one interrupt source: the Register C compare status, so we just clear the SR without testing the interrupt source. It becomes necessary to test the SR content bit by bit, if more than one interrupt source is enabled.

- ***Toggle the LED state:***
- *Replace the A3 Step 2.2 comment by one previously used GPIO function*
- *Compile this step*

- **GNU Tool-chain – GCC**
  An interrupt service routine is coded like a function but with the `__interrupt__` attribute:

```
__attribute__((__interrupt__)) void tc_irq( void )
{
  /* TC Interrupt */
  …
}
```

A3 – Step 3

You will now setup the interrupt handler and install the TC interrupt.

- ***Initialize the interrupt handler:***
- *Go to the TODO A3-Step 3 bookmark*
- *Replace the A3 Step 3.2 comment by:*
  - ***`INTC_init_interrupts();`***

- ***Register the timer interrupt:***
- *Replace the A3 Step 3.3 comment by:*
  - ***`INTC_register_interrupt(…,…,…);`***
    This function takes three arguments:
    - ***`&tc_irq`*** *is the address of the TC interrupt service routine*
    - ***`AVR32_TC_IRQ0`*** *is the timer interrupt group number*
    - ***`AVR32_INTC_INT0`*** *is the Interrupt priority level to assign to the group*

- ***Enable the timer interrupt:***
- *Replace the A3 Step 3.3 comment by:*
  - ***`tc_configure_interrupts(…,…,…);`***
    This function takes three arguments:
    - ***`&AVR32_TC`*** *is the address of the Timer Counter peripheral module*
    - ***`TC_IRQ0`*** *is the timer channel to start*
    - ***`&TC_INTERRUPT`*** *is the address of the variable that contains the initialization structure*

- **Software Framework – TC interrupt enable structure**
  The structure is named `tc_interrupt_t` and is defined in the “tc.h” file. It contains all the timer initialization fields. The `TC_INTERRUPT` variable is declared and initialized as follow:

```
static const tc_interrupt_t TC_INTERRUPT =
{
  .etrgs = 0,
  .ldrbs = 0,
  .ldras = 0,
  .cpcs  = 1,
  .cpbs  = 0,
  .cpas  = 0,
  .lovrs = 0,
  .covfs = 0
};
```

  Only the compare Register C status interrupt source is enabled. All the initialization value options of the structure members are detailed in “tc.h”.

  - *Compile and debug this step*
  - *Press now PB0*
  - *Compare LED2 and LED3 status*

- The LED3 blinks every 3 seconds.

Summary

The above exercise illustrates how to:

- Setup a timer channel in time base mode
- Write an interrupt service routine
- Initialize the interrupt handler
- Register an interrupt service routine
- Enable one of the timer interrupt source

Hands-On - Assignment 4

Objectives

The goal of this assignment is to set-up PWM output to control LED brightness.

In this Assignment you will:

- Use the Pulse Width Modulation (PWM) driver module of the Software Framework
- Compile and debug the application with the AVR32 Studio

UC3B Pulse Width Modulation

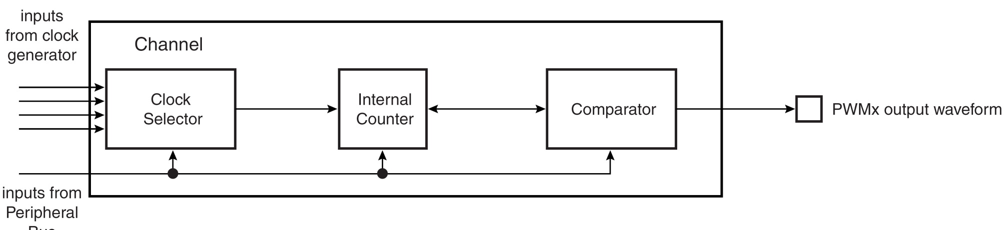
The Pulse Width Modulation (PWM) module contains seven independent PWM channels. Each channel has a 20-bit counter and controls one square output waveform. Characteristics of the output waveform such as period, duty-cycle, and polarity are configurable.

Channels can be synchronized to generate non overlapped waveforms. All channels integrate a double buffering to prevent glitches on the output waveform when modifying the period or the duty-cycle.

Each of the seven PWM Channels contains:

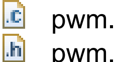
- One 20-bit counter
- Independent enable/disable
- Independent clock selection
- Independent period and duty cycle
- Double buffering of period and duty cycle
- Programmable output waveform polarity
- Programmable center or left aligned output waveform

The following figure illustrates the block diagram for one PWM channel.



Software Framework PWM Driver

The PWM driver is split between two files that define a useful set of functions for the PWM controller:



pwm.c
pwm.h

Below is a list of some functions in the Software Framework used during this hands-on:

- `pwm_init(…)`
  Initializes the PWM controller

- `pwm_channel_init(…)`
  initializes a specified PWM channel with a configuration structure
- `pwm_start_channels(…)`
  starts some PWM channels
- `pwm_interrupt_update_channel(…)`
  updates channel register CPRDx or CDTYx

Exercises

A4 – Step 0

Before starting this assignment, you need to add the PWM driver to the project.

- ***Add the PWM driver:***
- *Add the PWM driver using the Software Framework “Driver Import” wizard*

- Do not forget to include the PWM header file in “main.c” by adding the following line:

```
#include "pwm.h"
```

- ***Uncomment the PWM configuration variables:***
- *Go to the TODO A4-Step 0 bookmark*
- *Uncomment the commented block of code as follow:*
  - *Highlight some code inside the commented block*
  - *Click right and select the “Source” ->”Remove Block Comment” item*

A4 – Step 1

You will now use a PWM channel to control the LED1 brightness. The PWM channel 1 is used as alternate of PA08 which is the LED1 GPIO.

- ***Configure the PWM controller:***
- *Go to the TODO A4-Step 1 bookmark*
- *Replace the A4 Step 1.1 comment by:*
  - ***`pwm_init(…);`***
    This function takes one argument:
    - **`&pwm_opt`** *is the address of the variable that contains the initialization structure*

- **Software Framework – PWM initialization structure**
  The structure is named `pwm_opt_t` and is defined in “pwm.h”. It contains the entire PWM controller clock configuration. The `pwm_opt` variable is declared and initialized as follows:

```
// PWM controller configuration.
pwm_opt.diva = AVR32_PWM_DIVA_CLK_OFF;
pwm_opt.divb = AVR32_PWM_DIVB_CLK_OFF;
pwm_opt.prea = AVR32_PWM_PREA_MCK;
pwm_opt.preb = AVR32_PWM_PREB_MCK;
```

All clocks are derived from the master clock. The initialization values of the structure members are listed in the UC3B PWM controller header file.

- ***Configure the PWM channel:***
- *Replace the A4 Step 1.2 comment by:*

- ***pwm_channel_init(…,…);***
  This function takes two arguments:
  - ***pwm_ch_id*** *is the channel number: 1*
  - ***&pwm_channel*** *is the address of the variable that contains the configuration structure*

- **Software Framework – PWM channel configuration structure**
  The structure is named `avr32_pwm_channel_t` and is defined in the UC3B PWM controller header file. It contains the entire PWM channel configuration fields. The `pwm_opt` variable is declared and initialized as follows:

```
// PWM controller configuration.
pwm_channel.CMR.calg = PWM_MODE_LEFT_ALIGNED;      // Channel mode
pwm_channel.CMR.cpol = PWM_POLARITY_HIGH;          // Channel polarity
pwm_channel.CMR.cpd = PWM_UPDATE_DUTY;             // Update type
pwm_channel.CMR.cpre = AVR32_PWM_CPRE_MCK_DIV_32; // Channel prescaler
pwm_channel.cdty = pwm_duty;            // Channel duty cycle, should be < CPRD
pwm_channel.cprd = 1875;                // Channel period
pwm_channel.cupd = pwm_duty;            // Channel update, should be < CPRD
```

All the initialization value options of the structure members are detailed in the “pwm.h” file.

- ***Start the PWM channel***
- *Replace the A4 Step 1.3 comment by:*
  - ***pwm_start_channels(…);***
    This function takes one arguments:
    - ***1 << pwm_ch_id*** *is a bit field where each bit correspond to a channel*

- ***Enable the PWM function of the GPIO port:***
- *Replace the A4 Step 1.4 comment by:*
  - ***gpio_enable_module_pin(…,…);***
    This function takes two arguments:
    - ***AVR32_PIN_PA08*** *is the LED1 GPIO port*
    - ***AVR32_PWM_PWM_0_0_FUNCTION*** *is the PWM alternate function*
- *Compile and debug this step*

- The LED1 output is dim.

A4 – Step 2

You will now modify the PWM duty cycle to control the LED1 brightness. This can be accomplished by incrementing the CUPD member of the configuration variable `pwm_channel`.

- ***Update the PWM duty cycle:***
- *Replace the A4 Step 2 comment by:*
  - ***pwm_interrupt_update_channel(…,…);***
    - ***pwm_ch_id*** *is the channel number 1*
    - ***&pwm_channel*** *is the address of the variable that contains the configuration structure. Only the cupd member is used.*

- The LED1 brightness slowly increases thanks to the PWM duty cycle modification.

Summary

The above exercise illustrates how to:

- Assign an alternate function to a GPIO port pin
- Setup a PWM channel in fixed frequency and variable duty cycle mode
- Update the PWM duty cycle

Hands-On Summary

The training materials have provided:

- Knowledge of the AVR32 UC3 devices
- Development examples using:
  - UC3 Software Framework
  - AVR32 Studio V2
  - The AVR32 GNU tool-chain
- Application debug using:
  - The JTAGICE mkII emulator
  - The EVK1101 board

Resources

Below is a list of web resources available for the AVR32 products:

- AVR32 Home
  http://www.atmel.com/avr32/
- AVR32 Datasheets
  http://www.atmel.com/dyn/products/datasheets.asp?family_id=682
- EVK1100 Evaluation Kit information
  http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4114
- EVK1101 Evaluation Kit information
  http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4175
- AVR32 Studio
  http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4116
- AVR32 GNU tool-chain
  http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4118

Atmel Technical Support Resources

Atmel has several support channels available:

- Web portal: http://support.atmel.no/ All Atmel microcontrollers
- Email: avr@atmel.com All AVR products
- Email: avr32@atmel.com All AVR32 products

Please register on the web portal to gain access to the following services:

- Access to a rich FAQ database
- Easy submission of technical support requests
- History of all your past support requests
- Register to receive Atmel microcontrollers' newsletters