1 Introduction

The goal of this assignment is to familiarize yourself with the EVK1100 development board and cross compiler tools.

If you need help during the lab-session you should write your name on the whiteboard (first to write should write on top of the board) so that the lab assistant can help you in first-come-first- served order.

2 Introduction to AVR32

AVR32UC3A is a microprocessor architecture from the Atmel company. It is similar, but different, to the ARM architecture from ARM Ltd. The AVR32UC3A core is based on a 3-stage pipeline Harvard architecture. It contains a variety of peripherals.

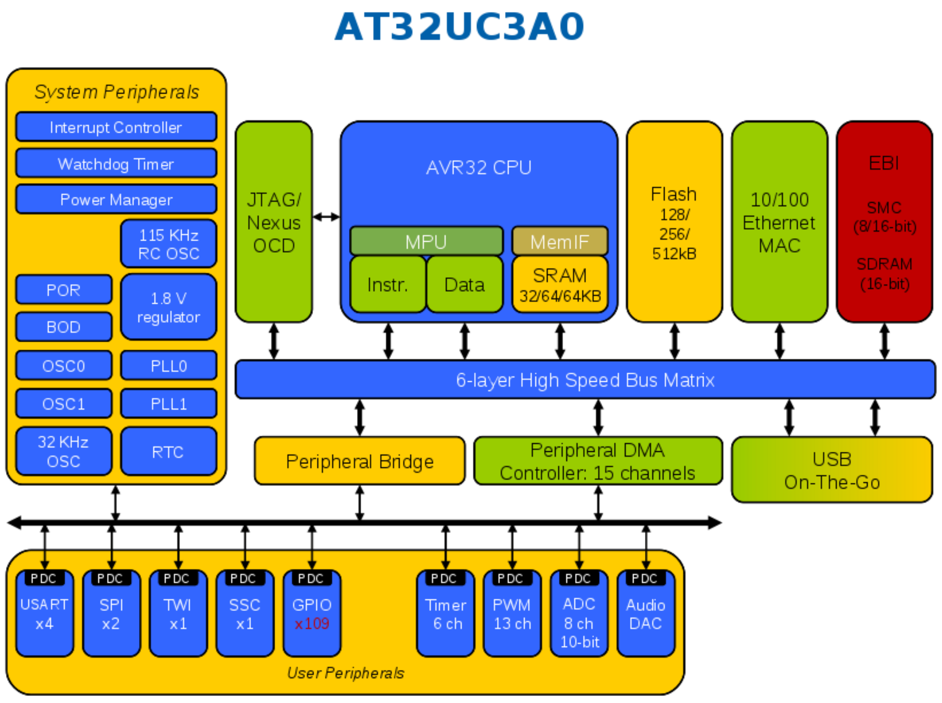


Figure 1: AVR32UC3A overview

2.1 Essential readings

Having these documents handy will help you through the lab.

- AVR32 Studio Getting started:
  - http://www.atmel.com/dyn/resources/prod_documents/doc32086.pdf
- AVR32UC3A0512 datasheet:
  - http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf
- The EVK1100 schematics:
  - http://www.atmel.com/dyn/resources/prod_documents/ /EVK1100_SCHEMATICS_REVC.pdf
- Defines of constants, like the GPIO_PUSH_BUTTON_0 etc, is found in the file *{project}/src/SOFTWARE_FRAMEWORK/BOARS/EVK1100/evk1100.h*

2.2 Abbreviations used

Dev. board = Development board
GPIO = General Purpose Input/Output
IDE = Integrated development environment
JTAG = Joint Test Action Group, a on-chip debug interface
LCD = Liquid crystal display
LED = Light emitting diode
MCU = Microcontroller unit
PWM = Pulse Width Modulation
SPI = Serial peripheral interface
TP = Twisted pair
TWI = Two wire interface
USART = Universal Synchronous Asynchronous Receiver Transmitter

2.3 Clarification of the "Program" confusion

The term "Program" has several meanings. When you write source code in the editor, you are said to "program" or "write the program".

When you write a binary into the dev. board MCU, you are said to "program" the flash, or "write the program (to the flash)".

This can be confusing but is nothing to worry about.

2.4 Required equipment

This equipment should be available in the case you receive from the lab instructor. If anything is missing when you return the case you will not pass the course.

- EVK1100 Development board (Dev. board)
- Serial cable

- USB cable or power adapter to power the EVK1100
- The Dragon JTAG debugger
- 10-pin ribbon cable with block headers
- USB cable for powering and communicating with The Dragon

3 Practise Assignments

3.1 Practise Assignment 1: Prepare Working Environment

Content: Prepare working environment terminal toolchain environment IDE Upgrade of JTAG device (should have been pre-upgraded by lab instructor)

Read through the entire document before applying power to any device. Then re-read and follow the instructions.

3.1.1 Preparations

In order to program and debug a dev. board it is necessary to have a set of tools on the desktop computer. First of all a toolchain consisting of a compiler, linker, debugger, object file converter, JTAG control program etc is needed. The GNU avr32 toolchain is suitable and should be installed on the computer you're going to use. The environment variables of your OS might need to be set properly.

Second, a source code editor and preferably an IDE is needed. Atmel Studio 6 is suitable for development on AVR32 devices. Before running Atmel Studio 6, make sure the debugger is plugged in and powered (The red and green LED on the dragon are lit). Run Atmel Studio 6, it will tell you if any tools are missing or if the debugger need a firmware upgrade. Fix any problem before continuing. Familiarize yourself with the various menu options and views. In the end you only need a small subset, but you need to know where to find them.

Third, a very handy tool when developing is a serial terminal connected via RS-232 to the dev. board. It is convenient due to the relative simplicity of serial communication and the efficiency of having debug messages emitted from the program running on the dev. board. There are various terminal programs. For *nix systems minicom or cutecom might be suitable. For win there is putty and terraterm amongst others.

When you have confirmed the availability of the necessary software tools you can go on with the next task.

3.2 Practise Assignment 2: Familiarizing With the Hardware

Content: Familiarize with the hardware. Identify I/O connectors.

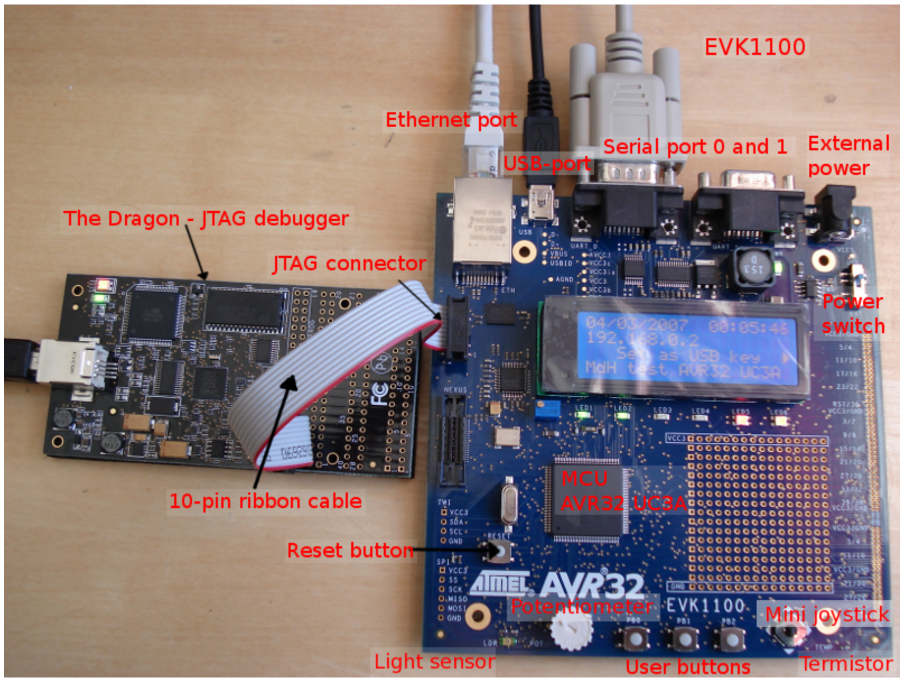


Figure 2: EVK1100

3.2.1 What the Hardware look like:

EVK1100 is an evaluation kit from Atmel. The EVK1100 can be programmed and debugged using a device called The Dragon. It is good to know the location of the connectors and what chip is which on the dev. board. The picture above shows some of the various components on the dev. board.

3.2.2 Some pointers

The RS-232 connector needs a straight pin-to-pin DB9 cable. No crossed null modem cable is necessary. The Ethernet connector takes either straight cables or crossed cables.

The JTAG connector is not keyed, it is possible to insert the JTAG cable with the wrong orientation. This will damage the dev. board and JTAG device. It is important to match pin 1 of the JTAG device to pin 1 of the dev. board JTAG port. Preferably use a 10-line ribbon cable with block headers. Note that one of the lines of the ribbon cable has a red stripe. Use this to orientate and match pin 1 to pin 1. Don't orientate the red stripe to match pin 10 to pin 10, even if there is nothing wrong and it works as good as the pin 1 to pin 1 orientation. Your friend, and the rest of the world who always orientate the red stripe at pin 1, will be confused.

Do not connect any external devices unless you know what you are doing.

If in any doubt at all about this, ask the lab assistant for guidance before proceeding. It will be less embarrassing than to ask for new hardware.

3.2.3 How to handle the hardware

The hardware is sensitive to static electricity and can easily be damaged. Discharge yourself through a metallic, grounded object before unpacking and touching the dev. board. Connecting power to the board require a centrum positive 8-15 V power transformer. Or simply power the board through the mini-USB from your desktop PC. The termistor of the EVK1100 has nickel terminals, which is good to know if you are allergic to nickel.

...and "No", the display of the EVK1100 is not detachable. It is soldered to the PCB.

3.3 Practise Assignment 3: Interactive Development Environment (IDE)

Content: Find, familiarize and use cross compilation tools & IDE. Compile a "Hello LED" program

3.3.1 Hello LED

We will look at how to access and control I/O-pins of the MCU. We will investigate the I/O control by using a LED. It can be useful to signal the status to the user using a LED. There is more than having the LED on or off. A blinking LED might be the only way for the program to tell the user something happened in a simple system. Various blinking rates or intensities of the light can be options for communication with the user. Even a LED that doesn't blink or light up when it is supposed to can be informative.

3.3.2 About AVR32 GPIO

On the AVR32UC3A0512 there is a GPIO (General Purpose Input/Output) controller. There are four 32-bit GPIO ports: PA, PB, PC and PX. Each port has a number of physical pins connected to the MCU depending on how the chip has been constructed.

The GPIO pins are multiplexed with other MCU peripherals, e.g. the USART. Each pin functionality is controlled by a set of hardware registers.

In order to address a specific GPIO pin, the GPIO port number and the pin bit position on the port need to be calculated. To find the corresponding port and pin from a AVR32_PIN_xyz GPIO number, i.e. the pin number of the MCU as they are defined in the header files used by Atmel Studio 6, the following calculations can be used:

```
MAX_PIN_NUMBER = 32
GPIO_PORT = AVR32_PIN_xyz / MAX_PIN_NUMBER      //Port index
GPIO_PIN = AVR32_PIN_xyz & (MAX_PIN_NUMBER - 1) //Bit position
```

For instance, xyz could be substituted with PB27, which is the GPIO pin connected to LED0. However, most useful GPIOs have already been predefined. For instance the name LED0_GPIO can be used instead of AVR32_PIN_PB27.

The GPIO_port number is used as a index to a base address to get the correct hardware register address in the GPIO controller. The base address is defined as AVR32_GPIO.

To get the address of a GPIO port use:

```
volatile avr32_gpio_t * gpio_port = &AVR32_GPIO.port[GPIO_PORT];
```

GPIO_PIN is used to get the correct bit position on the GPIO port. Writing a bit_value in this bit position can affect the GPIO port in one way or another.

To set the GPIO_PIN position on the GPIO_PORT you would do something similar to:

```
bit_value = (1 << GPIO_PIN);  //Put a 1-bit at the correct position
gpio_port->ovrs = bit_value;  //Write the bit to the gpio_port
```

The gpio_port->ovrs is explained below in the hardware registers section. Also see the source code walk through (Hello LED) below how to use the ports and pins.

3.3.3 Hardware registers

The GPIO unit is controlled by hardware registers. Some of the GPIO functions have hardware registers available in four different versions:

1. Read or write the whole register,
2. Set individual bits in the register,
3. Clear individual bits in the register,
4. Toggle individual bits in the register.

We will briefly cover the necessary hardware registers:

Register	Read/Write	Description
gper	**r/w**	**GPIO Enable Register.**
gpers	w	Set bit in gper.
gperc	w	Clear bit in gper.
gpert	w	Toggle bit in gper.
oder	**r/w**	**Output Drive Enable Register**
oders	w	Set bit in oder.
oderc	w	Clear bit in oder.
odert	w	Toggle bit in oder.
ovr	**r/w**	**Output Value Register**
ovrs	w	Set bit in ovr.
ovrc	w	Clear bit in ovr.
ovrt	w	Toggle bit in ovr.
pvr	**r**	**Pin Value Register.**

Table 1: AVR32 GPIO Hardware registers

Each bit in the registers correspond to a GPIO pin.

The Set, Clear and Toggle hardware registers are write only. Only the bits set to one (1) in the Set/Clear/Toggle registers are affected. Bits set to zero (0) remain unaffected in the corresponding r/w register. The result from the Set/Clear/Toggle can be read from the r/w register (e.g. ovr). Masking have to be utilized to read individual bits from the r/w register.

Setting bits in ovr only make sense if the corresponding bits in oder is set.

Reading the pvr value returns the electrical state of the whole GPIO port. Again, masking has to be utilized to read individual bits. pvr is read only.

How to access the registers is shown in the pseudo-example:

Pseudo-Example:

```
int reg_before, reg_after;
volatile avr32_gpio_t * gpio = &AVR32_GPIO.port[GPIO_PORT];

/* Read the ovr register */
reg_before = gpio->ovr;
/* Imagine reg_before equals 0x34235020 at this point */

/* Set bit 8 in gper */
gpio->ovrs = 0x100;

/* Read the ovr register */
reg_after = gpio->ovr;
/* reg_after is now 0x34235120 */

/* Clear bit 5 */
```

```
gpio->ovrc = 0x20;

/* Read the ovr register */
reg_after = gpio->ovr;
/* reg_after is now 0x34235100 */

/* Clear the entire register */
gpio->ovr = 0x0;
```

3.3.4 Source code walk through

We will walk through some source code and examine how the GPIO is set up and controlled.

First we include the necessary files. *board.h* and *compiler.h* are always used when compiling projects in Atmel Studio 6. They contain definitions specific to the EVK1100 board and the MCU, e.g. AVR32_GPIO and LED0_GPIO which we will be using.

```
/****************************************************
 * Name : main.c
 * Author : Marcus Jansson
 * Copyright : Not really
 * Description : EVK1100 Hello LED
 ****************************************************/
/* Include Files */
#include "board.h"
#include "compiler.h"
```

The GPIO pin connected to the physical LED on the EVK1100, marked as "LED1" on the dev. board, is in fact defined as LED0_GPIO in the AVR32 Studio header files.

We define the GPIO_PORT and GPIO_PIN that we will use, and also the BIT_VALUE for the GPIO_PIN. Let us call them LED0_PORT, LED0_PIN and LED0_BIT_VALUE. This is done to make the source a little easier to read and change.

```
/* Define the ports and pins to be used */
//The maximum number of pins in a port
#define GPIO_MAX_PIN_NUMBER 32

//This is the port which LED0_GPIO is located on
#define LED0_PORT (LED0_GPIO / GPIO_MAX_PIN_NUMBER)

//This is the bit position of the GPIO pin
#define LED0_PIN (LED0_GPIO & (GPIO_MAX_PIN_NUMBER -1))

//This is a 1-bit written to the bit position of the GPIO pin
```

```
#define LED0_BIT_VALUE (1 << LED0_PIN)
```

The first thing the program needs to do is to set up the GPIO port to control the LED. This includes choosing the GPIO controller as peripheral (since the IO pins are multiplexed with other peripheral units) and enable the output drive of the GPIO pin connected to one of the EVK1100 LEDs.

```
/*
 * Init a GPIO pin for driving a LED
 * Purpose: Set up the LED GPIO line as output and at
 * value 1 which, contra intuituve, will turn the LED off.
 */
void initLED(void)
{
    /*
     * First make access to the GPIO port registers
     * shorter to type, just for our convenience
     */
    volatile avr32_gpio_port_t * led0_port;
    led0_port = &AVR32_GPIO.port[LED0_PORT];

    /*
     * Choose GPIO peripheral function,
     * by writing a 1 to gpers, GPIO Enable Register Set
     */
    led0_port->gpers = LED0_BIT_VALUE;

    /*
     * Set the output value register to 1 in order to
     * turn the LED off (1=off, 0=on in this case),
     * by writing a 1 to ovrs, Output Value Register Set
     */
    led0_port->ovrs = LED0_BIT_VALUE;

    /*
     * Set the output enable register,
     * by writing a 1 to oders, Output Drive Enable
     * Register Set
     */
    led0_port->oders = LED0_BIT_VALUE;
}
```

This simple main function demonstrate the use of the GPIO pins by setting and clearing a bit in the ovr hardware register, using ovrs (set) and ovrc (clear).

```
int main(void)
{
    initLED();
    /* Main loop that will toggle a single bit on the GPIO port
*/
    while(1) {
        /* Clear register bit */
        AVR32_GPIO.port[LED0_PORT].ovrc = LED0_BIT_VALUE;

        /* Set register bit */
        AVR32_GPIO.port[LED0_PORT].ovrs = LED0_BIT_VALUE;
    }

    /* Do not exit this function */
    while(1);
}
```

An alternative method for toggeling the pin would be to
use ovrt:

```
/* Main loop that will toggle a single bit on the GPIO port
*/
while(1) {
        /* Toggle register bit */
        AVR32_GPIO.port[LED0_PORT].ovrt = LED0_BIT_VALUE;

        /* Toggle register bit again */
        AVR32_GPIO.port[LED0_PORT].ovrt = LED0_BIT_VALUE;
    }
```

3.3.5 Create a new project

Now we will create a new project and compile this source code.

- Use Atmel Studio 6.
- Create a new Project.

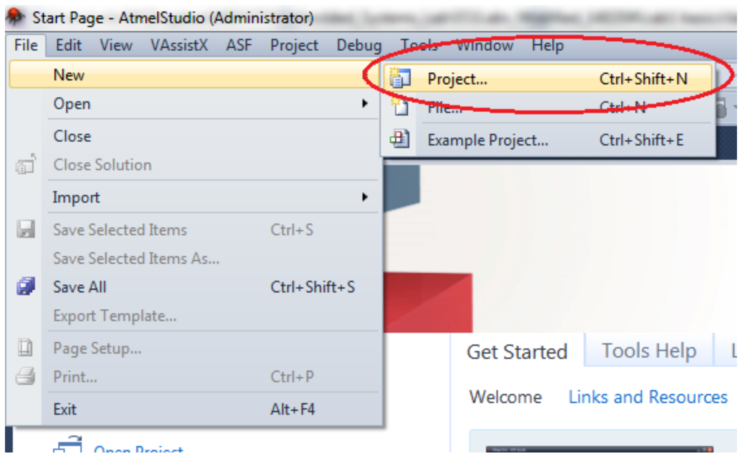
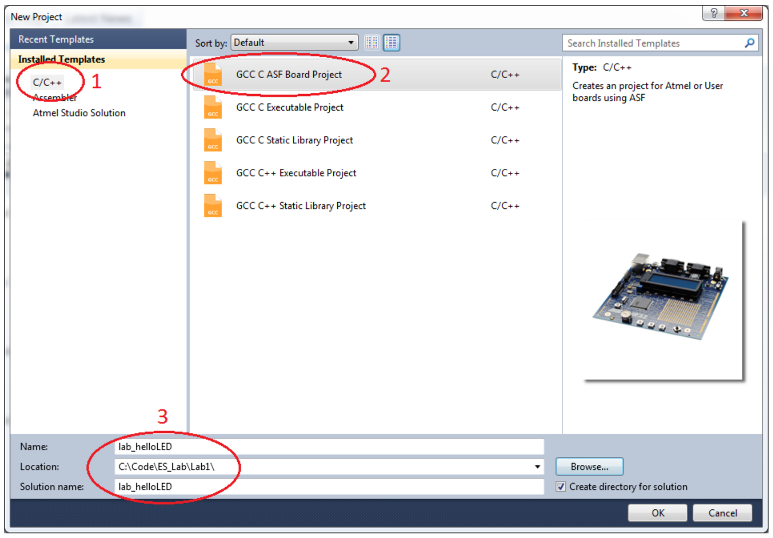


Figure 3: Creating a new project

- Select the C/C++ template. (1)
- Select Project type: GCC ASF Board Project. (2)
- Give the project a name and select a location. (3)
- Click Ok.



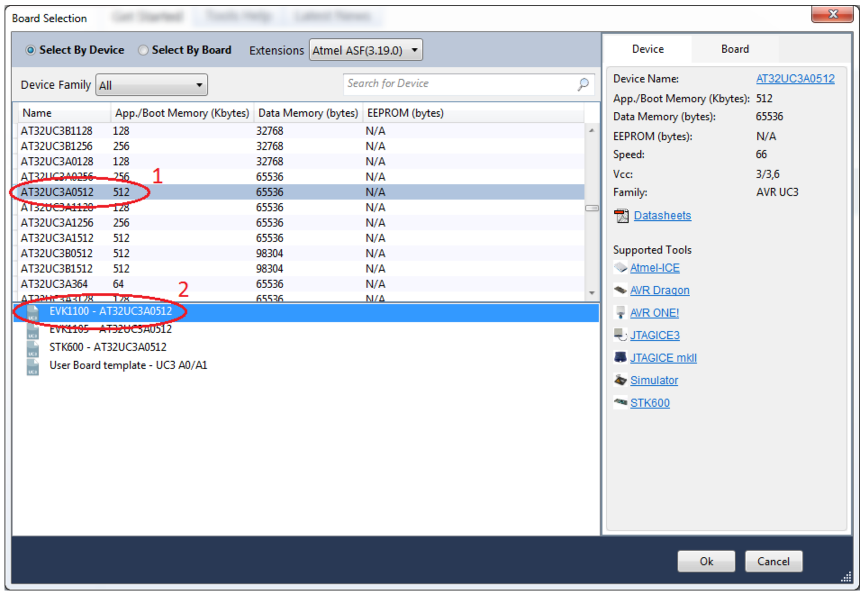
GCC C ASF Board Project	C/C++
GCC C Executable Project	C/C++
GCC C Static Library Project	C/C++
GCC C++ Executable Project	C/C++
GCC C++ Static Library Project	C/C++

Type: C/C++
Creates an project for Atmel or User boards using ASF

Name: lab_helloLED
Location: C:\Code\ES_Lab\Lab1\
Solution name: lab_helloLED
[✓] Create directory for solution

Figure 4: Creating a new project

- Select Target MCU: AT32UC3A0512. (1)
- Select EVK1100-AT32UC3A0512. (2)
- Click Ok to create the project.



Name	App./Boot Memory (Kbytes)	Data Memory (bytes)	EEPROM (bytes)
AT32UC3B1128	128	32768	N/A
AT32UC3B1256	256	32768	N/A
AT32UC3A0128	128	32768	N/A
AT32UC3A0256	256	65536	N/A
AT32UC3A0512	512	65536	N/A
AT32UC3A1128	128	65536	N/A
AT32UC3A1256	256	65536	N/A
AT32UC3A1512	512	65536	N/A
AT32UC3B0512	512	98304	N/A
AT32UC3B1512	512	98304	N/A
AT32UC3A364	64	65536	N/A
AT32UC3A3128	128	65536	N/A

EVK1100 - AT32UC3A0512
EVK1105 - AT32UC3A0512
STK600 - AT32UC3A0512
User Board template - UC3 A0/A1

Device Name:	AT32UC3A0512
App./Boot Memory (Kbytes):	512
Data Memory (bytes):	65536
EEPROM (bytes):	N/A
Speed:	66
Vcc:	3/3,6
Family:	AVR UC3

Supported Tools
- Atmel-ICE
- AVR Dragon
- AVR ONE!
- JTAGICE3
- JTAGICE mkII
- Simulator
- STK600

Figure 5: Create new project

In the Solution Explorer (usually left hand side of the window) you will find the new project.

- Double click the project name to open the file hierarchy.
- Double click the src folder.
- Double click the main.c source file.
- Erase the code skeleton of the main.c file and type (copy/paste) in the source code from the walk through above.

Before compiling we will make sure some of the project settings in Atmel Studio 6 are ok.

- Right click on the project and select "Properties"

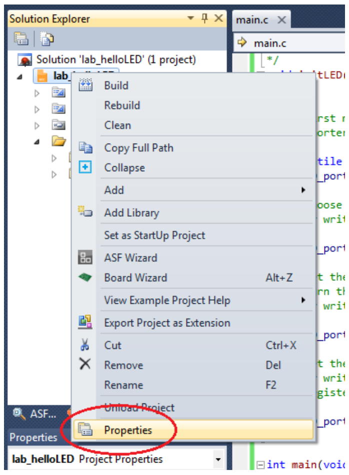


Figure 6: Open the project properties

- This will open the project settings in the main window.
- Select "Toolchain" on the left of the project settings.
- You can edit all compiler and linker options in this view.
- AVR32/GNU C Compiler: Make sure the Debug Level in the "Debugging" entry is set to maximum (-g3)
- AVR32/GNU C Linker: Make sure that, under the "General" entry, Not using standard start files (-nostartfiles) is selected.
- Now, click the "Device" tab to make sure the correct MCU is chosen, UC3A0512. There was a bug in AVR32 Studio 2.5 that made the project

forget which MCU to use, so it might have to be reset from time to time. You will notice this by strange compile errors that is not caused by your code.

- When done, save the file.
- Compile the project (in the top menu "Build/Build Solution" or press F7).

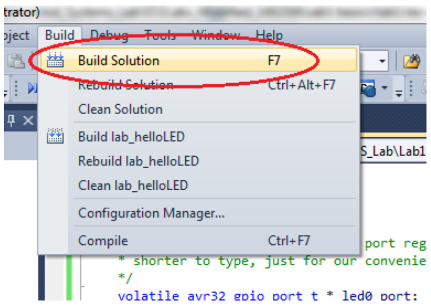


Figure 7: Build Solution

- If there were no compile errors you will find a .elf file, which is a generic binary image, in the Debug or Release folder of the project or respectively in the "Output Files" entry of the Solution Explorer.

Next we will program the MCU. See below.

3.4 Practise Assignment 4: Programming the Flash Memory

Content: Program the flash memory of the MCU with the "Hello LED" binary. We will verify that the JTAG debugger, The Dragon, is functional.

3.4.1 Programming the MCU

From the compile stage we have produced a .elf file that can be programmed into the flash memory of the MCU. In order to program the flash memory we need a JTAG programmer, which is a device able to communicate through the JTAG protocol (IEEE1149.1) with the chip on the dev. board.

First power everything off before connecting the JTAG device to the dev. board. Power up the JTAG device and last power up the dev. board.

This sequence is done to protect the JTAG device from harm. When the JTAG device is connected and everything is powered up, we can communicate with the dev. board MCU, also known as the "Target MCU", via a program called *avr32program*. AVR32 Studio integrates avr32program in the AVR32 Targets view.

1. Power everything off.
2. Connect the JTAG device to the dev. board.
   Important!
   Pin 1 on the JTAG device should match pin 1 on the dev. board JTAG connector. Use a 10-wire ribbon cable, put the red stripe to pin 1 at both the JTAG device and the dev. boards JTAG port.
3. Make sure The Dragon is correctly connected to the EVK1100 with the 10-pin ribbon cable. Make sure once again.
4. Power up the JTAG device.
5. Power up the dev. board.
6. Test the communication with the dragon from within Atmel Studio 6.
7. Open the "Device Programming" by either clicking "Tools->Device Programming" or by pressing Ctrl+Shift+P.

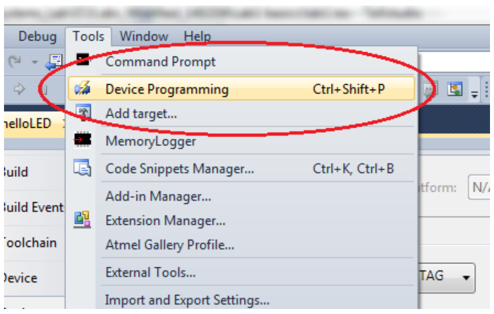
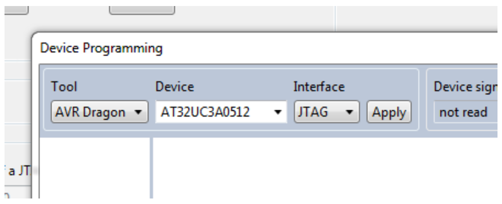


Figure 8: Open device programming

8. Make sure that you select the "AVR Dragon", the "AT32UC3A0512" MCU and "JTAG" as shown in the picture below. If so. click "Apply" to connect to the Dragon.



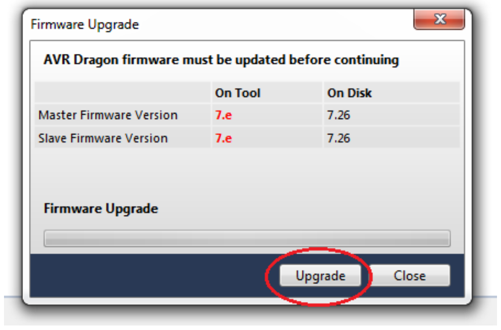
Device Programming

Tool	Device	Interface		Device sign
AVR Dragon	AT32UC3A0512	JTAG	Apply	not read

Figure 9: Setup the connection

9. Most likely you will need to upgrade the firmware on the Dragon. A popup window informs you about this, click "Upgrade".

10. This might take some time and a window will inform you that an operation is pending, you can wait or cancel. Click "Wait 1 more Minure" in order to allow the update to complete.
11. Click "Close" after the update finished.


Firmware Upgrade

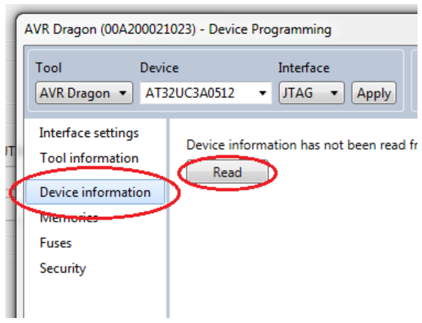
AVR Dragon firmware must be updated before continuing

	On Tool	On Disk
Master Firmware Version	7.e	7.26
Slave Firmware Version	7.e	7.26

Firmware Upgrade

Figure 10: Upgrade the firmware on the Dragon

12. You need to restart the Dragon. Unplug and plug the USB-cable. The red and greed LED should be turned on now.
13. In the "Device Programming" window click "Apply" to connect to the Dragon.
14. On the left menu select "Decive Information" and click "Read" to read information from the board.

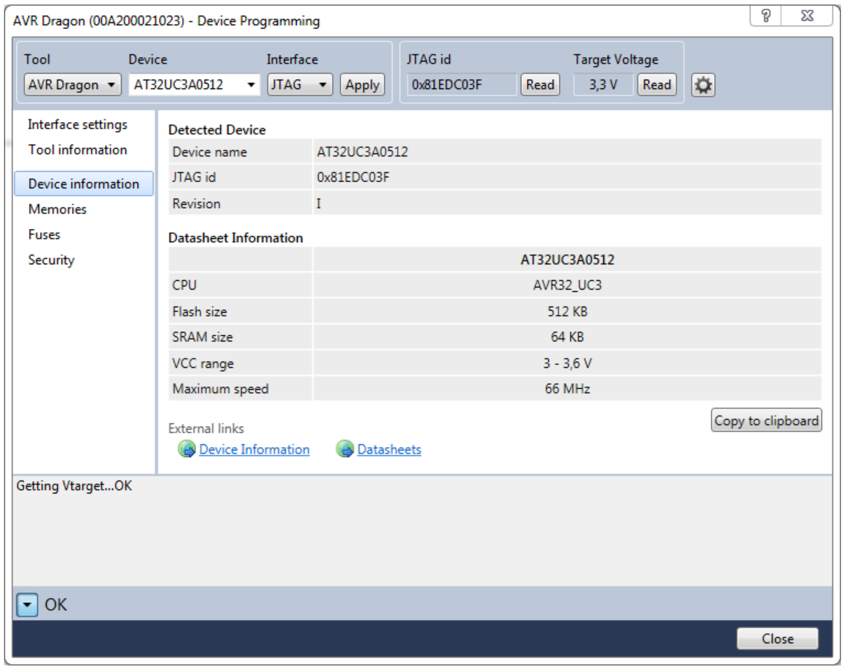

AVR Dragon (00A200021023) - Device Programming

Tool: AVR Dragon
Device: AT32UC3A0512
Interface: JTAG

Device information has not been read fr

Figure 11: Read the device information

15. The windows should look similar like the picture below with the "Device Name" AT32UC3A0512.


AVR Dragon (00A200021023) - Device Programming

Tool: AVR Dragon | Device: AT32UC3A0512 | Interface: JTAG | Apply | JTAG id: 0x81EDC03F | Read | Target Voltage: 3,3 V | Read

Detected Device

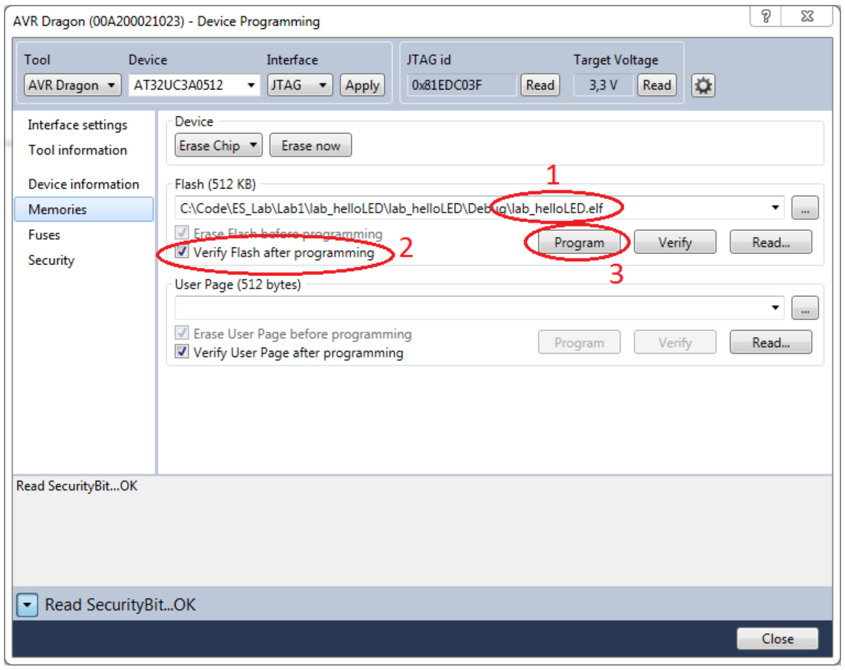
Device name	AT32UC3A0512
JTAG id	0x81EDC03F
Revision	I

Datasheet Information

	AT32UC3A0512
CPU	AVR32_UC3
Flash size	512 KB
SRAM size	64 KB
VCC range	3 - 3,6 V
Maximum speed	66 MHz

```
Getting Vtarget...OK
```

Figure 12: Device information



AVR Dragon (00A200021023) - Device Programming

Tool: AVR Dragon | Device: AT32UC3A0512 | Interface: JTAG | Apply | JTAG id: 0x81EDC03F | Read | Target Voltage: 3,3 V | Read

Device: Erase Chip | Erase now

Flash (512 KB): C:\Code\ES_Lab\Lab1\lab_helloLED\lab_helloLED\Debug\lab_helloLED.elf

☑ Erase Flash before programming
☑ Verify Flash after programming

User Page (512 bytes)

☑ Erase User Page before programming
☑ Verify User Page after programming

Read SecurityBit...OK

Read SecurityBit...OK

Figure 13: Device Programming - Memories

If the communication test was successful you are ready to program the flash. In AVR32 Studio, locate the JTAG device in the AVR32 targets window. Right click the JTAG device.

- Select Memories in the "Device Programming" window.
- Make sure the .elf file is selected, otherwise browse to the right location (The .elf file is located in the *Release* or *Debug* directory of your project).(1)
- It is useful to select the option "Verify Flash after programming". (2)
- Click on "Program" to write to the Flash. (3)
- Reset if necessary and run the program on the dev. board.

Does the dev. board respond according to your expectations?

If no error occurs during programming, you are ready to use the debugger to step through the program.

Chip Erase can also be done manually if necessary, by selecting the Memories option of the Device Programming window and then clicking "Erase now". This will remove any program already on the chip, setting it back to “normal”.

3.5 Practise Assignment 5: Debugging

Content: Use the JTAG Debugger to single step the "Hello LED" program.

3.5.1 Set up the debug device

We need to tell Atmel Studio 6 to use the Dragon for debugging. To do this we open the Project Properties and select the "Tool" tab on the left. In the main window we chose the AVR Dragon from the dropdown menu "Select debugger/programmer". Note, in oder to appear in the menu the Dragon needs to be connected to the PC.

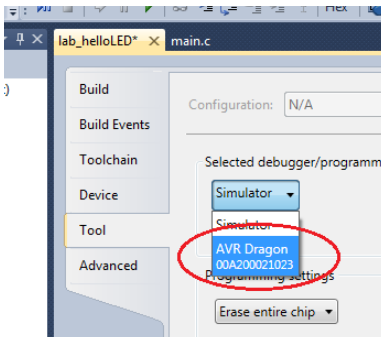


Figure 14: Select the debugger/programmer

3.5.2 Set a breakpoint:

Set a breakpoint at the start of the int main(void)-function. This is done by double clicking the grey column (left edge of the source code), or right click the source code line and use the menu that appears.

A breakpoint is indicated by a small circular symbol.

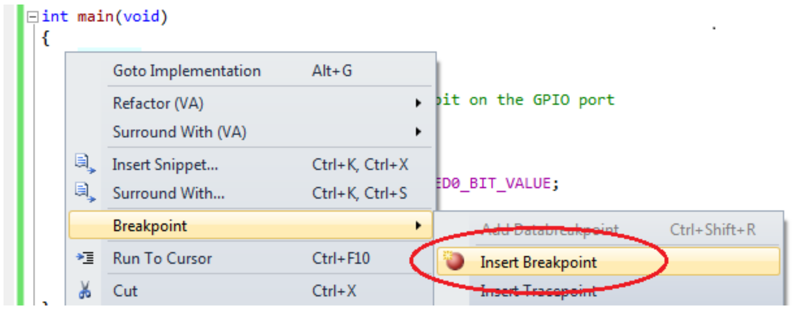

```
int main(void)
{
```

Figure 15: Breakpoint

3.5.3 Start a debugging session:

To start the debugging select "Debug -> Start Debugging and Break " or press Alt+F5.

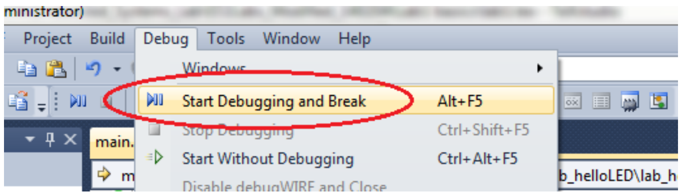
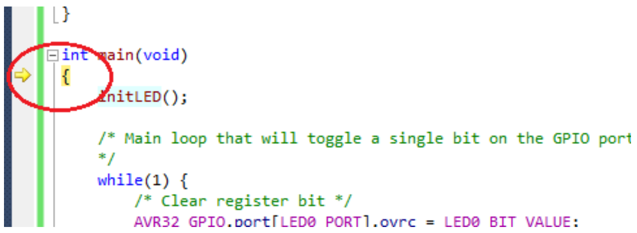

Project
Build
Debug
Tools
Window
Help
Windows
Start Debugging and Break Alt+F5
Stop Debugging Ctrl+Shift+F5
Start Without Debugging Ctrl+Alt+F5
Disable debugWIRE and Close
main.
b_helloLED\lab_h

Figure 16: Start debugging

Your code will be uploaded to the dev. board and the debugging session starts. Note that the view changes. You are now in debugging mode and other information is visible. This can take a short moment. You will see that the program execution stopped at the beginning of main (remember we selected "Debug -> Start Debugging and Break "). This is indicated with a yellow arrow. This error shows you which line is under execution.



```
}
int main(void)
{
    initLED();

    /* Main loop that will toggle a single bit on the GPIO port
    */
    while(1) {
        /* Clear register bit */
        AVR32_GPIO.port[LED0_PORT].ovrc = LED0_BIT_VALUE;
```

Figure 17: Debugging

You can use the options in the "Debug" menu of the toolbar or the shortcuts in the toolbar to control the debugging session. Investigate the result of every step taken through the source. Is the dev. board responding according to your expectations?

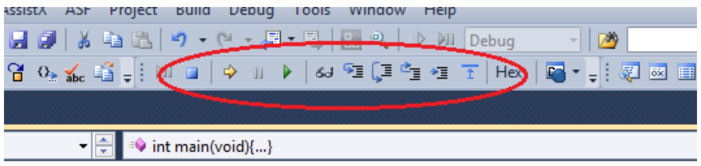


Figure 18: Debug control

You can terminate the debugging session by clicking on "Debug -> Stop Debugging".

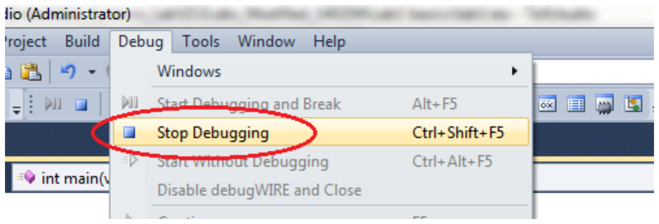

io (Administrator)
roject
Build
Debug
Tools
Window
Help
Windows
Start Debugging and Break Alt+F5
Stop Debugging Ctrl+Shift+F5
Start Without Debugging Ctrl+Alt+F5
Disable debugWIRE and Close
int main(v

Figure 19: Debug control

4 Button input

To read a GPIO pin it need to be set up as input. All GPIO pins are set as input as default when the board is powered up or reset in order to avoid unsafe operation of external units connected to the MCU. Except from enabling the GPIO peripheral unit with the gper register, setting a GPIO pin to input is done by clearing the corresponding bit in the oder register.

Example:

```
#define BUTTON_PORT (GPIO_PUSH_BUTTON_0 >> 5)
#define BUTTON_PIN (1 << (GPIO_PUSH_BUTTON_0 & 0x1f))
...
button_port->gpers = BUTTON_PIN;
button_port->oderc = BUTTON_PIN;
...
```

Reading the state of a button (i.e. GPIO pin) is done through the pvr register, Pin Value Register.

Example:

```
...
button_state = button_port->pvr & BUTTON_PIN;
...
```

Here we need to consider the electronic schematic of the EVK1100. The buttons are connected in a way that make the button_state != 0 if the button is not pressed (i.e. GPIO pin is connected to Vcc by not pressing the button). And button_state == 0 if the button is pressed (i.e. GPIO pin is connected to GND by pressing the button).

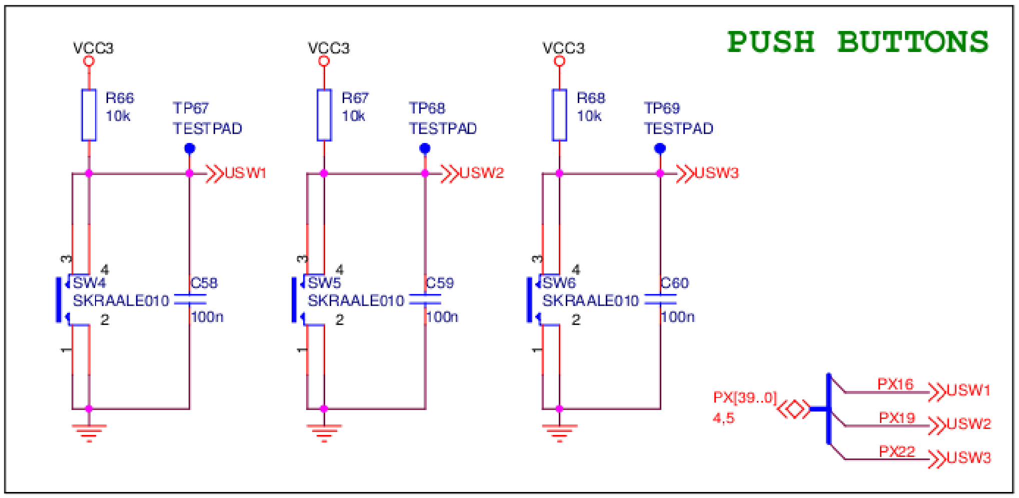


Figure 20: Push buttons schematics

5 Assignments

1. **Button input.** Write a program to start blinking LEDs as an action of button events. The LED should start blinking after you press the button.

2. Write a program that uses a button to light a LED. When the button is pressed the LED should be lit. When the button is released the LED should go off. Program another button to act as a on/off switch. Pressing and releasing the button once should light the LED. The next press/release cycle should turn the LED off.

Note: Comment your code well. Demonstrate all the labs and after approval email the code of all the labs to both lab assistants.

6 Report

The report should include the answers of the following questions:

1. How do you terminate the debugging session?

2. How do you terminate the program running on the dev. board?

3. **Bitwise operations**

   Content: Bitwise operations, &, ^, |, ~, <<, >> * Logical operations, &&, || * Relational operators ==, !=, <, >, <=, >=

   Source file: lab1/src/bitwiseoperations.c

In embedded systems it is often necessary to alter or read single bits in various registers. This task will train you in the bitwise operations available in the C language.

- Use the Atmel Studio 6 IDE.
- Create a new project.
- Open the source file, lab1/src/bitwiseoperations.c, and manually read through the source and note the value of 'result' at each time there's an assignment to the 'result' variable.
- Compile the program.
- Program the dev. board with the produced .elf file from the compile.
- Set a breakpoint at the beginning of the main()-function.
- Start a debugging session of the program.
- Step through the program and verify your predicted values of the 'result' variable.

Questions:

(a) What is the value of the 'result' variable at each occasion it is assigned a value? You must provide screen shots for every computed result.

(b) What happens at the end of the program?

7 Optional Assignment

1. Make a LED chaser program. The idea is to have the LEDs light up in sequence one after another to give the impression that the light is moving back and forth on the LED array.