Magnus Persson
Martin Grimheden
Per Mackegård
Jad El-Khoury

Power management lab

Version 2013-11-03

Aim

Familiarity with the clock settings and sleep modes of the AVR32, practical experience of power consumption of CMOS devices at different clock speeds, usage of the PM (power manager) and RTC (real time counter) on-board peripherals.

Literature

Data sheet "AT32UC3A" AVR32 32-Bit Microcontroller (the manual)

- Chapter 13. Power Manager (PM)
- Chapter 14. Real Time Counter (RTC)

Document "AVR32739: AVR32 UC3 Low power software design"
http://www.atmel.com/dyn/resources/prod_documents/doc32093.pdf

Header files:	`pm.h`, `rtc.h`, `power_clocks_lib.h`
C files:	`pm.c`, `rtc.c`, `power_clocks_lib.c`

Brief summary

This exercise gives a basic introduction to the power management features of the AVR32 chip, both in terms of setting different clock speeds for the CPU and using the different sleep modes. We also get an introduction to the real-time counter.

Further, practical measurements on the power consumption of the CPU and other devices of the EVK1100 board will be performed, to give you a gut feeling for what consumes power in an embedded device.

We will use a simple LED blinking program for demonstration purposes.

Reporting

Demonstrate the programs for the teaching assistants. Discuss the results of the power measurements with your teaching assistant – are the results as expected from theory? If not, discuss possible sources of the discrepancy.

0. Introduction and overview

In this lab you are expected to practically evaluate the theory of this week's lecture and also get acquainted with the power saving features of the AVR32 chip.

This exercise is based on the following setup and philosophy:

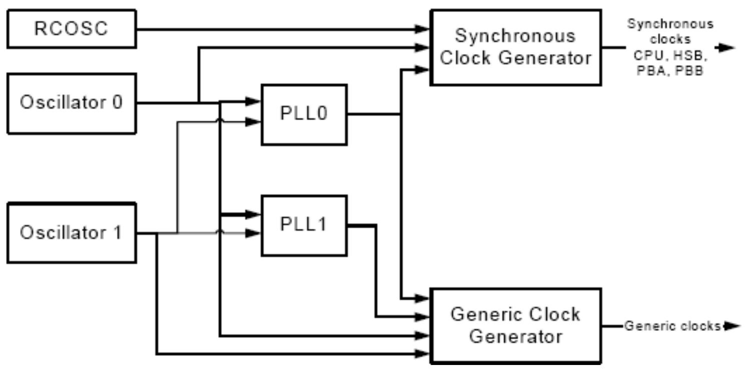
1) First, the clock management functions of the AVR32 will be introduced
2) Then you will perform practical measurements of a simple program with the processor running at different frequencies.
3) You will further set up the real-time counter to get a reliable time source
4) You will then use the sleep modes of the processor to reduce power usage, and compare it to previous power usage.
5) You will also practically measure and evaluate the power usage of some of the external and on-chip peripherals of the EVK1100 board that you have used in the course.
6) As the last part, you will implement a simulated pace-maker using the board.

1. Using the clock management functions

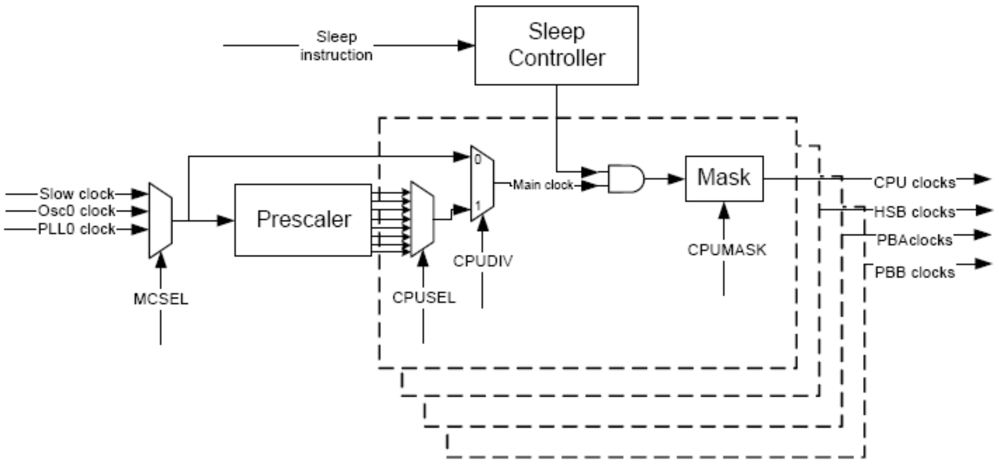
The clocks of the AVR32 can be derived from several sources, an internal high-precision 32 kHz source, an RC oscillator (low-precision) at approximately 115 kHz, and an off-chip crystal oscillator. The EVK1100 board is equipped with a 12 MHz oscillator.

The main purpose of the 32 kHz oscillator is to work as a high-precision time source for the RTC (real-time counter). The RC oscillator and external crystal oscillator(s) are used mainly as sources for the clocks for the CPU and the different on- and off-chip buses.

Within the AVR32 chip, the off-chip oscillator may be used to generate several different clocks through the use of PLLs (phase-locked loops, a device that can produce a high-frequency signal at an integer ratio of a low-frequency one) and dividers. There are two different PLLs: PLL0 and PLL1. Only PLL0 can be used for setting up a CPU clock, but both can be used to setup generic clocks for other purposes. The latter type of clocks will not be covered in this lab. The setup is shown in the following block diagrams:

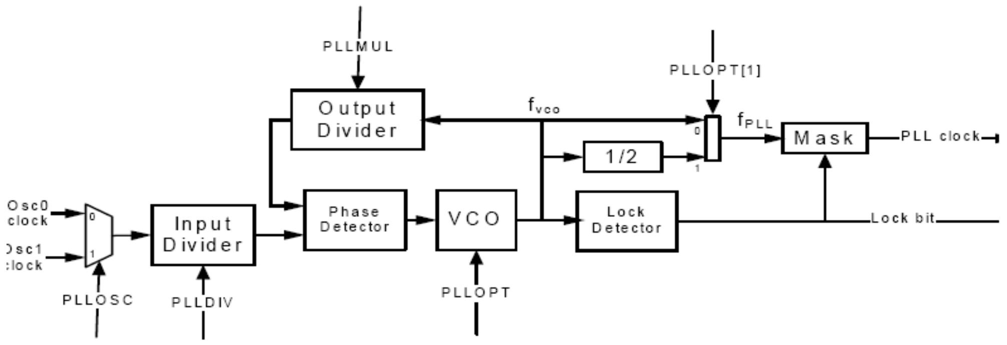


The CPU and the different buses on the chip may run at different clock speeds. By default, the CPU runs off the RC oscillator (~115kHz) and hence is quite slow. By switching the CPU to the external crystal oscillator instead, and using the PLL, a significantly higher clock speed can be reached, as you have already tried. The aim of this task is to help you understand how the clock speed is set.



Setting up the clock and PLL manually

There are several different clocks – the CPU and the buses can all be clocked separately depending on your needs. The normal way of adjusting the clock frequency is to adjust the PLL settings. This can be done either at startup or on-the-fly during program execution, giving the possibility of dynamic adjustment of the processor frequency based on workload.



The PLLs have the following main parameters:

- **PLLOSC**, which chooses if the PLL should run off of oscillator 0 or 1. In this lab exercise, we will stick to 0.
- **PLLMUL**, which gives the multiplier for the PLL. See the formula below!
- **PLLDIV**, which gives the divider for the PLL. See the formula below!
- **PLLOPT**, which gives the options for the PLL.

The frequency of the PLL oscillator is calculated as follows:

$$f_{VCO} = \frac{(PLLMUL+1)}{PLLDIV} \bullet f_{Osc} \text{ if } PLLDIV>0$$

$$f_{VCO} = 2 \bullet (PLLMUL+1) \bullet f_{Osc} \text{ if } PLLDIV=0$$

If PLLOPT[1] is set to one, $f_{PLL} = f_{VCO}/2$

If PLLOPT[1] is set to zero, $f_{PLL} = f_{VCO}$

The following functions from `pm.h` can be used to set the clock to an arbitrary clock speed (within the chip specifications):

```
        pm_switch_to_osc0();
        pm_pll_setup();
        pm_pll_set_option();
        pm_pll_enable()
        pm_wait_for_pll0_locked();
        pm_cksel();
```

We will use the example code previously used to setup the clock for 33 MHz operation as an example to explain the code executed to manually setup the clocks and PLL through the power manager module. Please refer to the `pm.h` file for further reference for the function calls.

```
pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);
```

The clock is switched to the external oscillator. The parameters give the address to the power manager, the clock frequency, and the startup time for the oscillator.

```
pm_pll_setup(&AVR32_PM, 0,  // pll.
                     10,  // mul.
                     1,   // div.
                     0,   // osc.
                     16); // lockcount.
```

PLL0 is setup to use the multiplier 10, the divider 1, is to use oscillator 0, and the lockcount 16. This results in the following VCO clock frequency:

$$f_{VCO} = \frac{(PLLMUL + 1)}{PLLDIV} \bullet f_{Osc} = \frac{10+1}{1} \bullet 12\,\text{MHz} = 132\,\text{MHz}$$

```
pm_pll_set_option(&AVR32_PM, 0, // pll.
                          1,  // pll_freq.
                          1,  // pll_div2.
                          0); // pll_wbwdisable.
```

The options for PLL0 are set. The frequency range is 80-180 MHz, i.e. `pll_freq` is set to 1 instead of 0 (corresponds to 160-240 MHz), the frequency is divided by 2 after the PLL (`pll_div2` is set to 1), and the special wide bandwidth mode (allowing a faster startup) is not used.

This gives $f_{PLL} = f_{VCO}/2 = 66$ MHz.

```
pm_pll_enable(&AVR32_PM, 0);
pm_wait_for_pll0_locked(&AVR32_PM);
```

Enable PLL0 and wait for it to properly lock.

```
pm_cksel(&AVR32_PM,
                  1,   // pbadiv.
                  0,   // pbasel.
                  1,   // pbbdiv.
                  0,   // pbbsel.
```

```
                      1,   // hsbdiv.
                      0);  // hsbsel.
```

Choose the new clocks for the buses. If the relevant DIV setting is set, the clock will be subdivided. For example, if CPUDIV is set to 1, the CPU speed will be

$$f_{CPU} = \frac{f_{main}}{2^{(CPUSEL+1)}}$$

The `pm_cksel` function always sets the CPU speed to the same clock speed as the high speed bus (CPUSEL = HSBSEL and CPUDIV = HSBDIV). In this call, this means that the CPU speed will be

$$f_{CPU} = \frac{66\,\text{MHz}}{2^{(CPUSEL+1)}} = \frac{66\,\text{MHz}}{2^{(0+1)}} = 33\,\text{MHz}$$

If you want to run the CPU (and high speed bus) at 66 MHz instead, you may change the parameter `hsbdiv` to “0” instead, to use an undivided clock for the CPU and high speed bus. If you have low requirements on bus bandwidth, you may also choose to use different dividers for the buses. At this point, the setup of the new PLL is complete.[1]

Please note that the maximum clock frequency for the CPU is 66 MHz, for bus A is 33 MHz, and that the bus frequencies must not be larger than the CPU frequency.

```
pm_switch_to_clock(&AVR32_PM, AVR32_PM_MCCTRL_MCSEL_PLL0);
```

This line of code makes the power manager switch to using the new clock.

Task

a. Create a new project, include the software framework drivers GPIO, INTC, PM and RTC which are the ones that will be used in this exercise.
b. Build a simple LED blinking application which toggles a LED at approximately 1 Hz, by using a software delay through e.g. the following function:
```
static void software_delay(void)
{
  volatile int i;
  for (i=0; i<1000000; i++);
}
```
Make sure that the CPU is running at 12 MHz by including the following line early in your main function:
```
pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);
```
c. Fill out the table on the following page with clock settings for the different clock speeds. Start by choosing a suitable frequency to be output from the PLL. Please note that the PLL does not work with all settings – if the ones you have chosen do not work, try alternative ones! Specifically, the PLL will fail if you run it outside of its specifications (e.g. allowed frequency range).

[1] If you later intend to use the flash controller using a higher frequency than 30 MHz, you will also need to insert an additional wait state for the flash controller by executing the following code before switching to a new clock with a higher clock frequency than 30 MHz:
```
flashc_set_wait_state(1);
```

d. Test all your clock setting combinations, measure (e.g. with your wristwatch) or simply estimate the blinking frequency, and measure the power consumption using an amperemeter. In order to connect it, you need to supply power through the external jack instead of through USB (switch in position "EXT"). There are custom-made cables that you may use– its red cable needs to be connected to +5V and the black one to ground.

	Clock speed [MHz]				
Clock settings	**2**	**12**	**33**	**48**	**66**
PLL frequency chosen					
PLLMUL					
PLLDIV					
PLL_FREQ					
PLL_DIV2					
PBADIV					
PBASEL					
PBBDIV					
PBBSEL					
HSBDIV					
HSBSEL					
Works as expected?					
Approximate blinking freq.					
Measured power consumption					

Hint: to simplify measurement, you may use the pushbuttons on the EVK board to dynamically switch clock frequency at runtime.

Report
Demonstrate that you got the different clock frequencies running properly to one of the lab assistants.
Add your measurements to the brief report. Discuss your findings shortly (a few sentences only!)

2. *Setting up the clock and PLL using the support functions from power_clocks_lib.h*

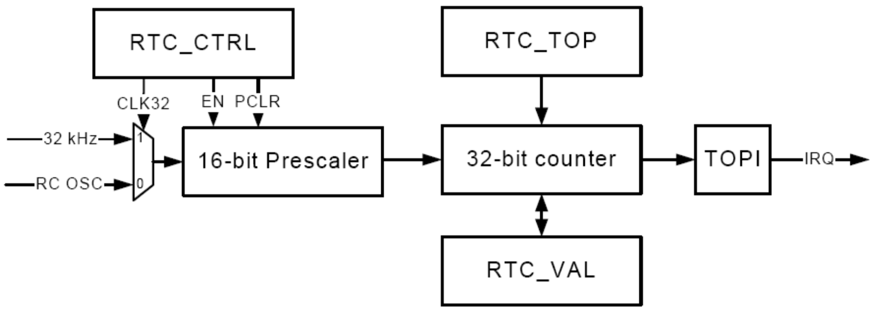
Setting up the clock and the PLL manually, as you just learnt, is quite cumbersome. Hence, the power manager comes with a dedicated support library that helps you with the calculations.

Task

a. Read the content of `power_clocks_lib.h`. Specifically check out the function `pcl_configure_clocks`. Which function can you use if you want to run from the RC (115 kHz) oscillator?
b. The function to be called to run the clock from the external oscillator needs a struct as input. What is the needed content?
c. Try running your program in different clock frequencies using the support library instead of directly (manually) adjusting the clock and PLL settings.

3. *Using the real-time counter*

The real-time counter is relatively simple. It can be run off of two different clock sources, either the high-precision 32 kHz clock source or the low-precision RC oscillator with an approximate frequency of 115 kHz. In this lab, we will keep to using the 32 kHz source. The counter uses two main parameters: the prescaler and the top value.



The prescaler can be used to set the base frequency for the RTC. The base frequency is given by the following formula:
$f_{RTC} = 2^{-(PSEL+1)} \cdot (f_{RC} \text{ or } 32 \text{ KHz})$
For every tick of the base frequency, RTC_VAL is increased by 1. When the value RTC_TOP+1 has been reached, the counter wraps over and an interrupt is given if the configuration has been done correctly. This means that an interrupt will be given with every clock tick if RTC_TOP is set to 0, every second time if it is set to 1, and so on.

Task

d. Read the description of the RTC in the datasheet, and the support functions in rtc.h. There is a predefined value which gives a 1 Hz counter with the 32 kHz clock input. Which?
e. If you needed an interrupt every 2½ seconds, how would you change the setup of the RTC?
f. Use the counter to rebuild your LED blinking application, toggling the LED every 1 s based on an interrupt from the RTC.

   Hints:
   - It is a good idea to use the interrupts from the RTC. To get the interrupts to work, you need to include the following code into your main function (the following code assumes that your RTC interrupt handler is named `rtc_irq`):
     ```
     Disable_global_interrupt();
     #if __GNUC__
       // Initialize interrupt vectors.
       INTC_init_interrupts();
       // Register the RTC interrupt handler
       //to the interrupt controller.
       INTC_register_interrupt(&rtc_irq, AVR32_RTC_IRQ, AVR32_INTC_INT0);
     #endif
     Enable_global_interrupt();
     ```
   - In your interrupt service routine, of course you also need to clear the interrupt. This can be done by the following code:
     ```
     rtc_clear_interrupt(&AVR32_RTC);
     ```
   - Further, you need to setup the RTC properly. That is done by successive calls (while interrupts are disabled) to the following functions:
     ```
     rtc_init
     rtc_set_top_value
     rtc_enable_interrupt
     rtc_enable
     ```
   - There is also an example project for the RTC driver, building a timer clock which you may use as inspiration and help.

g. Try using different clock frequencies of the CPU. Does the behavior of the LED change? Is the power consumption different compared to earlier? What is the advantages of using the RTC instead of a software idle loop?

4. Using the sleep modes of the CPU

Finally, you are supposed to use the different sleep modes of the processor. There are several different sleep modes in the processor, gradually shutting of additional parts of the microcontroller and being hard to wake up from. Waking up from a sleep mode is done automatically when an interrupt occurs – either from an internal source or an external one. Hence, it is a good idea to build an event-driven system when trying to build a power-scarce solution with an AVR32 processor.

The sleep modes range from *idle* (where only the CPU is shut down) to *static* (where essentially everything is shut down and an external interrupt is needed to start the microcontroller again).

In your current project, we already use an interrupt to trigger the change of status. Hence, it is simple to put the CPU into the sleep mode when nothing needs to be done (if no interrupts had been used, the CPU would simply have gone into sleep mode and never woken up again!)

Task

a. Make your program use a sleep mode instead of the software delay if one of the buttons is pressed. Going to sleep can be done by using the following code in your main loop. Confirm that the code still executes in the same manner:
```
SLEEP(AVR32_PM_SMODE_IDLE);
```
b. Measure the power consumption of the board. There should be a significant difference! How big is the difference at different clock speeds, e.g. 12 MHz vs. 33 MHz or 66 MHz?
c. Try also the other sleep modes. With which sleep modes does the blinking application still work? With which ones does it fail to restart? Explain why it doesn’t keep working with some sleep modes.
d. Measure the power consumption when running with a clock frequency of 66 MHz, using each sleep mode and fill in the table.

Sleep modes	Blinking?	Power consumption
Idle		
Frozen		
Standby		
Stop		
Deep stop		
Static		

Report

Add your measurements to the brief report. Discuss your findings shortly (a few sentences only!)

Upload the C code to KTH Social.

5. *Exploring the power consumption of the other peripheral devices of the AVR32 and the EVK1100 board.*

Task

a. Explore the other devices of the AVR32 and the EVK1100 board. How much power do the USARTs use? The LEDs? The display? Is there any device that is more of a power hog than the others?

Report

Add your measurements to the brief report and discuss your findings in text (a few sentences only)

Hardware	Power consumption
USART	
All LEDs	
LCD	