Laboratory Practical 4:
AT91RM9200 Programmable Inputs and Outputs (PIO)

TEAM:	Two
DURATION:	3 hours
DOC REF:	PRAC04 revision 2

Contents

1. Introduction....1
  1.1. Rules....2
  1.2. Code Handin....2
2. Activities....2
  2.1. Collect a PASS....2
  2.2. Log in and obtain Prac04.tar.gz....2
  2.3. Connecting up to the CSB337....2
  2.4. The AT91RM9200 PIO Controller....3
  2.5. Turning LEDs On and Off TODO[3]....6
  2.6. Pushbuttons TODO[4]....9
3. Finalization....9

1. Introduction

This practicals builds on methods covered in Prac03, and involves using the programmable input/output port of the AT91RM9200 microcontroller. This practical involves programming the parallel input and output lines using hardware registers in C. For this practical, you are recommended to make use of the AT91RM9200.h and AT91RM9200_inc.h and lib_AT91RM9200.h (see Section 2.4 for details on these files).

This practical focuses just on getting the user LEDs and pushbuttons working – but getting these to work requires an understanding of other devices built into the microcontroller. The PIO (Parallel or Programmable Input Output) Controller is one such device. The devices on the AT91RM9200 microcontroller that we will use in this practical include:

1. The PIO control peripheral (of which there are four in the microcontroller)
2. The Power Management Controller (PMC) which controls clock signals to devices.
3. Other devices used automatically: ARM920T core and the memory control unit.

BE WARNED and BE PREPARED: This Prac involves a lot of reading... that is something that comes with the territory of understand this microcontroller. Reading the 650 page datasheet could be worse.

IMPORTANT: HAND IN project framework[1] for this PRAC – see Section 3

1 Note that by project framework, I mean the entire ESAOA *Project* that you modified for this practical. Tar and gzip the entire project directory (i.e. include all the files, not only the files you changed).

1.1. Rules

The rules for this practical are as follows:

- Arrive at the lab at any time during the assigned practical times. If you have booked a seat, you have preference to getting a CSB337 and workstation. You are *encouraged* to arrive when the session starts so that you will have enough time to complete the practical assignment. You can also work on this prac in your own time outside of the assigned lab sessions, however help during such times is not guaranteed.
- Collect a PASS (Practical Assignment Solution Sheet) from the tutor before starting the prac.
- You may be asked from time to time to demonstrate or explain aspects of what you have done, to ensure that both partners in groups of two are involved, and working on, the assignment. If you work in the assigned lab slot, this may be done during that slot; otherwise you may be called on at a later stage to demonstrate aspects of the prac if the lecturer requires it.
- You are encouraged to work in teams of two so that you and your partner work together. If you and your partner do not both start on the practical at the same time, seek approval of the lecturer/tutor beforehand.
- Use the class notes and textbooks if you want to. You can also make use of the web; but if you do so, you are expected to cite references for material used (except for material provided on connect / the textbook). Such citations should be in the code, report, or PASS (or duplicated in multiple of these submission media) whichever is most appropriate.
- Ask questions if you get stuck
- Hand in in your PASS at the end of the session
- The PASS answers, code solutions, reports and any other material you submit for this practical assignment must be your own work, i.e. copying of other students code or answers is not permitted.

1.2. Code Handin

Tar and gzip your entire Prac04 directory (you can do a "*m clean*" before doing so to save space). Please hand in the archive using the connect website. **NB:** code can be handed in one week after laboratory session B.

2. Activities

This section explains what you need to do in order to complete this prac. Main work starts in **Section 2.4**.

2.1. Collect a PASS

Have you got a Practical Solution Sheet, and have you and your team mate put your student numbers on it? Have you verified that you have all the equipment need, i.e. CSB337 with necessary connectors.

2.2. Log in and obtain Prac04.tar.gz

Follow the usual procedure to log in to our Linux server *forge.ee.uct.ac.za* and start up the WinAxe X-server if you want to use it. You may want to try using kdevelop3 to make things easier. You can uncompress the archive directly into your ~/aoa/Projects directory as follows:

```
$ enter-aoa
$ cd Projects
$ tar -zxf  /EEE374W/Pracs/Prac04/Prac04.tar.gz
```

2.3. Connecting up to the CSB337

If you've forgotten how to connect up the board, look back at Prac02. You need to create your local subnet, which basically means you need to check that MicroMonitor on the CSB337 has been configured to IP address 192.168.0.2, and that you can ping that address from the command prompt.

CHECK THAT YOUR LOCAL SUBNET IS WORKING BEFORE CONTINUING !!!

TIME ESTIMATE: 10 minutes

2.4. The AT91RM9200 PIO Controller

Control of the LEDs and pushbuttons involves using **Programmable Input Output** (PIO) controllers (also celled *Parallel* Input Output Controllers in the ATMEL documentation). Four PIO controllers are built into the AT91RM9200 microcontroller, and are referred to as: PIOA, PIOB, PIOC, and PIOD.

Appendix A provides an excerpt of the block diagram provided in ATMEL's summarized datasheet for the AT91RM9200 (see document *at91rm9200-datasheet-summary-doc1768s.pdf,* which is stored on forge in folder /EEE3074W/Documentation/Hardware/Datasheets/AT91RM9200)[2]. The diagram in appendix A is annotated using circles to indicate devices used in this prac and how they are connected to the core. Lecture 13 (from slide 20 onwards) provides a brief look at using PIO.

2.4.1 GPIO and Multiplexed Peripheral I/O Lines

Two types of PIO pins are used to connect control lines to the PIO controller, these pins are referred to as **PIO pads**. Each PIO controller on the AT91 has 32 PIO pads. Each PIO pad is configured as either:

- A General-Purpose I/O (GPIO) line only: in this state the pad is *not* connected to one of the embedded peripherals shown on the left of **Illustration 1**, but connects either to a leg of the microprocessor (in order to connect external peripherals) or to some control line in the chip.
- An I/O line multiplexed with one or two peripheral I/Os: in this state, the pad connects to a control line of one of the two embedded peripherals shown on the left of **Illustration 1**.

A General Purpose I/O (or GPIO) line is a PIO bit that the product manufacturer (i.e. ATMEL in our case) decided to leave disconnected to on-chip peripherals so that external hardware components (such as a LCD screen or parallel port printer) can be connected. Note that all these bits are *not necessarily* routed to physical legs on the chip package but may be used for other internal purposes.

Each PIO pad is configurable by the hardware developer according to product needs. ARM designers made the initial choices in their reusable ARM9 core design. Then chip designers at ATMEL made further choices when designing the AT91RM9200 microcontroller by building on the the ARM9 design. Designers at Cogent Computers then made further choices, but only in terms of adding external peripherals, to the AT91RM9200 on the CSB337 evaluation board, which involved routing PIO pads connected to legs of the microcontroller package to other peripherals.

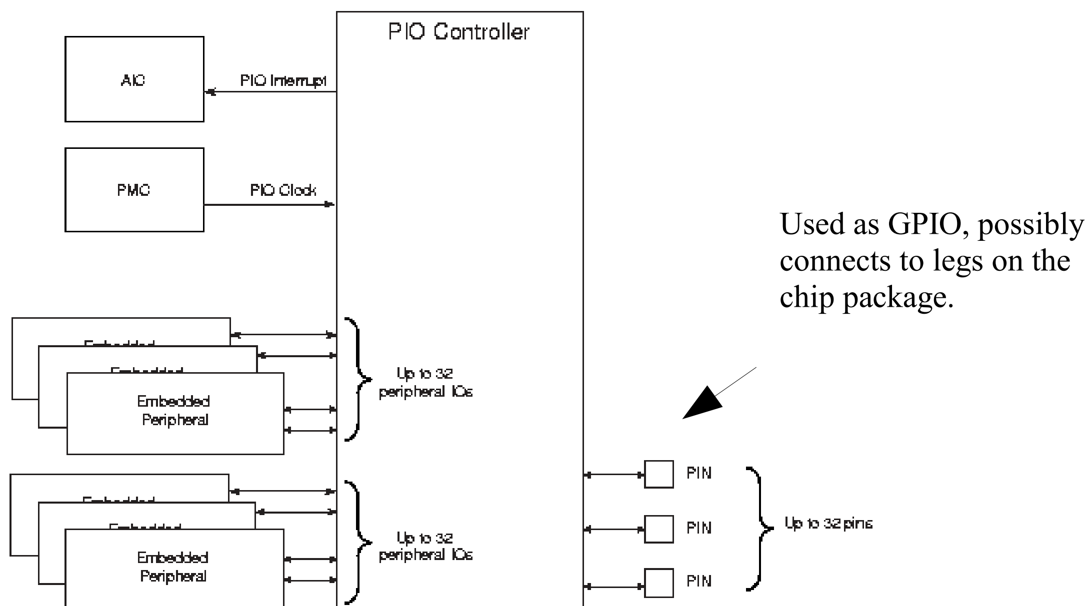


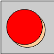
Illustration 1: Overview of PIO Controller (pg 334 of AT92RM9200 datasheet).

2 If you are logged in to forge.ee and have the Xserver enabled, are in the ESAOA environment, in directory Proc04, then you can enter in the command **ds 3** to bring up the summarized datasheet without tying the path names. Use **ds** without parameters to show the listing of relevant datasheets.

It is the responsibility of the *software developer* to *tell* the PIO controller when to access an embedded peripheral via a PIO pad, or to set the pad in GPIO mode. It is the responsibility of the *hardware developer* to make sure the necessary physical links are in in place either internally in the chip, or externally via tracks on the printed circuit board or using wires. When a certain PIO pad is in GPIO mode, it is not connected to any of the embedded peripherals shown on the left of **Illustration 1**, but is rather controlled by the PIO controller via the microcontroller core and used as a general purpose output or input pin.

The components that are swapped using the multiplexing of a PIO pad, when in *multiplexed peripheral* I/O mode, is hardware defined (i.e. decided by the hardware manufacturer). The *PSR* (or Peripheral Select Register) needs to be assigned appropriately in software to use one of the two multiplexed states – which is under control of the software developer. For example, ATMEL hardware designers decided pin 5 of PIOA (abbreviated to PA5 in the documentation) will be used to control either the UART3 transmit line, or to perform an SPI chip select, both of which are on-chip peripherals[3]. In this example, it is the software developer choice when to use UART3, or to use SPI chip select, in the code.

>>> NB : QUESTION TO TEST YOUR KNOWLEDGE <<<



Q-1: Briefly describe your understanding of a **PIO pad** based on section 2.4.1. Provide a rough drawing help your explanation. For your diagram, do not simply reproduce Illustration 1, but provide a simply view (e.g. just a few PIO pads) in their various states.

3 (both of these peripherals are most likely licensed Intellectual Property obtained in VHDL format and integrated into the AT91RM9200 VHDL code)

2.4.2 PIO Control Logic

The annotated illustrations below express the working of the PIO on output (Illustration 2) and input (Illustration 3). The output and input descriptions are separated as showing them together makes the diagram somewhat more complicated. Besides, one can only use a particular bit as input or output at a certain time. For more detail on PIO operation, see pages 333 to 358 in the AT91RM9200 datasheet. See Appendix B for a description of the diagrammatic notation used in the illustrations below.

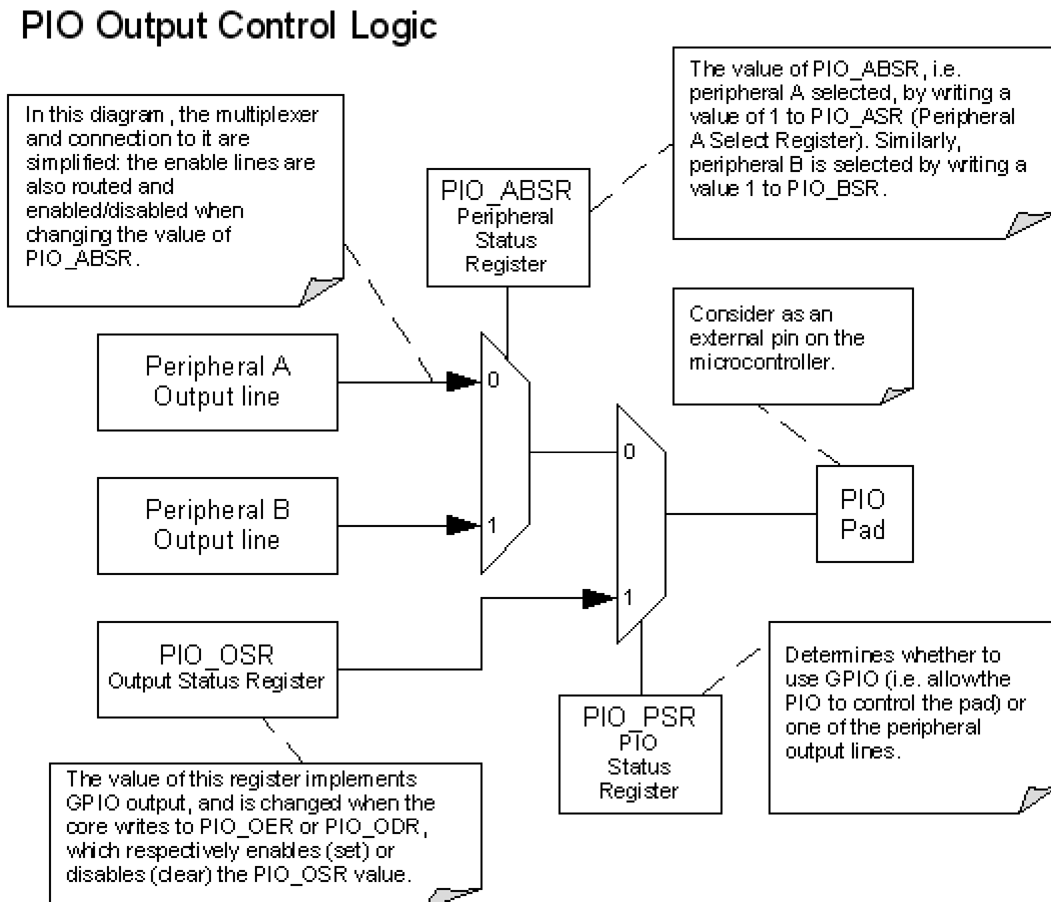


Illustration 2: Output control logic for the PIO controller.

As discussed in section 2.4.1, the bit called the "PIO pad" is a physical control line inside the chip, which the chip manufacturer decides where to connect. Writing a 1 to PIO_PER (PIO Enable Register) enables GPIO, writing a 1 to PIO_PDR (PIO Disable Register) sets multiplexed peripheral mode.

When in GPIO mode (i.e. PIO_PER is set), write a 1 to PIO_OER (Output Enable Register) to pull the pad high, or write a value 1 to PIO_ODR (Output Disable Register) to pull the pad low.

When in multiplexed peripheral mode (i.e. PIO_PER is cleared), the pad is controlled by either peripheral A or peripheral B. Giving control of the pad to peripheral A is done by writing a 1 to PIO_ASR (A Select Register); while giving control of the pad to peripheral B is done by writing a 1 to PIO_BSR. An example of why you might want to multiplex a peripheral is to switch from using the on-chip UART to a different on-chip peripherals when communication over RS232 is not needed.

As you can see, the PIO controller adds a significant amount of versatility to a microcontroller, allowing the on-chip peripherals to be used at different times, using the same set of pads. This illustrates a form of reconfigurable hardware at the level of SoC periperals.

PIO Input Control Logic

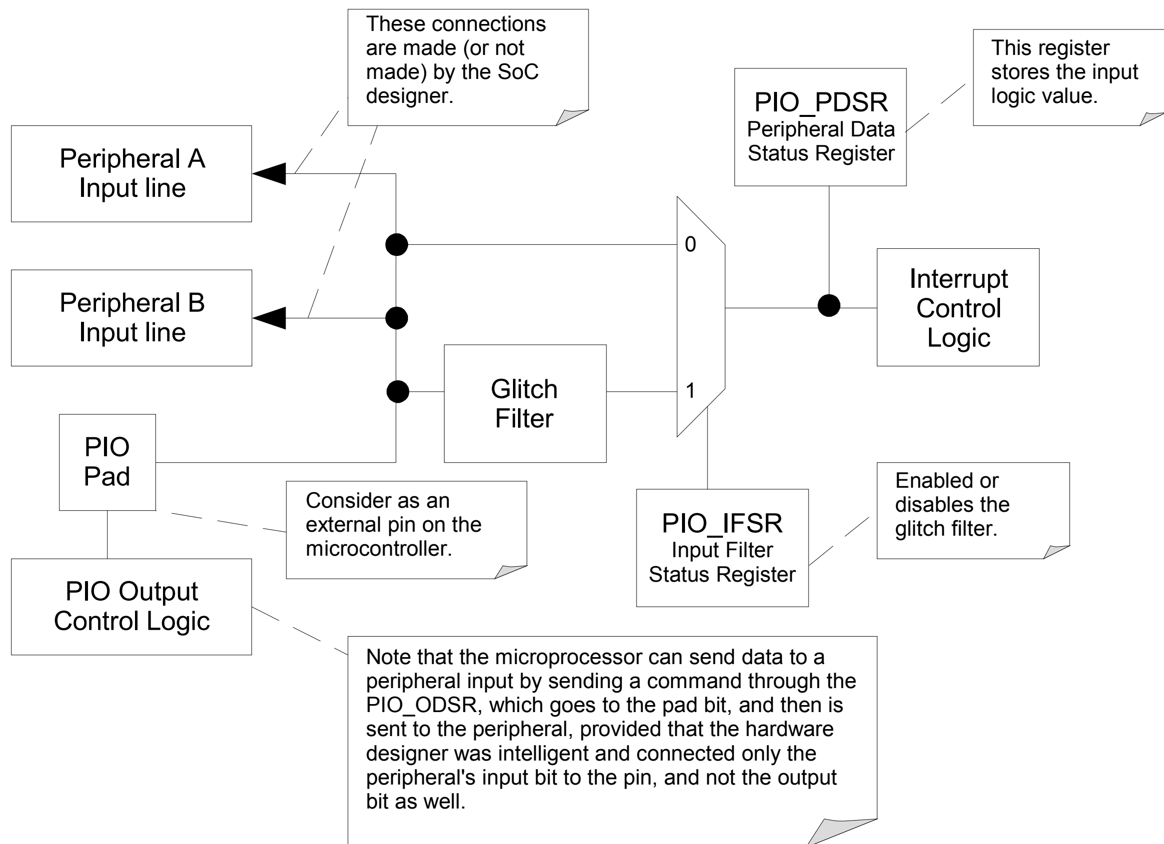


Illustration 3: Input control logic for the PIO controller.

There are 128 PIO pads available on the AT91RM9200, and most are used to control on-chip, or on-board peripherals. Of the GPIO lines that ATMEL left available, and routed to the legs of the chip, Cogent leaves only a few of the lines are available to us for use as GPIO for controlling external hardware that we may want to connect up to the CSB337 without using one of the provided standard interfaces (i.e. not using SPI, I2C, RS232, Ethernet). These lines are available on header P4 (which we will make use of in a later practical).

It is important that the software developers are advised of the hardware developer's choice as to which PIO pins are configured as GPIO, and of the remaining pins, which peripheral connects to the pin in a certain multiplex state. Fortunately, Cogent Computers provided the **CSB337 user manual** which summarizes this information in a series of tables for each PIO device. From the software side, and for using the CSB337 in a certain application, we still need to decide which on-board peripherals to use, and when to use them, and then to figure out how to configure the PIO controllers to use them. And that's the focus for the rest of this prac.

>>>>> Panic Questions <<<<<

Using the PIO controller is no simple undertaking. We need to remember that we are dealing with a highly sophisticated microcontroller which not only executes instructions quickly, but can reconfigure its own internal connections, and share input/output pins, in elaborate ways. Effective use of the CSB337 depends on you understanding the PIO controller; therefore I have some questions to test your knowledge on this:

PANIC **Q-2:** How many GPIO pins can be linked to one PIO pad? (a) None, (b) 1 , (c) 2 , (d) 3

PANIC **Q-3:** Who chooses *when* an embedded peripheral I/O line linked to the PIO controller is connected to the pad and thus made accessible to the ARM920T core and external devices if the pad is connected to a leg of the chip package? *Choices:* (a) The hardware developer, or (b) The software developer

CHECKLIST for 2.4: Q-1, Q-2, Q-3 *TIME ESTIMATE: 30 minutes*

2.5. Turning LEDs On and Off TODO[3]

The LEDs are on **PIOB** (see PIOB in the CSB337 user manual). As discussed in the previous section, PIO pads may either be configured as General Purpose I/O (GPIO), or as on-chip peripheral control lines multiplexed between two peripherals. Therefore, in our startup code (i.e. the *start()* function of our C programs), we need to execute code to configure the PIO controller so that the pads used to control the LEDs are configured as GPIO (STEP 2 below involves performing doing this task, but before making the changes, you need to do STEP1 to know what PIVs are).

> For this practical, edit files in directory Prac04/Software/PDM/CSB337/MicroMonitor, so from the root of Prac04, execute the command:
>
> **cd Software/PDM/CSB337/MicroMonitor**

STEP 1: Understanding Platform Integration Values

The ESAOA framework employs the strategy of separating platform deployment modules (PDM) from the application modules. The PDM modules contain *platform-dependent* code which performs low-level operations (e.g. init.c and asm_meths.S) and are stored under directory $H/Software/PDM). The application modules are stored in application folders in directory $H/Software/Applications. Application modules make calls to functions in the types of modules described in **Table 1**.

Table 1: Functions callable from application modules

Module Type	***Location***
Application modules: an application can make calls to function in the same or other application module.	Software/Applications
Utility modules: modules that store functions commonly used by different applications (e.g. string handling, FFT routines).	Software/Utils
CCW (Common Component Wrapper) modules: these modules provide an interface to lower-level modules which implement hardware access routines.	H/interface: Software/CCW C/implementation: Software/PDM
Standard libraries: If the application is designed to use standard libraries (such as stdlib.h, math.h, etc) then these includes can be used. However, if there are no guarantees that these libraries are to be available, then CCW interfaces should be used instead.	Standard installation path, or Software/Libs

A form of association is needed between the CCW function interfaces, and their implementations in the PDM directory. For example, a function to turn on one of three a LEDs is likely to take an integer parameter that indicates which of 3 LEDs to turn on. This association is done using *Platform Integration Values* (or PIVs).

Let's use an example to illustrate PIVs. Consider that a CCW function interface needs to be implemented for two different platforms, platforms A and B (see Illustration 4). The microprocessors in both platforms are connected to a comms status LED (an orange LED), a slave/master slider switch, and to a shared bus via a One Wire Protocol (or OWP) circuit. The OWP needs two lines for full-duplex communication, one for reading, and one for writing.

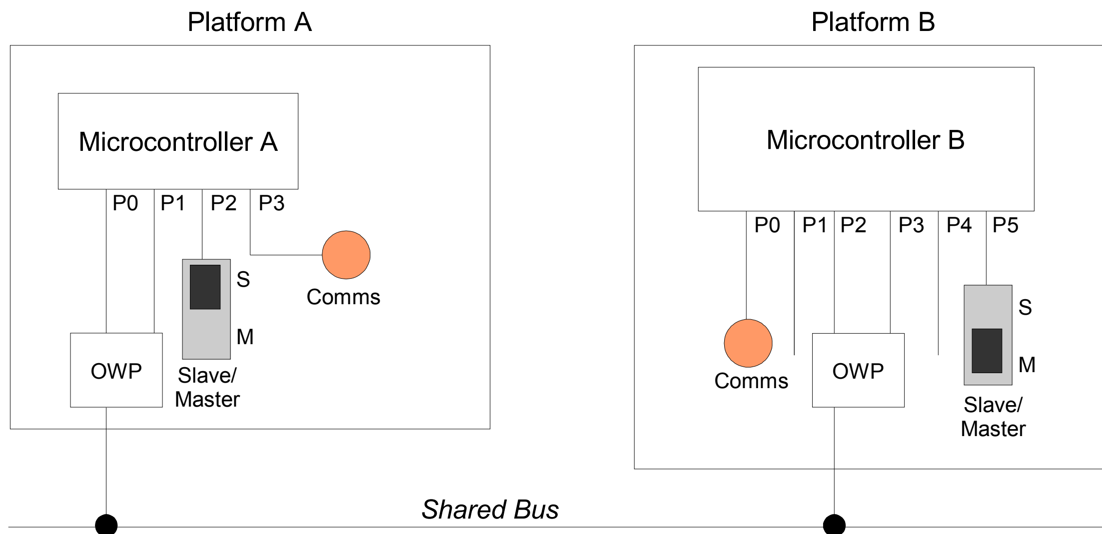


Illustration 4: Example for demonstration the use of PIV values.

Clearly, platforms A and B in Illustration 4 have a certain number of components in common (the Comms LED, Slave/Master Switch, and two pins connecting to the OWP). But, due to design conditions, these components are not accessed using the same pins. For example, turning on the Comms LED on platform A requires setting the third I/O line (P3); while on platform B it requires setting the first I/O line (P0). The same thing occurs when accessing the other "common components". Moreover, the way that an I/O line is set or cleared may differ; for example Microcontroller A may be an AT91 that uses two separate registers, PIO_OER to turn on, and PIO_ODR to turn off, the line; while platform B may be a simple PIC that uses the same register for setting and clearing an output line.

A *Platform Integration Value* (PIV) is used to provide platform-dependent information to application code in a format and data type that is generic between supported platforms. Typically, a PIV is defined as a word and set to a power of two, used to reference a control bit of a peripheral register. You can think of PIVs as something like the keys used in database programs to uniquely reference an entry in a table. This is somewhat of a simplification for PIVs: they could be a more complex data structure used to pass structured data between application code and platform deployment modules[4].

Consider again the platform A and platform B example again. Code for platform A may define a constant value LED_COMMS equal to 8 to indicate that the Comms LED is connected to the 4th I/O line on the microcontroller. But code for platform B would need to define the value of LED_COMMS equal to 1, because on that platform the Comms LED is on the first I/O line of the microcontroller.

C code used to implement common component functions may be exactly the same for both platforms; but the PIV definitions would likely change between platforms. Thus, instead of having to maintain two code modules, in which only the PIV definitions change, it makes more sense to put all the PIV values for a particular platform in one H file dedicated to that platform. Then, when compiling a module for a certain platform, only the PIV H file related to the platform concerned is included during the compilation. In such a situation, the implementation of the CCW functions can be provided at a higher level in the Software/PDM directory (e.g. in directory Software/PDM instead of PDM/CSB337/MicroMonitor).

The design of the ESAOA framework is influenced by the notion of separate PIV.h files for each platform. In the PDM (Platform Deployment Modules) directory, you will find a PIV.h in each directory that implements platform deployment code for a certain platform. For example, PIV.h in PDM/CSB337/MicroMonitor defines platform integration values for code developed for application software to be executed under MicroMonitor

4 An example of a more complicated PIV is a date/time structure used to interact with a platform's real-time clock (RTC). One platform may have a low resolution RTC that uses only 32 bits to store a date and time; a second platform may require 64 bits. The PIV.h for the first platform may have *struct DT { int t }*; the PIV.h for the second platform *struct DT { int t; int ms; }*. Platform-independent code would then pass a DT variable to communicate with CCW functions to interact with the RTC.

on the CSB337 hardware platform.
Although the use of PIVs may appears to make platform-independent code into platform-dependent code, this does not happen if the following conditions are maintained:

PIV.h Usage Strategy

- Platform-independent code modules using PIV.h must not make assumptions regarding the values defined in the PIV.h file (e.g. on some platforms the bit number of LED2 may be the next sequential bit after LED1; but that may not be so for all platforms, so a command like led_on(LED1 * 2) in order to turn on LED2 is not safe).
- Each platform needs to be provided with the same set of PIV variables and macros, but the value of these can differ between platform PIV files.

Disclaimer for the ESAOA structure using PIV.h and CCW interfaces: the proposed structure is not guaranteed effective for all projects. These are general guidelines to aid the reader in writing more portable and reusable code. If portability and reuse is *not* an important consideration for you, then this approach is not applicable.

Q-4: A PIV is used to... (choose the correct answer)
(a) Pass platform-dependent data through platform-independent CCW interface functions without having to defining platform-dependent items within application modules.
(b) Define application-specific information for use only in application modules.
(c) Provide data structure definitions accessible only to PDM modules.

STEP 2: Defining PIVs

We're finally getting closer to doing some coding and experimenting...

We need to determine which bits of the PIOB port correspond to which of the LEDs. You need to use the CSB337 usermanual to solve this problem. The user manual is called csb337-usermanual.pdf and is stored on forge.ee in directroy /EEE3074W/Documentation/Hardware/Datasheets/CSB337 (running command **ds 2** will make xpdf display the document if X windows is configured). Look at the table on page 13 explaining **Port B GPIO Assignments**. There are three user LEDs available, LED0, LED1, and LED2.

Open up the file **PIV.h**, and look for **TODO[3.1]**. Fill in the bit numbers for the LEDs[5], i.e. by replacing the value ***your_value*** with what you think the value should be. I already did the entry for LED_USER0. If you prefer, you can include the header file AT91RM9200.h in PIV.h, and then assign, for example LED0 to AT91C_PIO_PB2.

STEP 3: Configuring LEDs

Next, you need to instruct the PIOB *controller* that you want to assign the bits corresponding to the LEDs for use as GPIO (i.e. to make the PIO controller control the PIO pads that the LEDs are connected). These bits need to be configured as outputs -- slot in two lines of code under **TODO[3.2]** to do this. In order to figure out how to do it, take a look at the section on the PIO in the AT91RM9200 datasheet, starting on pg 333, although pg 341 might give some particularly useful insights. Also look at the rest of the *init_pio* function. You can make use of the **AT91S_SYS** structure defined in the file **AT91RM9200.h**.

The following appendices will be useful in this and the next step:

- Appendix C1: How to control a PIO control register
- Appendix C2: Setting and clearing data registers

5 There are some options here: you could set LED_USER0, LED_USER1, etc to bit values (i.e. powers of two, that correspond to bit numbers), or you could use an index (0, 1, 2) and then implement in code an array or function to convert the index to a power of two. The array method is fast but possibly unsafe.

>>>>> PANIC QUESTION <<<<<

Q-5: Why would the statement *AT91_SYS->PIOA_PSR = 0x4* do absolutely nothing when run on the AT91RM9200? i.e. the 3rd bit of the PSR register for PIOA would not be set if it was previously clear.

SUMARY OF WHAT TO DO TO CONFIGURE PIO AND USE THE LEDS
1. Enable PIO for each bit corresponding to the LEDs. i.e. set AT91_SYS->PIOA_PER to the value LED_USER0 | LED_USER1 | LED_USER2
2. Configure the bits as outputs, i.e. set AT91_SYS->PIOB_OER to the value LED_USER0 | LED_USER1 | LED_USER2
3. To turn off an LED: AT91_SYS->PIOB_SODR = LED_USER0; // turn *off* LED 0
4. To turn on an LED: AT91_SYS->PIOB_CODR = LED_USER0; // turn *on* LED 0

Notice that the LEDs use reverse logic (when you set the PIO pad high, it turns off the LED).

STEP 4: Turn On/Off LED0

Now that you've completed step 3, it should be fairly easy to to turn on, and then turn off the LEDs using writes to the **PIOB_CODR** and **PIOB_SODR** data bit registers, so that you can test your code.
See item **TODO[3.3]** in init.c. Also implement the slight delay as requested; **DO NOT** call the **pause()** function as the timer is not yet configured; you could call short_pause or implement a delay as in prac3.

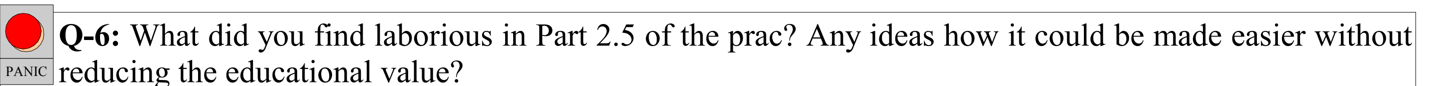
STEP 5: Flash LED0 in loop

Find and complete **TODO[3.4]** and **TODO[3.5]** which respectively turns off and on the LED.

Reflections on Section 2.5

Now you've experienced in one of the more frustrating aspects of programming embedded systems: trying to figure out what pins to control, how to control them, and the irritation associated with *inverse logic* (where false is true, and true is false – for the record, that's *not* why I chose this evaluation board). Of course, in a real project, this could be even more laborious without some pointers on which fiddly bits of code need to be changed, and where to find the required information. In some companies, you may be fortunate to have an expert colleague sit down with you and show you what he does in such situations; but oftentimes, and speaking from my own experience, such an expert is not always available, and it's up to you to find a solution.

So, lets take a moment to think about this tricky step of learning how to configure hardware from software, as it is a problem that all embedded system developers encounter, but occurs especially regularly for novice developers. Take a few minutes now to consider how you could approach this problem in a better way in future... To ensure that you are pondering these things, I'd like you to jot down a few points (at least two) on what you found most laborious and how you think it could be better approached (see **Q-6** below).



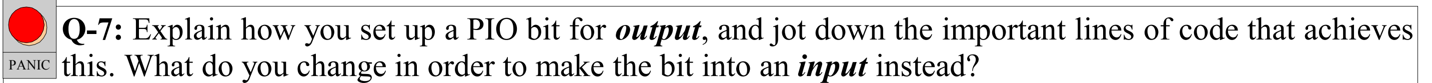
Q-6: What did you find laborious in Part 2.5 of the prac? Any ideas how it could be made easier without reducing the educational value?

CHECKLIST for 2.5: TODO [3.1] – [3.5], Q-4, Q-5, Q-6 *TIME ESTIMATE: 80 minutes*

2.6. Pushbuttons TODO[4]

It should be pretty obvious now how the pushbuttons work and they are already configured as inputs in the init_pio function. Implement item **TODO[4]**, so that the while loop in init_pdm terminates when pushbutton 1 is pressed. (You do not need to implement button debounce as the program terminates when it is pressed).

Hint: Ignore or remove the (pb_pushed < 3) expression in the while condition and add a read of pushbutton 1.



Q-7: Explain how you set up a PIO bit for ***output***, and jot down the important lines of code that achieves this. What do you change in order to make the bit into an ***input*** instead?

CHECKLIST for 2.7: TODO [4], Q-7 *TIME ESTIMATE: 30 minutes*

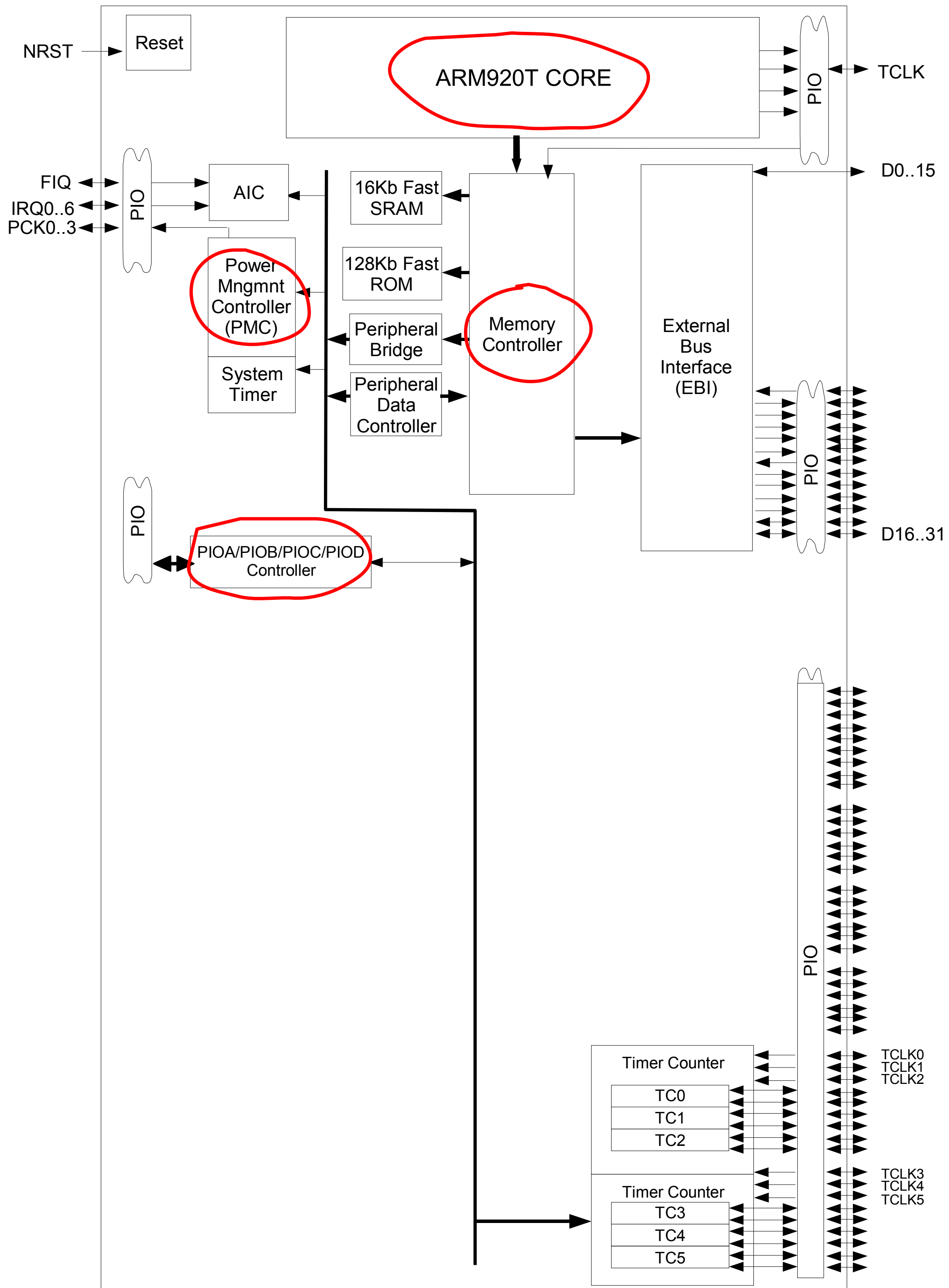
3. Finalization

Tar and gzip your modified Prac04 project folder (you can execute the ESAOA commands **h; tz** to do this – h gos to the project room, and tz makes a tar.gz archive of the current directory). Submit using the Practical 4 assignment on Vula.

TOTAL TIME ESTIMATE: 180 minutes

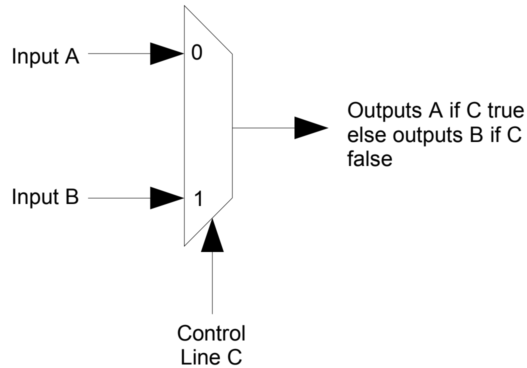
APPENDIX A: Block Diagrams

AT91RM9200 Block Diagram Excerpt

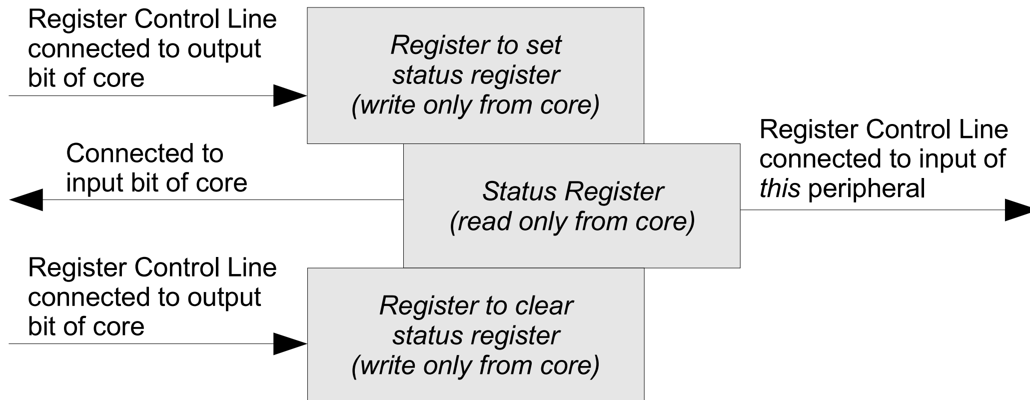


APPENDIX B: Diagrammatic Notations

Multiplexer



ATMEL Register Notation



APPENDIX C: Coding Procedures

C1: How to control a PIO control register

The PIO registers come in sets of three. The Diagrammatic notation used in ATMEL datasheets is explained in Illustration 5. The middle register, called the Status Register, stores a value and can only be read by the microcontroller core. The top register (or set, or enable register) is used to set the status register to logic 1. The bottom register (or clear, or disable register) is used to clear the status register.

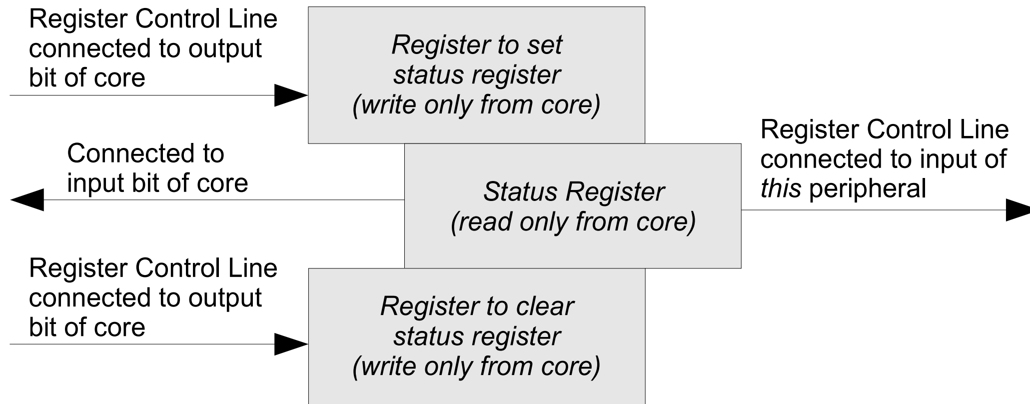


Illustration 5: Diagrammatic notation used by ATMEL for control register sets.

Each control register has 32 bits. All bits of the register work in the same way.

Example of setting a PIO control register

Lets consider an example: you want to set the **third bit of the PIOA status register** (PIOA_PSR) to **logic 1**. You will want to be in directory to follow this procedure Proceed as follows:

1. Search for PSR in file AT91RM9200.h (e.g. grep PSR AT91RM9200.h).

   You will immediately see there are a whole lot of results:

   <u>*Sample Output:*</u>
   ```
   AT91_REG         PIOA_PSR;      // PIO Status Register
   AT91_REG         PIOB_PSR;      // PIO Status Register
   AT91_REG         PIOC_PSR;      // PIO Status Register
   AT91_REG         PIOD_PSR;      // PIO Status Register
   AT91_REG         PIO_PSR;       // PIO Status Register
   #define AT91C_PIOD_PSR  ((AT91_REG *)   0xFFFFFA08) // (PIOD) PIO Status Register
   #define AT91C_PIOC_PSR  ((AT91_REG *)   0xFFFFF808) // (PIOC) PIO Status Register
   ....
   ```

The first four lines, showing `PIOA_PSR`, `PIOB_PSR`,etc, are particularly interesting. The AT91RM9200.h actually defines multiple structures that reference the same functionality (since some programmer prefer one method to another). We will unceremoniously use the method I like...

2. Open AT91RM9200.h and scroll down to the first occurrence of PIOA_PSR (in vi, in command mode you can type in /PIOA_PSR and then hit enter to find it). Clearly, PIOA_PSR is a field within a structure, called _AT91S_SYS (with typedef name mapped to AT91S_SYS). The PIOA_PSR is of datatype AT91_REG. Further investigation of the file shows that AT91_REG is defined as *volatile unsigned int* (i.e. whenever the value of a variable of that type is read or written, the memory location is read or written, not just a cached value for that variable).

3. The AT91S_SYS structure itself defines all control registers available to the microcontroller core. Therefore, if we want to make use of this structure we need to:
   a) Instantiate a pointer of type AT91S_SYS that points to the start address of the control registers... this has already been done for you in init.c: if you search for AT91PS_SYS, you will find it used near to top of the file to instantiate the global pointer variable named AT91_SYS which points to address AT91C_BASE_SYS (which is the starting address of the control registers).
   b) We want to assign the register somewhere in the code (e.g. after the comment containing TODO[3.2]). To do so, you may assume you can use the field PIOA_PSR in the structure AT91S_SYS just like you would for assigning a normal variable, for example:

```
AT91_SYS->PIOA_PSR = 0x4;  // assign the 3rd bit to logic 1
```

      If you were to do the above, you would see no result. *Why?... see panic question Q-5 ...*

   BUT the following line WILL achieve the desired result of setting the 3rd PIOA_PSR bit to 1.

```
AT91_SYS->PIOA_PER = 0x4;  // assign the 3rd bit to logic 1
```

NOTE: Instead of using a number value such as 0x4, you should rather use a defined PIV value to make the code both more portable, and more readable. For example, when doing this operation for configuring the LEDs, you could say :

```
AT91_SYS->PIOA_PER = LED_USER0;  // configure 3rd PIO pad as GPIO to access LED0
```

C2: Setting and Clearing Data Registers (to set/clear PIO pad)

The clear output data register (**PIOB_CODR**) and set output data register (**PIOB_SODR**) memory addresses need to be written to in order to change the bit status of the PIO registers (reading these memory addresses are of no use whatsoever). Writing the value of 0xFFFFFFFF to PIOB_CODR will actually set all the output bits to 0, while writing the value of 0 to PIOB_SODR will do nothing at all (I know: it leads to rather counter-intuitive code; but that's life).