Microcontroller Lab Exercises

March 2, 2010

Preface

This set of exercises covers all course material, ranging from Assembler- (and C[1])-Programming over in-depth treatment of the AVR microcontroller to the external hardware used. It has been designed to accommodate a wide range of experience levels, from novices to students with previous microcontroller experience.

To help you assess the difficulty of the exercises, we have rated each exercise with a difficulty level (DL), ranging from 0 (preparatory), 1 (Easy) to 5 (Difficult). Furthermore, if an exercise is based on knowledge conveyed in another exercise, this dependency is often stated as well to help you plan your schedule. For recommended exercises rated ≤ 1, we have put in references to the manuals and datasheets to help you get your bearings.

Since you should have the freedom to customize the course to your needs, only some of the exercises, which cover the basic skills and knowledge necessary to pass the course, are recommended. From the remaining exercises, you may choose as you please.

Microcontroller programming and debugging can be very challenging and satisfying. Exploring unexpected behavior, formulating theories, testing them, and finally coming up with an explanation is both thrilling and extremely instructive. We hope that our exercises will trigger your interest and motivate you to explore relationships and interesting hardware behavior beyond the scope of our course, while at the same time conveying microcontroller programming skills and some basic hardware/electronics skills along the way.

We wish you a lot of fun!

Acknowledgements

Our thanks go to the following people for contributing exercises to this collection[2]:
Martin Biely, Thomas Mozgan, Christian Trödhandl

[1] We expect you to be proficient in C. However, we offer some C exercises so you can refresh your knowledge.

[2] If you have an idea for an application exercise that you believe is interesting and which can be done with our course hardware, mail your suggestion to mc-org@tilab

General Remarks

- Only set/clear the bits you need. Leave all others unchanged.
- From Part II on, use timers for all delays unless the exercise explicitly states that you may use busy-wait.
- Use the controller features whenever possible (e.g., let the controller generate a PWM signal if possible).
- Use interrupts instead of polling except when explicitly requested.
- Enter an appropriate sleep mode whenever the situation warrants.
- Avoid floating point operations except when explicitly requested.
- From Part II on, pay attention to keeping your code re-useable (in particular the display, keypad, and UART code).

Recommended Exercises

The following exercises are recommended.

Note: All exercises of Part I should be programmed in Assembler.

Part I	**Part II**	**Part III**
1.1, 1.2, 1.3, 3.1, 3.2, 4.1, 4.2		

Contents

1 Assembler Programming

The exercises of this section have to be programmed in Assembler.

1.1 Demo Program (DL0)

Based on: 9.1, 9.2

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)

Take a look at the demo program you can download from the course homepage (topic `Manuals`, ATmega16-docs). The program contains one syntactical error. Call `make` to assemble the program, identify the location of the error from the assembler output, correct the error, and assemble again. Then download the resulting hex-file to the controller. As soon as the download is finished, the controller starts executing the program. If necessary, ask your tutor for help connecting LED7:0 and CC_LEDS to see what the program is doing.

Play around with the program to familiarize yourself with the assembly language. Make sure you know how to initialize the stack pointer, how to declare and use variables, how to set registers, how to jump to a label, how to do loops, how to call subroutines, and how to place code at specific memory locations.

References

- Assembler Tutorial
- Simple I/O board
- ATmega16 controller board
- ATmega16 manual, stack pointer: p. 12
- ATmega16 manual, general purpose register file: p. 11
- ATmega16 manual, SRAM data memory: p. 17

1.2 Makefile (DL0)

Based on: 1.1

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)

Take a look at the makefile demo program you can download from the course homepage (topic `Manuals`, ATmega16-docs). The program contains one syntactical error (see exercise 1.1). Get familiar with the toolchain and write a Makefile that assembles the demo program

and downloads it to the microcontroller. Do NOT use the one provided by the assembler demo program. If necessary, ask your tutor for help connecting LED7:0 and CC_LEDS to see what the program is doing.

Play around with the makefile to familiarize yourself with the toolchain and the options to the assembler and the linker.

Questions

1. Our microcontroller has 3 different memories: Flash, EEPROM and SRAM. Which memory can be programmed how (during program execution / at program download)?

References

- Assembler Tutorial
- Simple I/O board
- ATmega16 controller board

1.3 Logical Operations (DL1)

Based on: 1.1

Pin Assignment

Simple I/O board	
LED3:0	PC3:0
CC_LEDS	VCC (+)

Task

Load a suitable free register Ra with a constant. This constant will be changed by you several times during testing, so make sure it can be changed easily and only in one place (use `.equ`). In your program, implement the following operations and display their results on the LEDs:

- LED0 := (Ra1 $\wedge$ Ra2) $\vee$ Ra3
- LED1 := (Ra1 $\vee \neg$ Ra2) $\wedge$ Ra3
- LED2 := Ra1 $\oplus$ Ra2 [Antivalence]
- LED3 := Ra1 $\Leftrightarrow$ Ra2 [Equivalence]

Remarks

- Ran denotes bit n of register Ra. Bit numbering starts with 0.
- Check out the bit load/store instructions. Also familiarize yourself with the bit test instructions and the skip instructions, they may come in useful (possibly in later exercises).

Questions

1. Do you need to set the stack pointer? Why or why not?
2. What is the reset value of the SP? When the SP is needed, why should one set it initially to the end of the SRAM?
3. What is the first address of the data memory that is available for the user program? What is located before that address?
4. There are two different types of set/clear bit operations: Those that operate on registers in general (SBR/CBR), and those that operate on I/O registers (SBI/CBI). Explain the differences between the two types.
5. Explain the differences between NEG and COM. If you want to invert a single bit in a register, which of the instructions NEG, COM, EOR can you use? (Show how; multiple answers possible.)

Pitfalls

- Note that the bit set and bit clear instructions work only on the first 32 I/O registers, see Remark 4 in the ATmega16 manual, page 335.
- When using `.equ`, do not use names starting with {r|R}. The Assembler expects a register number after the r.

References

- ATmega16 manual, instruction set: p. 336–338
- ATmega16 manual, architectural overview: p. 8–15

1.4 Fibonacci Numbers (DL2)

Based on: 1.1

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)

Task

Use 32 bit integer arithmetic to calculate the n-th Fibonacci number $F(n)$. After calculating the number, display the 4 bytes of the result, starting with the MSB, on the LEDs. Between each byte, wait for one second (adapt the busy-wait loop of the assembler demo program). After the last byte clear the LEDs, wait for another second and start again.

Remarks

- Define the index n for which the Fibonacci number is calculated in only one place (with `.equ`), so that it can be changed easily.

Questions

1. What is the highest n for which the Fibonacci number can be stored in a 32 bit integer?
2. If you do a recursive implementation without optimizations, how much stack do you need to compute $F(n)$? Give a formula for the stack usage $S(n)$ (with proof).
3. How much time $T(n)$ does your implementation take to compute $F(n)$? Give an estimation of $T(n)$ (upper and lower bound) in both clock cycles and seconds.

Pitfalls

- Do not implement a recursive solution, it is very inefficient.

Hints

- The Fibonacci numbers are defined by the formula $F(n) = F(n-1) + F(n-2)$, where $F(0) = 0$ and $F(1) = 1$.
- To compute $S(n)$, first write down the stack usage for $n = 0, 1, 2, \ldots, 5$. You will see a certain pattern and a correlation with $F(n)$ from which you can derive a formula. Prove this formula by induction.

1.5 Sieve of Erathostenes (DL2)

Based on: 1.1

Pin Assignment

Simple I/O board	
LED3:0	PC3:0
CC_LEDS	VCC (+)

Task

Program the Sieve of Erathostenes for prime numbers between 2 and TOP. Display the number of primes in $[2, TOP]$ on the LEDs (adapt the busy-wait loop of the assembler demo program if you need to wait between nibbles).

The algorithm works like this: Mark all numbers between 2 and TOP. Select the first marked number (2) and clear your marker from all multiples of this number (but not from the number itself). Select the next marked number (3) and again clear all its multiples. Repeat this (5,7,11, ...) until there are no more remaining marked numbers. All numbers still marked are prime numbers.

Remarks

- Define TOP in only one place (with `.equ`), so that it can be changed easily.
- Use subroutines for deleting multiples and for counting the remaining prime numbers.

Questions

1. Let us assume that you have initially set the SP to the end of the RAM. If you push the contents of R0 on the stack after it has been initialized, then at which address (the actual address, no symbols please) is the data located and to which address does the SP point?
2. What would happen if the program did not initialize the SP? Could you overwrite code? Could you overwrite data?
3. Can you easily state the time complexity of the algorithm with respect to deleting multiples (assume that the array size is n)? If so, do it. If not, explain why.

Pitfalls

- 1 is not a prime number. Start with 2.
- If a subroutine changes registers that you want unchanged in your main routine, push them on the stack before calling the subroutine (and don't forget to pop).

Hints

- Use an array (some consecutive and unused portion of the SRAM) to store the markers, and use one byte per number to make things easier.
- Use indirect addressing with post-increment to iterate through the array.
- For TOP=100, the result you want to see on the LEDs is 25.

2 C Programming

The exercises of this section have to be programmed in C.

2.1 Demo Program (DL0)

Based on: 9.1, 9.2

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_DIGIT0	VCC (+)
BTN7	INT0
SW1	PA4
SWCOM_1-3	GND (⊥)

Task

Download the C demo program from the course homepage. It demonstrates interrupt handling in C and also shows the effect of the `volatile` declaration. The program counts up a variable whenever you press BTN7 and displays its value on DIGIT0. Use SW1 to switch between a variable that was declared `volatile` and one that wasn't.

Remarks

- Take care when using the pre-defined interrupt macros: If you have a typo in the macro (e.g., you type SIG_INT0 instead of SIG_INTERRUPT0), the compiler will *not* warn you, and your ISR will simply not be called!
- Note that the default handler for interrupts causes a reset, so if you enable an interrupt, you should also install an ISR for it.

References

- avr-libc Reference Manual 1.0.3 (Interrupts: p. 69)
- avr-libc Reference Manual 1.0.3 FAQ section (recommended reading!)

2.2 Floating Point Operations (DL0)

Based on: 9.1, 9.2, 9.3

Pin Assignment

Simple I/O board	
SW1	PA4
SWCOM_1-3	GND (⊥)

Globally define the following variables:

```
volatile uint16_t erg;
float fA = 0.8;
float fB = 0.5;
uint16_t A = 8;
uint16_t B = 5;
```

Initialize PC0 as output and set it to zero. In your program, set PC0 to 1, then check the value of SW1 and execute `erg = (uint16_t)(10*fA + fB)` if SW1 is ON and `erg = (10*A + B)/10` if SW1 is OFF. After that, set PC0 back to 0 and enter an empty endless loop.

As is apparent, both variants compute the same result, which is 8 (the $+0.5$ resp. $+5$ are for rounding to the nearest integer). The first uses floating point operations, the second computes the same result, but converts the floats to integers (by multiplication with 10 to do away with the comma), does everything in integer arithmetic, and then (integer) divides the result by 10 to revert the previous multiplication.

Connect the oscilloscope to PC0 and measure the duration of the two calls (reset the controller to restart with a different SW1 setting).

Questions

1. How long is the execution time of each of the functions (you can ignore the constant time that is added by checking the switch and calling the function)?
2. Compare the code sizes of the two functions (check the list file). What is the difference in code size (in bytes)?
3. What conclusions can you draw from your previous answers?
4. As an experiment, put the keyword `const` before the declarations of `fA` and `fB`. Measure the execution times again, and take a look at the list file. Describe your observations. What conclusions do you draw?

Pitfalls

- Make sure the variable definitions are global, that is, outside of `main()`, or you won't see any difference.

3 Digital I/O

3.1 Input With Floating Pins (DL1)

Pin Assignment

Simple I/O board	
LED1:0	PC1:0
CC_LEDS	VCC (+)
SW1	PA4
SWCOM_1-3	GND (⊥)

Task

Set pin PA4 to input and do *not* activate the pull-up. In the main loop, display register bit PINA4 on LED0 and bit PORTA4 on LED1.

Questions

1. Set SW1 to position OFF. Now wave your hand closely over the board, tap or wiggle the wire connecting SW1 to the controller. What do you see on the LEDs? What can you expect to see?
2. Set SW1 to position ON. Repeat the actions of the previous question. What do you see now on the LEDs? What can you expect to see?
3. Explain the difference between the PORT and PIN registers when a pin is set to input.
4. For completeness' sake, activate the pull-up and repeat the experiments with the switch. How does it behave now? Where are changes compared to the behavior without pull-up?

References

- ATmega16 manual, I/O ports: p. 50–67, registers PORTx, DDRx, PINx
- Simple I/O board
- ATmega16 controller board

3.2 Digital I/O (DL1)

Based on: 3.1

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_DIGIT0	VCC (+)
SW1	PA4
SWCOM_1-3	GND (⊥)

Task

Implement a 4-bit counter that counts within 0 and 15, and display its value as a hexadecimal number ('0' to 'F') on DIGIT0 of the numeric display. While SW1 is in position ON, the counter should count up. If SW1 is set to OFF, the counter should reverse its direction immediately and count down. Let it wrap when it reaches its bounds.

Use a suitable time between counter increments that allows to track the change of the counter on the display (you can employ a busy-wait loop).

Remarks

- Store the values for the digits '0' to 'F' in an array (SRAM or Flash).

Questions

1. What is the difference between SW1, SW4, and SW7?
2. What happens if you set both CC_DIGIT0 and CC_DIGIT1 to VCC (+)? Explain why. Can adding more digits in such a manner change the brightness of the display?
3. Assume that you have n numeric displays, and each LED of the display requires 2mA. How many digits can you connect (LEDs connected in parallel) to the microcontroller (the cathodes of the displays are connected to GND ($\perp$))? (You can assume that there is nothing else connected to the microcontroller, only the LEDs of the numeric displays.) Please list all electrical characteristics you have to consider.

Pitfalls

- You have to prevent SW1 from floating.
- If you use the SRAM for your array, note that the SRAM cannot be initialized automatically, you have to initialize the array within your program.
- Your program should react to state changes of SW1 immediately (and not just at counter values 0 or 15)

Hints

- To find out how many digits you can use simultaneously, you have to check the electrical characteristics. The important aspects are how much current one pin can source, and how much current the whole port can source.

References

- ATmega16 manual, I/O ports: p. 50–67, registers PORTx, DDRx, PINx
- Simple I/O board
- ATmega16 controller board

3.3 Voltage Levels (DL1)

Pin Assignment

Simple I/O board	
LED0	PC0
CC_LEDS	VCC (+)
TRIM2	PA0

Task

Set PA0 to digital input and continuously display its state on LED0. Connect the oscilloscope to TRIM2 and LED0.

Questions

1. Which voltage levels map to a logical ”1”, which map to a ”0”? When does the controller switch from one logic state to the other?
2. What are the theoretical bounds for the voltage levels as stated by the manual? Do the match your observations?
3. If you set TRIM2 to a voltage level between the upper bound for ”0” and the lower bound for ”1”, does this increase the probability for a meta-stable state?

3.4 I/O Delays (DL1)

Pin Assignment

Simple I/O board	
LED2:0	PC2:0
CC_LEDS	VCC (+)

Task

Set PA3 to output. In the initialization part of your program, put in the following code:

```
SBI PORTA, 3 ; set PA3 high
IN Ra, PINA ; read PINA
IN Rb, PINA
IN Rc, PINA
```

Display the value of the third bit stored in Ra on LED0, the third bit of Rb on LED1, and the third bit of Rc on LED2. Ra-Rc can be any suitable free registers. The main loop of the program should be empty.

Questions

1. What do you see on the LEDs and why?

2. Judging from the output, what is a lower bound on the input delay? What is an upper bound? Can you give tight bounds just by observing your program's output? Justify your answers.
3. What do your observations imply for your programming practice?

Pitfalls

- When using `.equ`, do not use names starting with {r|R}. The Assembler expects a register number after the r.

References

- ATmega16 manual, input delay: p. 52–53

4 Interrupts

4.1 Interrupts (DL1)

Pin Assignment

Simple I/O board	
LED3:0	PC3:0
CC_LEDS	VCC (+)
BTN7	INT0

Task

Initially turn on LED0. Write an interrupt service routine (ISR) that rotates LED3:0 to the left by one whenever it is called (that is, if LEDi is on prior to calling the ISR, then after calling the ISR LED$\{i + 1 \bmod 4\}$ should be on. Install this ISR as the INT0 ISR and let the LEDs rotate by one whenever the button is pressed down.

Questions

1. Explain the difference between the RET and the RETI instructions. Why do you need RETI to return from an ISR? What happens if you use RET instead?
2. What happens if you are in an ISR and an interrupt with higher priority occurs? Will your ISR get interrupted (assume that you do not touch the Global Interrupt Enable bit in the ISR)?
3. Some experienced programmers save the status register on the stack in the ISR, regardless of what the ISR does. What reasons could they have?

Pitfalls

- Note that the addresses given in the interrupt vector table of the ATmega16 manual (and defined in `m16def.inc`) are *word* addresses, whereas the Assembler requires byte addresses.

Hints

- The Assembler instruction SWAP might be helpful. Also, the half-carry bit H might come in handy.
- Like with most controllers, the port pins of the ATmega16 have alternate functions. You just have to enable the alternate function (in this case, the external interrupt function) to be able to use it.
- After the reset, interrupts are globally disabled. Use the SEI/CLI instructions to globally enable/disable interrupts.
- Use the vector table on page 45 of the ATmega16 manual to determine the address for the INT0 vector. The microcontroller expects a jump instruction to your ISR at this address (see the example code on p. 44).

References

- ATmega16 manual, reset and interrupts: p. 13–15
- ATmega16 manual, interrupt vectors: p. 45–46
- ATmega16 manual, external interrupts: p. 68–70 registers MCUCR, MCUCSR, GICR
- ATmega16 manual, status register: p. 9–10, register SREG
- ATmega16 manual, instruction set: p. 336–338, instructions SEI, CLI, RETI

4.2 Interrupt Latency (DL1)

Pin Assignment

Simple I/O board	
LED0	PC0
CC_LEDS	VCC (+)

Task

Initialize LED0 for output and turn it off. Initialize INT0 as output and set it to 0. Turn on the external interrupt feature of INT0 and let an interrupt be generated for the rising edge. Then set INT0 to 1. In the INT0 ISR, turn on LED0 in the first instruction. Use the oscilloscope to determine the interrupt latency from the signals on INT0 and LED0.

Questions

1. Which delays make up the interrupt latency of the AVR controller?
2. What additional delays (apart from the latency itself), if any, are part of the time you have measured?
3. What determines the accuracy of your measurement? How accurate can you get?

4.3 Interrupt Modes (DL1)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
SW2:1	PA5:4
BTN7	INT0
SWCOM_1-3	GND ($\perp$)

Task

Configure INT0 as an input and install an ISR for it. Take the interrupt mode configuration of the MCUCR from SW2:1 (if SW2:1 are both ON, that is, they both read 0, then the interrupt mode bits should be set to 0). In the INT0 ISR, rotate the LEDs (LED7:0) by one (initially, LED0 should be turned on). The ISR should react when BTN7 is pressed down.

Remarks

- You may put a delay into your ISR for observation purposes (especially for observing the effect of the level interrupt). But please be aware that busy-wait delays in ISRs are A Bad Thing and should be avoided.

Questions

1. For each of the interrupt modes, describe what happens when you press/release the button and while it is pressed.
2. If you set the interrupt mode to low level and press the button for one second, how often is the ISR called?
3. For each interrupt mode, state which sleep modes it can interrupt.

Pitfalls

- Do you need the pull-ups for the switches and the button?

Hints

- You can compute the answer to Question 2 from the information in the ATmega16 manual. But if you cannot figure out how to do it, you may also determine the value experimentally. But please state in the protocol that you did this and describe how you determined the value.

5 Timer

5.1 Input Capture (DL1)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
BTN7	INT0

Task

We want to measure the time between button presses. To this aim, activate the timer with prescaler 1024 and just let it run freely. Install an ISR (SIGNAL) for BTN7 which triggers an input capture interrupt on ICP1. In the ICP ISR, read the captured timestamp and compute the time since the last capture interrupt. Display the time (in seconds, with one place behind the comma) since the last capture on the LEDs. Use a BCD representation and display the seconds (0-9) on LED0-3, and the fractional part (0-9) on LED4-7.

Remarks

- This exercise is to show you both the input capture feature and the fact that input captures can be triggered by software by setting the ICP1 pin. Therefore, pin ICP1 is not connected to any external device, and its signal is controlled by software. Were this a real task, you would of course connect button BTN7 directly to ICP1.
- You can only display values from 0.0 to 9.9 seconds. To determine the value you should display, use round-to-nearest. In case of an overflow, turn on all LEDs.
- Do not forget that the timer can wrap more than once between two button presses. You need to count those wrap-arounds.
- Note: Normally, it would be a better solution to connect BTN7 directly to ICP1. But in this exercise, we took the opportunity to not only show you how input capture works, but also how an input capture event can be triggered by software.

Questions

1. Would it have been sufficient to simply read the current count register in the INT0 ISR instead of triggering an input capture interrupt?
2. How close together can two interrupts occur and still be distinguished (an estimation suffices)?
3. Which events can trigger an input capture interrupt?

Pitfalls

- Do not forget to set ICP1 to output.

Hints

- To get the number you want to display, compute the time T between two capture interrupts in a unit of 100ms. Display T/10 on LED0-3, and T%10 on LED4-7.

5.2 Phase (and Frequency) Correct PWM Mode (DL1)

Task

Use Timer 1 in Phase Correct PWM mode to generate a PWM signal on PD5. The signal should have a high time of 75 μs and a period of 100 μs. Verify the timing of the signal with the oscilloscope.

As an experiment, set the timer prescaler to its highest value (keep all other timer settings – the period will get longer). Again, verify the timing of the new signal with the oscilloscope. If there are any deviations from the expected signal timing, correct them and document this in the protocol.

Remarks

- Let the timer generate the signal automatically.
- The oscilloscope is accurate well below 1 μs.
- When you initialize the timer, it is best to first select the mode, then set the registers (like OCR or whatever you need), and last start the timer.

Questions

1. Which timer modes can you use for this exercise? Which one did you choose?
2. Explain the differences between Phase Correct and Phase and Frequency Correct PWM.
3. How can you generate a Phase Correct PWM signal that is constant (either constantly low or constantly high)?

Pitfalls

- Do not forget to set PD5 to output (see ATmega16 manual, p. 100).
- Note that the timer counts from 0 to (including!) TOP (or OCR1A) before raising an interrupt, so the duration is TOP+1 ticks. This is easily overlooked and does not make much difference for a small prescaler, but makes a huge difference for a large prescaler.

References

- ATmega16 manual, Timer 1: p. 89 f.
- ATmega16 manual, Timer 1, Timer Modes: p. 101–110
- ATmega16 manual, Timer 1, Phase (and Freq.) Correct PWM: p. 104–108 and p. 111 f.

5.3 Generating Periodic Signals (DL2)

Task

You want to generate a periodic signal on PD5 which is high for 50 μs and low for 25 μs. Consider the following ways of achieving this signal:

(1) **Busy-Wait:** In your main loop, you set PD5 to high and enter a busy-wait loop that waits for 50 μs, then you set PD5 to low and enter another loop that waits for 25 μs.

(2) **Timer Overflow:** You set PD5 to low and set up Timer 1 in normal mode to generate an overflow interrupt after 25 μs. In the overflow ISR, you set PD5 to high and set up Timer 1 to generate an overflow interrupt after 50 μs. In the next call of the ISR, you set PD5 to low and set the timer to 25 μs and so on. (Use the register values for 25/50 μs here, do not adjust for the latency and ISR code.)

(3) **Output Compare:** You set PD5 to low and set up Timer 1 in CTC mode (OCR1A) to toggle PD5 after an output compare match and to generate an output compare interrupt after 25 μs. In the output compare ISR, you alternately set the OCR-register to 50 μs and 25 μs.

(4) **PWM:** You configure Timer 1 to generate a Fast PWM signal using ICR1 as TOP register.

Implement each of these methods (write four programs). For interrupt-driven methods, enter an appropriate sleep mode in main.

Remarks

- Do not spend too much time to get method (1) completely accurate. But do try to come close. The point here is to show how hard it is to wait for a specific amount of time in a C busy-wait loop. You may not use the delay loops from the avr-libc.
- When you initialize the timer, it is best to first select the mode, then set the registers (like OCR or whatever you need), and last start the timer.

Questions

1. How long is the signal really high and low in each of these methods and why? How do you compute the actual high- and low-times? Verify your calculations with the oscilloscope.
2. Compare the four methods with respect to their accuracy, their use of hardware resources, and their processor load.
3. What are the shortest periods supported by methods (3) and (4)? What happens if the periods become shorter than these bounds?
4. If you connected a LED to PD5, what effect would you observe (as opposed to simply setting PD5 to high)?
5. Do you have to observe any order when accessing the 16 bit registers in normal and CTC mode?

Pitfalls

- Do not forget to set PD5 to output (see ATmega16 manual, p. 100).
- Note that the timer counts from 0 to (including!) TOP (or OCR1A) before raising an interrupt, so the interval is TOP+1 ticks. This is easily overlooked and does not make much difference for a small prescaler, but makes a huge difference for a large prescaler.

References

- ATmega16 manual, Timer 1: p. 89 f.
- ATmega16 manual, Timer 1, Timer Modes: p. 101–110

5.4 PWM Signals and Glitches (DL2)

Task

Write a program that demonstrates what kinds of glitches can occur in each of the Fast PWM, Phase Correct PWM, and Phase and Frequency Correct PWM modes. Do do so, write three programs, one for each PWM mode. In them, generate a periodic signal on PD5 that is initially high for 50 μs and low for 25 μs. Then alter the compare and/or top values appropriately to produce all glitches (changes in the period, high/low time, or symmetry of the signal that neither correspond to the old PWM signal nor to the new one) possible in the given mode.

Remarks

- Since this program should demonstrate the glitches, they should be easily and reliably visible on the oscilloscope.

Questions

1. What kinds of glitches can occur in each of the modes?
2. What did you have to do to reliably produce these glitches in your programs?

6 Analog Module

6.1 Analog Comparator (DL1)

Pin Assignment

Simple I/O board	
PHOTO1	AIN0
TRIM1	AIN1
LED4	PC4
CC_LEDS	VCC (+)

Task

Use the analog comparator to turn on LED4 whenever the voltage of the photo transistor exceeds the threshold voltage provided by TRIM1, and turn off the LED when the voltage falls to or below the threshold. Use an ISR to set the LED and put the processor into an appropriate sleep mode in the main loop.

Questions

1. Do you have to enable the pull-up for PHOTO1?
2. What happens to the output voltage of the photo transistor when it grows darker?
3. Is it sufficient to enter the sleep mode once before executing an empty main loop, or do you have to execute the sleep instruction in the main loop? Explain your answer.
4. Explain what you have to do to let the analog comparator trigger an input capture event.

Hints

- You do not need the ADC multiplexer.
- In the ACSR, only 3 bits are important for this exercise.

References

- ATmega16 manual, analog comparator: p. 202–204, register ACSR
- ATmega16 manual, sleep modes: p. 32–34

6.2 Analog Conversion (DL1)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
TRIM2	ADC0

Task

Use the a/d converter to determine the voltage level of ADC0. Configure the converter for maximum resolution and set the voltage reference to AVCC. Display the highest 8 bits of the conversion result on the LEDs. Use an ISR to react to the conversion complete event and to display the result.

Remarks

- The full description of the module is not important for the exercise. Just read the introductory text on pages 202–206 and the register description from page 215 on. After you have completed the exercise, read the rest of the module description.

Questions

1. Do you have to enable the pull-up for TRIM2?
2. To which value do you have to set the prescaler and why?
3. If you assume that the ADC has the conversion range [0, 5] V, then what is the granularity of the AD converter (i.e., the voltage value of the lsb of the result)? How do you compute this value (give a formula for it)?
4. Assuming no noise, is the lsb (the least significant bit) of the conversion result always correct? Explain why or why not.
5. Assume that for a constant trimmer position the LEDs representing the lowest 4 bits of the conversion result flicker. Can you assume that these four bits are noisy?

Pitfalls

- Do not forget to set the reference voltage to AVCC with external capacitor at AREF, and to set the prescaler so that you get maximum resolution (see page 205).
- When you read the ADCL register, the controller does not update the ADC data register until you read ADCH.

Hints

- To make things easy, use the single conversion mode and ignore the auto trigger function. You might also want to use the left-justified mode.

References

- ATmega16 manual, analog converter: p. 205–222, registers ADMUX, ADCSRA, ADCL, ADCH

6.3 Noise During Analog Conversion (DL2)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
TRIM2	ADC0
SW2:1	PA5:4
SWCOM_1-3	GND (⊥)

Task

Connect TRIM2 to pin ADC0 of the controller and use the ADC to determine the voltage level of ADC0. Set the prescaler so that you get maximum resolution. Display the lowest 8 bits of the conversion result on the LEDs. Now we want to see how noise affects the conversion result. Use SW2:1 to select different operating modes which control what you do in the main loop:

SW2	SW1	operating mode (action taken in main loop)
0	0	do nothing (except check SW2:1)
0	1	enter noise cancellation mode
1	0	set PA1 to digital out and toggle its value in each iteration
1	1	set PA1 to digital in, connect it to PD5, and toggle PD5 in each iteration

Remarks

- Use the single conversion mode.

Questions

1. Why do we now display the lowest 8 bit of the conversion result, not the highest 8 bit as in Exercise 6.2?
2. Monitor the analog signal on the oscilloscope. How do the modes affect the signal?
3. How do the operating modes affect the conversion result? How many bits are affected in each of the modes?
4. Does the analog value itself influence the accuracy (i.e., can simply changing the trimmer to another value produce a different accuracy)?

Pitfalls

- When you read the ADCL register, the controller does not update the ADC data register until you read ADCH.

6.4 Prescaler and Accuracy (DL2)

Based on: 9.5

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2
TRIM2	ADC0
SW3:1	PA6:4
SWCOM_1-3	GND ($\perp$)

Task

Use the a/d converter to read the value from TRIM2. Read every value twice: First trigger a conversion after you have configured the converter to the highest prescaler, then set the prescaler to the setting specified by the DIP switches and trigger a second conversion. Display the result of the first conversion as a hexadecimal value on the leftmost three digits of the numeric display, the result of the second conversion on the rightmost three digits. Activate the dot point on DIGIT3 to separate the two values.

Questions

1. What influence does the prescaler setting have on the result? What behavior do you observe? Do a nice presentation (table or graphical) of your findings.

Pitfalls

- When you read the low byte of the conversion result, it remains frozen until you read the high byte.

7 Interfaces

7.1 UART (DL2)

Pin Assignment

Simple I/O board	ATmega16 controller board
SEG_DP:SEG_A	PC7:0
CC_DIGIT0	VCC (+)
ROW3:0	PB3:0
COL3:0	PA7:4

Task

Program the send unit of the USART module (asynchronous mode, 8 bit, 1 stop bit) of the ATmega16 to send every character you enter on the keypad of the I/O board to the PC (over the USB connection), where it should be displayed in a terminal (minicom). It is your choice whether to use parity or not. Furthermore, program the receive unit of the USART module to receive characters from the PC and display them on the numeric display of the I/O board (assuming the character received is in {0..9, A..F}; use □ (segments c,d,e,g) to display all other characters).

The resulting program should display every character entered on the PC keyboard on the ATmega16 display and vice versa.

Remarks

- Use the keypad mapping of exercise 9.7.
- Use interrupts for the USART, not polling.
- To make your code versatile, it might be wise to implement functions for sending characters and whole strings. To test them, you can for example send a welcome message at the start of the program.
- Read the whole section on the USART in the manual before you begin to program. Use a state diagram to visualize how the asynchronous mode works, what registers you should set at which stages, and which bits are set/which interrupts are raised by the controller.

Questions

1. (a) What is the difference between a UART and a USART? (b) Come up with some advantages for each of synchronous and asynchronous mode.
2. (a) Which pin is used for the clock signal in synchronous mode? (b) Why do you not need it in asynchronous mode? (c) Who generates the clock signal in synchronous mode? (d) Where does the clock come from in asynchronous mode?
3. Tables 68-71 of the ATmega16 manual tell you how to set the UBRR to use a given baud rate.
   (a) How is the UBRR used to generate the baud rate?
   (b) What does the error column in the tables signify and how is it computed?
   (c) Why does a clock of 11.0592 MHz not produce any errors for baud rates between

2400 and 230.4k, but a clock of 16 MHz does?
(d) Why is 1 Mbps not available for the 11.0592 MHz clock?

4.

5. (a) Experiment with the baud rate: Which baud rates can you use in your program to communicate (error-free) with the PC? (b) Choose a suitable baud rate for your program and justify your choice.

Pitfalls

- Do not forget to set `minicom` to the communication settings you have selected.
- Do not forget to use SIGNAL() for RXC and UDRE interrupts. Since they are not set back by the call to the ISR, using INTERRUPT() will generate a stack overflow!
- Do not forget you have to set the URSEL bit when writing to UCSRC. Note that you cannot simply write `UCSRC |= ...` because there are special considerations when reading UCSRC.

References

- ATmega16 manual, USART module

7.2 SPI (DL1)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT0	VCC (+)
SW1	PA4
SWCOM_1-3	GND ($\perp$)

Task

Write a program that increments a counter whenever a keypad button is pressed and that sends the new counter value over the SPI interface. Simultaneously, your program should read from the SPI interface and display the value it receives as a decimal value on the numeric display. With SW1, you can select whether your target should be the master (SW1 is on) or not (SW1 is off).

To test your program, connect two targets and download your program on both of them. On each target, you should see the number of times a button was pressed on the other target's keypad.

Remarks

- The SPI pins are used by the serial programmer, so if you use the serial interface for programmning, you have to disconnect the programmer cable from the board after downloading your program. Then press the RESET button to restart your program.
- Think of a way to get data from the slave to the master without unacceptable delays.

Questions

1. How efficient is the communication (ratio of data bits to all transmitted bits)?
2. Can the slave initiate a communication?
3. Can you set both communication partners to master?
4. What do you have to do to detect transmission errors (bit flips)?

7.3 TWI (IIC) (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:seg_C	PC7:2
SEG_B:SEG_A	PA3:2
CC_DIGIT0	VCC (+)
SW1	PA4
SWCOM_1-3	GND ($\perp$)

Task

Write a program that increments a counter whenever a keypad button is pressed and that sends the new counter value over the TWI interface. If your program receives a value over the TWI interface, it should display it as a decimal value on the numeric display. With SW1, you can select whether your target should be the master (SW1 is on) or not (SW1 is off).

To test your program, connect two targets and download your program on both of them. On each target, you should see the number of times a button was pressed on the other target's keypad.

Remarks

- Select appropriate addresses.

Questions

1. Why do you need the pull-up resistors on the TWI lines?
2. What is your bit rate?
3. How efficient is the communication (ratio of data bits to all transmitted bits)?
4. Can you set both communication partners to master?
5. What do you have to do to detect transmission errors (bit flips)?

Pitfalls

- The TWI bus needs external pull-up resistors. Ask a tutor for a breadboard. Connect the row next to the red line to VCC (+), and the other end of the resistors to SCL and SDA.

8 Controller Misc

8.1 Watchdog Timer (DL1)

Pin Assignment

Simple I/O board	
LED0	PC0
CC_LEDS	VCC (+)
SW1	PA4
SWCOM_1-3	GND (⊥)

Task

Show that the watchdog timer works: Arm the watchdog and use SW1 to indicate whether the timer should be periodically reset by your program (SW1 is ON) or not (SW1 is OFF). Use LED0 to show that the watchdog timer works, and integrate the Watchdog Reset Flag into your solution. Use an appropriate timeout period.

Remarks

- Forget about power consumption issues in this exercise and just poll SW1 in an infinite loop.

Questions

1. Name two situations in which a watchdog is useful.
2. Discuss how the watchdog helped or did not help in the 1997 Mars Pathfinder mission.
3. How can you disable the watchdog? Why is the procedure so complicated?
4. How can you determine at the start of your program whether the user has pressed the RESET button?

Pitfalls

- Do you need the internal pull-up for SW1?
- Be aware that the Watchdog Reset Flag is only cleared by a power-on reset, not by an external reset via the reset button.

Hints

- To be able to verify that the watchdog works, you have to do something at the start of your program, like turning on a LED for some time.

8.2 Pipelining (DL1)

Pin Assignment

Simple I/O board	
LED0	PC0
CC_LEDS	VCC (+)

Task

On page 13 of the ATmega16 manual, it is stated that the AVR CPU pipelines instruction fetches and executions to obtain up to 1 MIPS per MHz. Demonstrate or refute this claim using LED0 and the oscilloscope.

Questions

1. What is the idea behind your program?
2. What did you find out?
3. Are the cycle numbers given in the Instruction Set Summary of the ATmega16 manual with pipelining or without?

8.3 Memory Considerations (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT0	VCC (+)

Task

Display the numbers 0...9 one by one for an appropriate time on the numeric display (start again with 0 after you are done). Use a table to store the values for the numbers.

Now create three versions of your program which differ in the location of the table in memory: Your first program should have the table in SRAM, your second in the Flash, and your third in the EEPROM.

Remarks

- To make things easy, use a busy-wait loop to implement the delay.
- Remember that the SRAM cannot be initialized automatically, you have to initialize the table within your program.
- Note that the controller gets halted by access to the EEPROM.

Questions

1. Compare the three programs with respect to ease of setting up the table (definition and initialization) and of accessing it in the program.
2. How fast (in system clock cycles) can you read a table entry if it is in SRAM? How fast if it is in the Flash? How fast if it is in the EEPROM? Include everything that belongs to the read operation, like writing to the address registers in case of the EEPROM.
3. How fast can you write a table entry in SRAM? How fast in EEPROM? Why do we not ask about Flash writes?
4. For how long did you display each number? Could you display them for exactly one second? If yes, then demonstrate how you would do it, if not, explain why not.

References

- ATmega16 manual, Memories: p. 16–23

9 Hardware

9.1 Hooking Up the Hardware (DL0)

Task

This will most likely already have been done for you by the teaching staff. But if you ever find the Simple I/O board completely disconnected, here is how to get power to it.

1. Identify the ground connectors on the ATmega16 controller board.
2. Identify the ground connectors on the Simple I/O board.
3. Use a wire to connect a ground connector pin on the ATmega16 controller board to one on the Simple I/O board (use the dedicated power supply ports; on the Simple I/O board, the power supply port also contains TRIM1 and some other hardware pins).
4. Identify the VCC (+) connectors on the ATmega16 controller board.
5. Identify the VCC (+) connectors on the Simple I/O board.
6. Use a wire to connect a VCC (+) connector pin on the ATmega16 controller board to one on the Simple I/O board.

Please do not mix anything up on this, you could short-circuit the boards if you are not careful. If you are not sure, ask for help! Please do not use the PWR-pins, always use + (VCC).

References

- Simple I/O board
- ATmega16 controller board

9.2 Getting to Know the Hardware (DL0)

Task

Familiarize yourself with the ATmega16 controller board and the Simple I/O board, with their components and their pinouts. You can find the board descriptions and pinouts on the course homepage. Go through the components and pins on both boards and make sure you understand what they are there for. On the Simple I/O board, make sure that for each hardware component you understand where the appropriate pin(s) is (are) located on the connectors.

Remarks

- Note that both the LEDs (LED7:0) and the numeric display segments (SEG_DP:SEG_A) are connected to the connectors labeled (P,G:A). That is, LED0 = SEG_A = A on the Simple I/O board.

References

- Simple I/O board
- ATmega16 controller board

9.3 Oscilloscope (DL0)

Based on: 1.1

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)

Task

Connect the probes of the TD220 oscilloscope to LED1 and LED0. Make sure you do not create short-circuits. Download the Assembler demo program of Section 1.1 on the target and start it. Do the following things:

- Display the two signals on the oscilloscope.
- Experiment with the resolution (voltage and time).
- Find out how to freeze the picture.
- Shift the signal horizontally and vertically, zoom in and out of it.
- Measure the time (and frequency) between two edges of the signal from channel A. Use the cursors for this purpose.
- Experiment with the trigger and find out how to change the trigger from channel A to channel B. Check out the trigger level.
- Let the oscilloscope trigger on edges (rising, falling).
- Configure the oscilloscope so that it detect peaks in the signal.

Remarks

- Do not change settings you do not understand. If you manage to completely mess up the oscilloscope settings and do not know how to set it back to useful values, set it back to its factory defaults.
- If you are not sure the oscilloscope is set right, connect the probe to the output called "Tastkopfabgleich". It generates a square wave signal of 5V with 1kHz amplitude. Use this signal to get the oscilloscope settings right. If all else fails, set it back to its factory defaults.

Pitfalls

- The probe has a slider that selects 1x or 10x attenuation. The slider should be at 1x. The oscilloscope has a corresponding setting, which should also be at 1x.

9.4 Button Debouncing (DL1)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_DIGIT0	VCC (+)
BTN8	INT1

Task

Write an ISR that increments a counter (modulo 10) whenever the button is pressed and that displays this counter value on digit 0 of the numeric display. Take counter-measures to prevent bouncing.

Remarks

- Bouncing is a side-effect and unpredictable. It may happen that you have a board where BTN8 does not (yet) bounce. In that case, try one of BTN1-6 (connect the common pin to GND). Chances are good that you will quickly find a button that does bounce. For this exercise, use the button that bounces most.

Questions

1. What is the difference between BTN7 and BTN8? What happens if you try out your program with BTN7? Explain why.
2. Examine the bouncing signal with the oscilloscope and try to capture bouncing on the scope. How long does it generally take until the signal is stable? What is the maximum bounce time you observed? How often does the signal bounce before it stabilizes? Does it only bounce when pressed, or when released, or in both situations?
3. Describe the debouncing strategy you have implemented. How does it work? Which assumptions, if any, does it make?
4. List at least two other (significantly different) methods to debounce the button and explain how they work. Compare them to your method in terms of ease of use, coverage, and processor load.

Pitfalls

- If your solution disables the INT1 interrupt for some time, do not forget to clear the interrupt flag before enabling the interrupt again.

9.5 Display Multiplexing (DL1)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2

Task

Display the last 6 places of your Matrikelnummer on the numeric display. Use a display frequency between 80–100 Hz. Document which frequency you used.

Remarks

- To make the program versatile (you will need it again in later exercises), use an array to store what should be displayed on the digits.
- Write your program in such a way that the number of digits of the display can be changed easily. Changing the number of digits should not result in a different display frequency.

Questions

1. Why do you have to multiplex the display? How many I/O lines would you need to control all digits simultaneously? How many I/O lines does our board require?
2. Is there a way to reduce the number of necessary I/O lines even more (hardware solution)?
3. Does multiplexing the display have any disadvantages? On what does the brightness of the display depend?
4. Which timer mode did you use for multiplexing and why? Which display frequency did you use?

Pitfalls

- The display period is the time it takes to iterate through all digits once. So if the display period is 6 ms, then you have to change to a new digit every 1 ms (we have six digits). The display frequency is the reciprocal of the display period.

Hints

- Since the digits share the segment pins, you have to multiplex the display.

References

- Script, Chapter 5.8

9.6 Display Multiplexing Frequency (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2
TRIM2	ADC0

Task

Write a program that allows to experiment with different display periods. To this end, read the high byte of TRIM2 and use this value to compute the display period P = 10 + 0.2 * (value of the high byte), given in ms. Show the display frequency in Hz and the display period in ms with one place after the comma on the numeric display as "`ff pp.p`".

Remarks

- Treat the special case $f = 100$ Hz as you deem appropriate.
- You do not need (and should not use) floating point operations.

Questions

1. Experiment with the display period. Theoretically, which period should suffice to get a stable display (no flicker) and why? Try out this theoretical value. Would you recommend it, or would you prefer some other value (which one)?
2. What happens if the multiplexing period gets very small or very large (even beyond the bounds set by this program)? Does this have any effects?
3. Is it better to use the free-running mode of the ADC, or is it sufficient to read the last conversion result in response to a display interrupt and to start a new conversion afterwards? Compare the two methods. Which one did you use?

9.7 Matrix Keypad (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT0	VCC (+)
ROW3:0	PB3:0
COL3:0	PA7:4

Task

Read the 4×4 keypad, display each new entry on DIGIT0 of the numeric display. Set the COLs to input, and alternately select a ROW. Interpret the keypad in the following way:

	COL0	COL1	COL2	COL3
ROW0	0	1	2	3
ROW1	4	5	6	7
ROW2	8	9	A	B
ROW3	C	D	E	F

Remarks

- Only react once to each button while it is pressed on the keypad (i.e., no auto-repeat). If multiple buttons are pressed at the same time, react to each of them (but also only once).
- When you recognize that a key is pressed, you should still react to other keys.
- Use a timer to generate the keypad scan period.
- Put the processor into an appropriate sleep mode in the main loop.
- Only set the pin of the selected ROW to output, set the other ROW pins to input with pull-up.

Questions

1. Why will the COL bit of a pressed button be 0 if the ROW is set to 0?
2. Why is it not a good idea to set the ROW and immediately read the column states?
3. How many buttons can be pressed simultaneously and still be recognized without error (your answer should be true for any configuration)?
4. Why should you set unselected rows to input, why not just set them to output and 1? (Hint: think about what happens to ROW1 when you select ROW0 and press keys 2 and 5 simultaneously.)

Pitfalls

- Do not forget the pull resistors for the COLs.

Hints

- The keypad is generally polled one row at a time. The idea is to select a row by setting its ROW bit to 0. If a button is pressed in this row, the corresponding COL bit will read 0 as well. Since the keypad is operated by a human, we are under no tight time constraints here. Scanning a new row every 10 ms or so should be quite sufficient. If you already have an appropriate timer interrupt in use (e.g., for the display), you can use this interrupt to poll the keypad as well.
- Note that you do not have to be concerned with bouncing since you are not using interrupts.

References

- Script, Chapter 5.2

9.8 DC Motor (DL1)

Pin Assignment

L293 board	
Input 1	OC1A
Input 2	PD7
Simple I/O board	
EN	your choice
TRIM2	ADC0
SW2:1	PA5:4
SWCOM_1-3	GND (⊥)

Task

Write a program that controls the dc motor. Use a PWM mode to control the motor. The program should allow speed control with TRIM2 (zero to full speed) and direction control with SW1. If SW2 is in position OFF, then the motor should be disabled. Use a PWM frequency within 100 Hz - 1 kHz.

Remarks

- The trimmer value should be proportional to the PWM ratio.

Questions

1. Assume that you have a PWM signal on Input 1, and a constant level on Input 2. If you revert the direction by inverting Input 2, how do you have to change the signal on Input 1 to maintain the same speed?
2. At which PWM ratio does the motor start to turn? Does the PWM frequency influence this value? (Present your observations concerning the influence of the PWM frequency in a table or graph.)
3. How did you find out at which ratio the motor starts to turn?

References

- Motor board description

9.9 Stepper Motor with Driver IC (DL1)

Pin Assignment

UCN5804 board	
STEP	OC1A
DIR	PD7
Simple I/O board	
TRIM2	ADC0
SW1	PA4
SWCOM_1-3	GND (⊥)

Task

Write a program that controls the stepper motor which is connected to the UCN5804 stepper motor driver. Use a CTC or PWM mode to control the motor. The program should allow (linear) speed control with TRIM2 (zero to full speed). SW1 should select a direction.

Remarks

- Check the datasheet of the driver to find out its maximum allowable speed. The motor speed must be determined experimentally.
- You need to connect several more pins of the driver IC to make it work. Refer to the datasheet.

Questions

1. What is the maximum allowable speed for (a) the microcontroller, (b) the driver, and (c) the motor?
2. In half-step mode, is the maximum motor speed the same as in normal mode? Explain your observations.
3. Give three examples where using a stepper motor is better than using a dc motor.

Pitfalls

- You need to select a stepping mode (one phase, two phase, or half-step).

References

- UCN5804 datasheet

9.10 Direct Access to Stepper Motor (DL2)

Pin Assignment

Simple I/O board	
Phase A	PA0
Phase B	PA1
Phase C	PA2
Phase D	PA3
SW2:1	PA5:4
BTN1	PB4
BTNCOM_1-3	GND ($\perp$)

Task

Write a program that directly controls the phases of the stepper motor. To this end, use the stepper motor that is connected to the L293D driver and make the motor move with slow speed. Allow the user of your program to determine the direction of rotation with SW1 and to enter/leave half-step mode with SW2. BTN1 should start/stop the motor (debounce the button).

Questions

1. Which pattern did you use to control the phases throughout one rotation?
2. What are the highest possible speeds achievable by the stepper motor, by the driver, and by your software? Who can achieve the highest speed? Who determines the maximum speed of this application?
3. Is there a bound for the minimum speed?

10 Applications

10.1 Memory Game (DL2)

Pin Assignment

Simple I/O board	
LED3:0	PC3:0
CC_LEDS	VCC (+)
BTN3:1	PA6:4
BTNCOM_1-3	GND ($\perp$)
BTN7	PA7

Task

This game plays a light sequence (one LED at a time) on the LEDs which the user has to repeat on the buttons. If the player is able to repeat the sequence flawlessly, a new sequence is displayed which is longer than the last one by one LED, until the player fails to repeat the sequence correctly.

To implement the game, start with a sequence length of $k = 3$. When displaying the sequence, there should be a pause between turning off one LED and turning on the next LED to allow the same LED to be repeated. Now repeatedly randomly choose a sequence of length k, display it, then compare it to the sequence that the user enters. If the sequences match, increment k. Otherwise, the game is over.

Remarks

- Feel free to enhance this game with a high-score, two-player mode, or any other features you would like to have.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.2 Garden Light (DL2)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
BTN7	INT0
Photo1	ADC1

Task

Implement a garden light that goes on automatically when it gets dark. The application should turn on half of the lights at dusk and turn on all lights when it is dark. The ambivalent light level is sensed by the light sensor PHOTO1. The two thresholds which are used to control the lights should be programmable.

To implement the garden light, keep two light thresholds TH1 and TH2. As long as it is brighter than TH1, turn off all LEDs. When the ambivalent light is between TH1 and TH2, turn on LEDs 0, 2, 4, and 6. When it gets darker than TH2, turn on all LEDs.

If the user presses BTN7, enter a programming mode in which LED0 blinks (with about 1 Hz). This indicates that TH1 can now be programmed. When the user presses BTN7 again, take the current value read from the light sensor and store it as TH1. Then blink LED1 to indicate that you are ready to take over the second threshold. When the user presses BTN7 the next time, take the value of the light sensor over as TH2 and leave programming mode.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.3 Safe Lock (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT2:0	PD4:2
TRIM2	ADC0

Task

Program a safe lock. Three 2-digit numbers (#define) are the safe combination. Use Trim2 to unlock the safe.

To implement the safe, start with a locked safe. The display shows “- -” (safe locked) waiting for action. To start input turn Trim2 to its end positions within 500 ms. Now the display shows “xx” (xx $\in$ [00-99] is depending on Trim2). If the display value does not change during 1 s it will be taken over. The display changes to ”kxx”. The first digit k shows how many numbers are locked (activate 1, 2 or 3 horizontal segments) Now you turn the trimmer and enter the second number by waiting. After the third the display changes to “...” if the code was correct or to “0 - -” if the unlock failed. After 3 seconds the lock changes to waiting mode “- -”. Don’t forget to lock again.

Define the time constant (1 s) global and find good values for a safe and quick unlock procedure.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.4 Scrolling Text (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2
ROW3:0	PB3:0
COL3:0	PA7:4
BTN1	PB4
BTNCOM_1-3	GND ($\perp$)

Keypad Assignment

Keypad	COL0	COL1	COL2	COL3
ROW0	1	2	3	n/a
ROW1	4	5	6	n/a
ROW2	7	8	9	n/a
ROW3	0	$\leftarrow$	$\rightarrow$	$\hookleftarrow$

Task

At the start of the program, let the user enter a number (with maximum length TEXTLEN, set it to 20 in your program). Each new digit entered by the user should appear on DIGIT0, the other digits should move to the left. At any time, the user can scroll through the whole entry with the $\leftarrow$ and $\rightarrow$ keys. Whenever the user adds another digit to the entry, the program should again display the six least significant digits.

If the user presses $\hookleftarrow$, interpret the entry as complete. The user can then either scroll the number, or start a new entry by pressing 0-9. Clear the display before showing the new entry.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.5 Distance Sensor (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT2:0	PD4:2
BTN1	PB4
BTNCOM_1-3	GND ($\perp$)
PHOTO1	ADC1

Task

Our hardware does not include a distance sensor. However, when you hear someone propose to use the photo transistor for this purpose, arguing that as long as an object casts more shadow on the sensor the nearer it comes, one can use this fact to compute the distance, you are intrigued and decide to implement this idea to check its merits.

Implement a program that computes the distance of an object from the output of the photo transistor. Display the distance in cm on the numeric display. To compute the distance, use a mapping table with about 5-10 points and interpolate between these points. Include a calibration mode in your program to set up the mapping table. For each point in the table, the program should display its distance on the numeric display (activate the dot on DIGIT0 to indicate that this is the calibration mode). Then wait until the user presses BTN1 and store the current value of the photo transistor in the table. Iterate through all points in the table in this way. To enter calibration mode, the user should press BTN1. The mode is left after the last point has been entered.

Remarks

- We are aware that it is not so easy to get a good distance sensor out of our photo transistor, but we rely on your ingenuity to make this work (at least under some special conditions, for a demonstration to your friends and the tutor).
- You can decide for yourself which range would be best for the sensor. It should be at least a couple of centimeters (10 cm or more).

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. How well does the idea work? Do you have any suggestions how to make it work better?

10.6 Frequency Measurement (DL2)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT2:0	PD4:2
PHOTO1	AIN0
TRIM1	AIN1
SW2:1	PA5:4
SWCOM_1-3	GND ($\perp$)

Task

Put TRIM1 to its center position. Now measure the frequency with which the photo transistor is darkened. To achieve this, let timer 1 run freely and let the analog comparator trigger an input capture interrupt (you can choose whether to trigger an interrupt on toggle or on only one edge). In the input capture ISR, compute the time since the last capture interrupt and determine the frequency. Display the frequency on the numeric display.

Allow the user to select a display unit of s, ms, and μs with SW2:1. Within a unit, display frequencies within 0-9 as 0.00-9.99, frequencies within 10-99 as 10.0-99.9, and frequencies within 100-999 as 100-999. Use round-to-nearest to determine the value you should display.

Remarks

- Depending on the ambient light, you may have to set TRIM1 to some other than its center position to get good results.
- Use averaging to get nice results

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. Do you need to activate noise cancellation for the input capture? In which situations do you need noise cancellation?

10.7 Capacitor Measurement (DL3)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
BTN1	PB4
BTNCOM_1-3	GND ($\perp$)

Task

Find a method to identify the capacitor with the largest capacitance from a given set of $k \leq 8$ capacitors, and implement your solution on the microcontroller. Use the LEDs and BTN1 for user interaction.

Remarks

- Ask your tutor for capacitors to try out your solution.
- You can test your application with capacitors of the same value (serial, parallel or single connection).
- If you like, you can enhance your program to calculate the capacitance and to display it (or to send it to the PC by UART).

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. What electronic laws and formulas do you require?
5. What limits (min, max capacitance) does your method have?
6. What accuracy can you guarantee?

10.8 Memory Dump (DL3)

Task

Write a debugging routine that allows you to dump memory and register values and the contents of the stack to a computer terminal via UART. Call your routine from the receive UART ISR.

Test your routine with an application.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. Is Assembler or C better for this task?
5. What are the limits of your routine? Where is it useful?
6. Is a periodic routine execution (called by a timer ISR) better than a sporadic one (external signal, UART)? Why or why not?
7. Can you implement single-step debugging via UART monitoring? What limits are given?

10.9 Dynamic Memory Usage (DL3)

Task

Write a program that determines the dynamic memory (stack) usage of an application program. Test your routine with an application.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. Is your implementation always accurate? What are its limits?

10.10 Calculator (DL3)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2
ROW3:0	PB3:0
COL3:0	PA7:4
BTN1	PB4
BTNCOM_1-3	GND ($\perp$)

Keypad Assignment

Keypad	COL0	COL1	COL2	COL3
ROW0	1	2	3	$+$
ROW1	4	5	6	$-$
ROW2	7	8	9	$\times$
ROW3	$+/-$	0	=	$\div$

Task

Implement a simple calculator that can do the operations $+$, $-$, $\times$, and $\div$ on integers. The calculator should accept and display integers in the range ± 99999, where the leftmost digit is used to display the (negative) sign. If an overflow occurs, display "O" on the leftmost digit and clear all other digits. For errors (e.g., division by zero), display "E" on the leftmost digit and clear all other digits. Use BTN1 to reset the calculator.

Remarks

- You have to observe operand priority (i.e., $2 + 3 \times 4$ should result in 14, not in 20).

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.11 Video recorder forward/rewind (DL3)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT2:0	PD4:2
BTN1	PB4
BTNCOM_1-3	GND ($\perp$)
TRIM2	ADC0
Input 1	OC1A
Input 2	PD7

Task

Implement the fast forward/rewind functionality of a video recorder. To this aim, use the dc motor to move the tape and use the signal from one of the photo interrupters as position information. Display the current position of the tape on the numeric display (0-999). Do not allow any overflows (stop the motor at the end positions).

The tape speed is controlled by TRIM2: If TRIM2 is in the middle, interpret this as zero speed. If TRIM2 is turned to the left, rewind the tape. If it is turned to the right, fast forward the tape. In either case, the speed should be proportional to the position of the trimmer.

Use button BTN1 to reset the position counter to 0.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.12 Stepper Motor (DL3)

Pin Assignment

Simple I/O board	
LED7:0	PC7:0
CC_LEDS	VCC (+)
STEP	OC1A
DIR	PD7
PHA	ICP1

Task

Find the frequency f at which the stepper motor begins to lose steps. To this aim, start with a low stepping frequency and start the motor. Use the photo interrupters to count the number of steps the motor actually executes. After the second rotation, verify whether the motor executed all steps. If it did, stop the motor, increase the frequency and try again. Stop after you have observed a loss of more than two steps (it is theoretically possible to lose two steps through inaccuracies of our hardware setup) and display the responsible frequency on the numeric display.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. Briefly explain the differences between the three operating modes normal operation, one phase, and half-step (take a look at the timing diagram on the third page of the driver's datasheet).
5. Perform measurements for each of the operating modes of the stepper motor. Include the results in your protocol and interpret them (individually and in relation to each other).
6. Does the direction influence your results?
7. Which components pose limits on the stepper motor speed? What are these limits?

10.13 Motor Speed (DL3)

Based on: 9.8

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT2:0	PD4:2
Input 1	OC1A
Input 2	PD7
PHA	ICP1
TRIM2	ADC0
SW1	PA4
SWCOM_1-3	GND ($\perp$)

Task

Write a program that controls the dc motor via PWM. The program should allow (linear) speed control with TRIM2 (zero to full speed) and direction control with SW1. Use the photo interrupter to determine the motor's speed, and show the speed in rotations/second on the numeric display.

Remarks

- Make sure your input capture ISR can handle slow speeds (i.e., one or more timer overflows occur before the ICP ISR is called).

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).

10.14 Chipcard Access (DL4)

Pin Assignment

SW_CHIP	PB4
SDA_CHIP	SDA
SCL_CHIP	SCL
LED7:2	PC7:2
LED1:0	PA3:2
CC_DIGIT2:0	PD4:2
ROW3:0	PB3:0
COL3:0	PA7:4

Keypad Assignment

Keypad	COL0	COL1	COL2	COL3
ROW0	1	2	3	n/a
ROW1	4	5	6	n/a
ROW2	7	8	9	A
ROW3	–	0	n/a	E

Task

Program a cash reload/withdrawal station which has the following stages:

1. Check whether there is a chipcard inserted into the cardreader or not. If there is, display "CIS" (card in slot) and wait for the user to remove the card.
2. If there is no card in the reader, let the user enter a decimal number A within $[-199, 199]$ and display it ('–' toggles the sign). The entry is concluded by pressing 'E'.
3. Display "ICC" (insert chip card) and wait for the user to insert the chipcard.
4. After the user has inserted the chipcard into the reader, read the current cash balance C (within $[0, 999]$) and display it on the numeric display for 2 seconds.
5. If $C - A \notin [0, 999]$, display "AEL" (amount exceeds limits) for 2 seconds and go back to step 1.
6. Display the new amount $C - A$ and wait for the user to confirm the transaction by pressing 'E' or to abort it with 'A'.
7. If the user has pressed 'A', return to step 1.
8. If the user has pressed 'E', store the new amount $C - A$ on the chipcard. Display "PC" (process complete) for 2 seconds after you are done. Then return to step 1.

If the user inserts the chipcard at any step where it should not be in the reader, or removes the card without being instructed to do so, abort the transaction and return to step 1.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. Evaluate the system with respect to security. In what ways can an adversary attack it? Think of counter-measures for every type of attack, or explain why the system in its current configuration cannot be protected. If you could ask for different and/or additional hardware, what would you need to make the system secure?

10.15 Characteristic Curve of DC Motor (DL5)

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT2:0	PD4:2
Input 1	OC1A
Input 2	PD7
PHA	ICP1

Task

Write a program which obtains the speed characteristics of the dc motor for different PWM duty cycles. To this aim, display the active duty ratio (high time in percent of the period) on DIGIT2:0 of the display. Display "- - -" while there is no measurement in progress. Let the user start a new test with BTN7.

During a measurement, do the following for a set of duty ratios $\mathcal{R} = \{R_1, ..., R_n\}$ of the PWM signal:

1. Start the motor with duty ratio R_i.
2. For each encoder revolution k ($k \geq 1$), use the input capture feature to measure the time $t_{rev}^{(k)}$ one full encoder revolution takes. Set $t_{rev}^{(0)} = 0$.
3. From $t_{rev}^{(k)}$, compute the current motor speed $v_{mot}^{(k)}$ in revolutions/second. Set $v_{mot}^{(0)} = 0$.
4. Continue to measure $t_{rev}^{(k)}$ and compute $v_{mot}^{(k)}$ until $|t_{rev}^{(k)} - t_{rev}^{(k-1)}| < \epsilon_i$, that is, until the motor has leveled off at its top speed for duty ratio R_i.
5. Stop the motor.
6. Continue to measure $t_{rev}^{(k)}$ and compute $v_{mot}^{(k)}$ until $|t_{rev}^{(k)} - t_{rev}^{(k-1)}| > T_{i,max}$, that is, until the motor has stopped.

For each i and each measurement k, use the UART to send the data as an ASCII text to the PC in the following form (the data items are separated by spaces, the line should be terminated by a newline):

$t^{(k)}$ R_i $v_{mot}^{(k)}$

where $t^{(k)} = \sum_{j=1}^{k} t_{rev}^{(j)}$, that is, $t^{(k)}$ is the time from the start of the motor until the k-th speed measurement. Send an empty line after the data block of each R_i. At the PC, store the data received from the microcontroller in a file and plot it in `gnuplot` with the following commands:

```
set hidden3d
splot "filename" with lines
```

Your protocol should contain the results of all your decisions and a picture of the gnuplot plot. Include the data file with your measurements in the electronic archive you submit to your tutor.

Remarks

- Before you decide whether to react to the rising or to the falling edge of the photo interrupter signal, take a look at the signal with the oscilloscope; the decision should be clear after that.
- Use constants for all your design decisions so you can easily change their values (including the number of photo interrupter pulses per motor revolution).
- If you find a better or equivalent way to obtain the speed characteristics of the motor, you can implement your solution (but check it with your tutor first). Explain why your solution is better or equivalent in your protocol.
- If you find a better way to represent the collected data, feel free to use your method.

Questions

1. How much static memory does your program use (in Bytes)? How many percent of the available memory does that take up?
2. Estimate how much dynamic memory your program uses. Again, give the number in Bytes and as a percentage of the available memory.
3. Roughly estimate the required processor load (in percent).
4. Interpret your measurements. Pay particular attention to the slow speed range. Use `set view` to rotate the gnuplot view if necessary (`help set view`).
5. What error do you make when computing the motor speed? What components influence the error? Give a formula for the error (use parameters). What measurement strategy should you use to minimize the error?
6. To measure the time for one revolution, you have to add several ICP measurements. What is the worst case error you make in one ICP measurement? When adding these measurements to compute $t_{rev}^{(k)}$, does the error accumulate (explain why or why not)?

Pitfalls

- Since you need the ICR1 register for the input capture, you can only use timer modes 1-3 or 5-7 of Timer 1 for the PWM signal. Remember that the period of a Fast PWM signal lasts for TOP+1 timer clock cycles.

A Demo Programs

This section contains demo programs that you can download from our homepage. The programs generally demonstrate a peculiarity or pitfall, and often you have to find out which behavior was caused by which code. You are welcome to discuss your theories in the Microcontroller forum.

A.1 Enabled/Disabled Interrupts

Pin Assignment

Simple I/O board	
TRIM2	ADC0

Task

The demo consists of four programs. In each program, there are two timer ISRs. ISR1 sets PB0 when it is called and clears it before it finishes. ISR2 does the same with PB1. With TRIM2, you can select an offset between the calls to the ISRs.

The difference between the programs is that one of them disables the interrupts during execution of both ISR, two programs disable interrupts only for one ISR, but allow it for the other, and the last one allows interrupts for both ISR.

Look at the signals with the oscilloscope. Determine for each program whether interrupts are allowed or disabled.

A.2 Multiplexing Displays

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2
SW2:1	PA5:4
SWCOM_1-3	GND ($\perp$)

Task

When changing from one digit to the next, you have to select the new digit and change the value displayed on the segments. Since you cannot do both actions at once, the question arises how to do it correctly. Basically, there are four obvious methods, some of which display side effects:

1. set the new segment value, select the new digit
2. select the new digit, set the new segment value
3. deselect the old digit, set the new segment value, select the new digit
4. clear the old segment value, select the new digit, set the new segment value

The four methods are selected by SW2:1. Whether a digit is computed and set or set and computed is selected by SW3. It is your task to find out which state of the DIP switches belongs to which mode.

Remarks

- Darken the room to see the effects well.

A.3 Keypad Problems

Pin Assignment

Simple I/O board	
SEG_DP:SEG_A	PC7:0
CC_DIGIT5:0	PD7:2
ROW3:0	PB3:0
COL3:0	PA7:4
SW6:4	PA3:1
SWCOM_4-6	GND ($\perp$)

Task

The program reads from the keypad and displays the data on the numeric display. Our implementation sets the ROWs and reads the state of the COLs. Each button that is being pressed is shown (once) on the display and then ignored until it is released again. The key mapping is the same as in Exercise 9.7:

	COL0	COL1	COL2	COL3
ROW0	0	1	2	3
ROW1	4	5	6	7
ROW2	8	9	A	B
ROW3	C	D	E	F

With SW3:1, you can select one of several different ways the keypad is polled:

1. select the next ROW, immediately read from the COLs
2. select the next ROW, do 1 NOP, read from the COLs
3. select the next ROW, do 2 NOPs, read from the COLs
4. select the next ROW, do 3 NOPs, read from the COLs
5. select the next ROW, do 4 NOPs, read from the COLs
6. read from the COLs, select the next ROW (then wait until next period)

Determine which state of the DIP switches selects which mode.

Remarks

- It is possible that you cannot distinguish all states or that the program behaves differently on different boards.
- This program is also a good opportunity to try out the shadow button effect: Press buttons 0, 1, and 5 and see what happens. What can you do to remove this effect (in hardware and/or software)?

B Application Hints

B.1 Application 10.7

- A capacitor has an exponential charging curve that depends on its capacitance C. The higher C, the longer it takes for the capacitor to reach a certain voltage level.
- The analog comparator may be useful.

C Pin Assignment

The following table shows the pin assignments used in the exercises. The left half of the table is the pin description of the controller, which you can find in the ATmega16 manual on page 2.

Pin	Alt1	Alt2	Assign1	Assign2	Assign3
PA0	ADC0		TRIM2		PHASE_A
PA1	ADC1		PHOTO1		PHASE_B
PA2	ADC2			LED0	PHASE_C
PA3	ADC3			LED1	PHASE_D
PA4	ADC4		SW1	COL0	BTN1
PA5	ADC5		SW2	COL1	BTN2
PA6	ADC6		SW3	COL2	BTN3
PA7	ADC7		SW4	COL3	BTN7
PB0	T0	XCK		ROW0	
PB1	T1			ROW1	
PB2	AIN0	INT2	PHOTO1	ROW2	
PB3	AIN1	OC0	TRIM1	ROW3	
PB4	SS		BTN1	SW_CHIP	
PB5	MOSI		(Prog.)		
PB6	MISO		(Prog.)		
PB7	SCK		(Prog.)		
PC0	SCL		LED0/SEG_A	SCL	
PC1	SDA		LED1/SEG_B	SDA	
PC2	TCK		LED2/SEG_C		
PC3	TMS		LED3/SEG_D		
PC4	TDO		LED4/SEG_E		
PC5	TDI		LED5/SEG_F		
PC6	TOSC1		LED6/SEG_G		
PC7	TOSC2		LED7/SEG_DP		
PD0	RXD				
PD1	TXD				
PD2	INT0		CC_DIGIT0	BTN7	
PD3	INT1		CC_DIGIT1	BTN8	
PD4	OC1B		CC_DIGIT2		
PD5	OC1A		CC_DIGIT3	Input 1	STEP
PD6	ICP1		CC_DIGIT4	PHA	
PD7	OC2		CC_DIGIT5	Input 2	DIR