**FYS3240**: Mandatory exercise 3

Human-Machine interface and communication

Part 1

In the first part of this exercise you are supposed to play with a small screen, or display if you like. This is one step up from the General Purpose IO ports of mandatory exercise 2, but in principle just more of the same. But in this exercise you will have to take timing into account. That is, the sequence of port strobing.

The screen is a Liquid Crystal Display [LCD] of the passive or STN variety. This is the same kind found on home stereos, security systems and the like. But unlike the very naked displays mounted in this kind of home appliances, this display will be equipped with a dedicated display(micro)controller and thus, more to be considered a display module. This display controller will be a Hitachi HD44780U or equivalent protocol compatible clones like SANYO LC7985N or Samsung KS0066U.

![Image of a microcontroller board with a LCD display](image.png)

Loads of datasheets can be found [here](https://example.com), whilst this homepage offers a lot of help on the subject. All of these LCD controllers may drive a LCD display at a maximum of 1×80 or 2×40 [lines×characters], limited only by the amount of on-chip **Data Display RAM** of 80 characters. The display controller has built-in support for 1 or 2 lines in its firmware. But there is little stopping the manufacturers of these display modules physically (electronically) wrapping each line of the glass in two effectively producing a 4×20 character module. Even so, these modules will from a software point of view still look like a 2×40 character module. There are also 4×40 character modules on sale — these sport two display controllers with one separate enable [E] line each. Our module is primarily intended for pure text, specifically two lines of 16 characters each. The display controller may scroll text longer than 16 characters (up to a maximum of 40 characters,) each line by its own. The display controller will take care of rendering the text. Our display module comes at a cost of about twice the AVR microcontroller.

What you are supposed to do is feeding the display controller with strings of text. The **SRAM** based data memory of the **ATmega328** is rather limited. You are therefore to refer the strings of text to the **FLASH** based program memory and copy these constant strings from there into data memory as needed, scrapping them afterwards. A problem you will encounter in doing this is the fact that AVR is a so called
Harvard architecture meaning that the data- and program storage is completely separated with separate address spaces. To gain access to program memory you will have to perform specialised "Load Program Memory" instructions in assembly language. It can also be done from within AVR-GCC by means of a set of compiler extensions. These extensions are declared in the headerfile ("#include <avr/pgmspace.h>") Do note that ATmega323 program space is not writeable runtime, with the consequence that any data to be placed there must be placed there compile time. This problem can easily be solved by the following code extract:

```c
#include "HD44780U.h" // User supplied
#include <inttypes.h> // (u)int8/16/32_t
#include <avr/pgmspace.h> // "(E)EPROM" C routines

/*** DEPRECATED: ***/
// const uint8_t progmem */ *sofar;
// const uint8_t_attribute__ ((progmem)) Hello[] = "Hello World!"; // Const string to

PGM_Psofar;
prog_uint8_t Hello[] = "Hello World!"; // Const string to

uint8_t character;

int main(void){
  LCD_init( /* mode flags */);

 sofar = Hello;
  while ((character = pgm_read_byte(sofar)) &&
    (LCD_putc(character) >= 0)) sofar++;
}
```

No use looking for the headerfile "HD44780U.h" of the above example. You are supposed to write that yourself. Alternatively you may include all LCD routines in one and the same sourcefile together with the rest of the program. This is however an excellent opportunity to practise some "project management" where multiple source code files with corresponding headerfiles are to be linked together to the final program memory image. E.g. the already mentioned "HD44780U.h," the accompanying "HD44780U.c" plus the main file "Oblig3a.c"

Seize the opportunity to remind you that all source files will be compiled each by itself into corresponding object files, that in the last stage are linked together into the final image. Refer the GCC flow diagram. This is exactly the reason we need headerfiles for declarations (not definitions) of resources shared between source files. A headerfile exports symbols from one source file to the next. Any source file will therefor normally not import its’ own accompanying headerfile. Consider the headerfil the card of the source file. — contact information to be shared. If access to actual data is to be shared, then the declarations in the headerfile must be prefixed by a 'extern' keyword. The defenition proper is to reside in the source file as before. But you are to present a really good reason to share actual data (as opposed to methods,) or I will come smack your fingers. Declarations of functions are on the other hand and inconsequenty implied 'extern' by nature in the C programming language. But then again it is the latter that is supposed to be shared between source files, not data. This is the foundation of object oriented programming — share the methods data can be manipulated with, not the actual data.

The text strings you are to feed the display controller with is to be represented by a 8-bit character set that is defined in the datasheet of the display controller. The lower 7 bits of this character set is loosely to be considered ISO-646-US-ASCII (American National Standard Code for Information Interchange) or equivalently ECMA-006.

ASCII is a seven bit wide character set that goes all the way back to the age of the teletype/telex. Seven bit equals a table of 128 charactors or letters if you like. ASCII covers the English alphabet with both Capital and lowercase letters plus the numbers 0-9. The rest is used for control characters, exclamiation, period, etc. plus currency symbols. ASCII also represents the lower 7 bits of most western character sets in use, like the Norwegian DOS CP850 / CP865, Windows CP1252 and UNIX ISO/IEC 8859-1. The GCC compiler both expect and produce ASCII. Plain text should therefore not be a problem. To use specialised charaters you will have to employ the C language escape sequences ("\xYY") embedded into your strings. To speed things up a bit, the LCD font table is as follows wrt. character number:

Either { Function Set[F=0] }
[0x00..0x07]  => Eight user defined 5x7(8)-pixel bitmap characters stored in CGRAM.
[0x08..0x0F]  => Duplicate (alias) of above

Or { Function Set[F=1] }

02/12/09 09:52
Four user defined 5x10(11)-pixel bitmap characters stored in CGRAM.
Duplicate (alias) of above
Duplicate (alias) of above
Then 160 entries of 5x7(8)-pixel bitmap characters stored in CGROM
[ASCII] <SPACE> ! " # $ % & ' ( ) * + , - .
[0x20..0x2F] =>
[ASCII] 0 1 2 3 4 5 6 7 8 9
[0x30..0x39] =>
[ASCII] : ; < = > ? @
[0x3A..0x40] =>
[0x41..0x5A] =>
[ASCII] [ ] ^ _ ` |
[0x5B..0x60] =>
[ASCII] { } ~ -> <-
[0x61..0x7A] =>
[ASCII] a b c d e f g h i j k l m n o p q r s t u v w x y z
[0x7B..0x7F] =>
Lastly 32 entries of 5x10(11)-pixel bitmap characters stored in CGROM
Non-printable, non-standard characters. (Ref Dataseet)

Non-printable, non-standard characters. (Ref Dataseet)

The display module will from a hardware point of view behave like a readable- and writeable external memory (RAM.) Unfortunately, this bus protocol does not conform to the AVR external bus interface protocol. We will therefore fall back to bit-banging, in which the display is attached to two somewhat arbitrarily chosen GPIO ports of the AVR microcontroller. One port is in full to be dedicated data bus - DATA[7..0] - whilst three bits of the other GPIO port is dedicated control bus signals. These signals are "Register Select [RS], Read/NotWrite [R/~W]" and "Enable [E]". There are printed circuit boards to be fond in the lab featuring an LCD module and circuitry for direct attachment to STK500 ports.

The essenc of the schematics is as follows:

LCD_DATA[7..0] <= PORTA[7..0]
LCD_RS <= PORTB[5]
LCD_RW <= PORTB[6]
LCD_EN <= PORTB[7]

The data on the data bus (including direction) and the three control signals must be steered in the correct sequence. Study the timing diagram in the datasource of the display controller. The most important diagram is reproduced below. Not that this diagram covers two full bus cycles — first a write cycle, then a read cycle.
You are to emulate the LCD display bus protocol by steering ATmega323 GPIO-ports as if the were the control signals of the LCD bus. What text to write to the display is up to you. But it should be strings of more than 16 characters and both lines of the display is to be in use.

The task is simpler than it sounds as there is no upper limit to long a time you may use strobing the control lines. But you must not do it any faster as what is stated in the dispaly controller datasheet. Most noteworthy is the relatively long delay between two consecutive cycles. This delay is repeated in the last column of the below table. AVR performs one instruction per clock cycle. At standard STK500 clock frequency for target MCU of \textbf{3,68640 MHz} one clock cycle lasts \textbf{271 ns} It should now be possible to come up with a approximation for delay loops. Alternatively \textit{somebody} has already done these calculations for you in the form of ("#include <avr/delay.h>") Which in case requires you to declare correct clock frequency. And beware of \textbf{optimization} above level "-O1" as such (useless) delay loops then will be optimized out of existance.

<table>
<thead>
<tr>
<th>LCD module Instruction</th>
<th>Instruction Code</th>
<th>Description</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Display</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 1</td>
<td>Clears all of DDRAM to &quot;0x20&quot; and resets DDRAM Address Counter [AC] to &quot;0x00 (0x80)&quot;</td>
<td>1.53 ms</td>
</tr>
<tr>
<td>Function Set</td>
<td>0 0 0 0 1 DL N F</td>
<td>-</td>
<td>Interface data width (DL: 8-bits / 4-bits) Number of display lines (N: 2-lines / 1-line) and the CGRAM Font size (F: 5x10(11) [4 psc.] / 5x7(8) [8 psc.] dots)</td>
</tr>
<tr>
<td>Set CGRAM Address</td>
<td>0 0 0 1 AC5 AC4 AC3 AC2 AC1 AC0</td>
<td>Places a CGRAM address [0x40..0x7F] into the Address Counter [AC] register.</td>
<td>39 μs</td>
</tr>
<tr>
<td>Set DDRAM Address</td>
<td>0 0 1 AC6 AC5 AC4 AC3 AC2 AC1 AC0</td>
<td>Places a DDRAM address [0x80..0xFF] into the Address Counter [AC] register.</td>
<td>39 μs</td>
</tr>
<tr>
<td>Read Busy Flag and Address</td>
<td>0 1 BF AC6 AC5 AC4 AC3 AC2 AC1 AC0</td>
<td>Busy flag indicates pending operation. The contents of the Address Counter [AC] register is also returned.</td>
<td>0 μs</td>
</tr>
<tr>
<td>Write Data to RAM</td>
<td>1 0 D7 D6 D5 D4 D3 D2 D1 D0</td>
<td>Write data into the internal DDRAM/GRAM memory location currently pointed to by the Address Counter [AC] register.</td>
<td>43 μs</td>
</tr>
<tr>
<td>Read Data from RAM</td>
<td>1 1 D7 D6 D5 D4 D3 D2 D1 D0</td>
<td>Read data from the internal DDRAM/GRAM memory location currently pointed to by the Address Counter [AC] register.</td>
<td>43 μs</td>
</tr>
</tbody>
</table>

**Part 2**
In this part we will continue feeding the LCD panel, but no longer by predefined text. We will instead read this text from the serial port - or more precisely the 'Universal Asynchronous Receiver/Transmitter' port of the AVR micro controller. Serial data to and from the U(S)ART will have to pass through a levelshifter that translates the signal level from [0..5]VDC TTL level as found within the STK500 and up to ±[10..15]VDC RS232 level more suitable for long stretches of cable. There is a MAX202 levelshifter onboard the STK500 for this purpose. To utilize this levelshifter we will have to connect a strap from "PORTD[1..0]" onto the "TxD/RxD" pins on the STK500. This strap is visible in the picture in a black and red.

We attach a serial cable to the connector marked "RS232 SPARE" on the STK500. The other end of this cable is of cause to be attached to the host computer [PC]. How you manage to get data out of the host computer is irrelevant, but the simplest way is to use a so called terminal emulator. We have installed Tera Term Pro on the lab computers for just this purpose. Also note that RS232 goes back to the youth of computing and as such is completely free of protocol or operating system intervention. WYSIWYG is the rule, and exactly this absence of complicating elements is what makes RS232 such a useful tool for microcontroller purposes - RS232 is available for de casual developer.

Our ATmega323 is equipped with a USART. This is nothing but a UART with the addition of a dedicated clock line making this port capable of synchronous communication. We are to ignore this capability. It might therefore pay to read the shorter explanation of the UART in the datasheet of the AT90S8535 predecessor. All bits not found on the UART are to be left alone or set to zero on the USART. All registers of the ATmega323 not mirrored in the AT90S8535 is best left untouched.
UART is another step up from pure GPIO-ports as UART must be configured and then be enabled to be used. A UART is nothing but a set of parallel-to-serial shift registers with implied clock generation. What is just a fancy way of saying that sender and receiver has agreed upon a ‘baudrate’. Synchronisation of clocks is obtained by means of dedicated start- and stop bits embedded into the data stream. This is handled transparently by HW. UART is equipped with three interrupt request vectors - one for receive, one for transmit and one for data register empty. The latter is useful for buffered transmission. Each and every interrupt vector of the ATmega323 is tabulated in the ('\#include <avr/io323.h>') header file.

/* Interrupt vectors */
#define SIG_INTERRUPT0 VECTOR(1)
#define SIG_INTERRUPT1 VECTOR(2)
#define SIG_INTERRUPT2 VECTOR(3)
#define SIG_OUTPUT_COMPARE2 VECTOR(4)
#define SIG_OVERFLOW2 VECTOR(5)
#define SIG_INPUT_COMPARE1 VECTOR(6)
#define SIG_OUTPUT_COMPARE1A VECTOR(7)
#define SIG_OUTPUT_COMPARE1B VECTOR(8)
#define SIG_OVERFLOW1 VECTOR(9)
The UART of the AT90S8535 is a classic peripheral meaning that the UART is controlled through three(four) registers: A data register [UDR], a control register [UCR] and a status register [USR]. In addition the UART has a configuration register, the Baudrate divisor register [UBRR]. Atmel has messed up this on the ATmega323 renaming [UCR] control register B [UCRB], while the status register [USR] is renamed into control register A [UCRA]. In addition Atmel has introduced a third control register C [UCRC]. The latter is best ignored.

The Baudrate divider register [UBRR(High/Low)] specify the number the system clock [3,6864 MHz] has to be divided with to produce the appropriate baudrate [9600 bps] Table 24 in the AT90S8535 datasheet lists multiple examples. Typically you will load the correct divisor into the [UBRR(H/L)] register (Hint: UBRRH = 23) whereafter you will unmask any necessary UART interrupt source ([RXCIE]) Then you turn on the parts of the UART ([RXEN]) you intend to use.

You will also have to write a Interrupt Service Routine (ISR) that will handle interrupts (IRQs) from the receiver. An ISR is much the same as an ordinary function, but has to be surrounded by a ISR declaration. ISRs is another extension added to the AVR backend of the AVR-GCC compiler. A consequence of this is that the optimizer — that belongs to the GCC frontend — has no concept of ISRs. Any data shared between ISRs and normal program flow must be declared as "volatile". The optimizer will then be prohibited from simply assuming that the data has not changed since last access. The required declarations you will find in ("#include <avr/signal.h>") and ("#include <avr/interrupt.h>") files dealing with ISRs with global interrupts masked and unmasked, respectively. (The interrupt-on-interrupt subject.)

```
#include <avr/signal.h>
#include <avr/interrupt.h>

volatile uint8_t character;

 SIGNAL(SIG_UART_RECV) {
    /* code */
    character = UDR;
 }

void IO_init(void) { /* code */ }

int main(void) {
    IO_init();
    sei(); // Global Interrupt Enable
    /* code */
    cli(); // Global Interrupt Disable
    /* code */
}
```

A ISR is to be called by hardware directly, and must as such never be called upon by software. ISRs simply do not conform to C Calling Convention. Within ISRs you have to remember to clear the flag that caused the interrupt, or else you will immediately reenter the ISR upon 'Return from ISR.' In the case of [RXC] this is as simple as reading the data register [UDR], which you are supposed to do anyway. Implement as KISS a solution as possible.
Geir Frode Sørensen
ELAB, Fys-UiO
Rom 115, vestfløyen.
TEL: 22 85 64 39
geirfrs ved fys punkt uio punkt no