Lab 1: Intro to HCS12 ASM Programming

Goals:

In Computer Architecture you will learn the architecture, programming & debugging of embedded systems: in the end you've developed a radio-frequency controlled clock based on the DCF77 signal.

In the first lab, You learn how to program & debug software using Assembler, laying the ground-work for the later labs: learning how to control the UI based on LEDs, the LCD display and buttons as input.

In Lab 2, you'll develop the core of the clock, using the HCS12's timer module and some buttons, to let the user set the initial time. Additionally, the clock will use the analog-to-digital converter (ADC) to measure the room temperature.

In Lab 3, you'll extend the clock with a DCF77 radio-frequency interface, receiving the current date and time information via antenna, so that the clock automatically sets time, switches between normal time and daylight savings time and tracks the date and leap years correctly.

Successful completion:

In each Lab You each must deliver & present working software. Additionally in Lab 2 and Lab 3 you must deliver a lab report with documented software.

Please mark in Your source code, who worked on what parts of the code. The default is groups of **two persons**, any code copied (other than from the samples) must be marked.

Starting with Assembler

Preparation Task 1.1:

Write a short HCS 12 assembler program based on template **lab1a**. The program shall:

- Count a 16 bit value in a loop starting from 0 counting up to 63 in steps of 2 and then roll over,
- Output the lower 8 bits of the counter value on Dragon12 board's LEDs,
- The loop shall call a subroutine `delay_0_5sec`, which implements a time delay of 0.5sec.
  Hint: the subroutine shall not use global variables, as these would not work when the subroutine is used recursively. Global (read-only) constants are not critical.

Test your program in the simulator/debugger. Use the `IO_Led` Component to simulate the LEDs.
Note: The simulator/debugger does not run in real-time. Thus you cannot measure time with a watch. You must check the CPU clock cycles, which are show in the bottom right corner of the Register Window, and convert them into time by multiplying the CPU clock period by 1/24 MHz.

Preparation Task 1.2:

Make a copy of your CodeWarrior project of preparation task 1.1. Add a new file `delay.asm` to the project (CodeWarrior *File* → *New Text File*) and move `delay_0_5sec` to the new file.

Add another file `led.asm` to implement four new subroutines:

`initLED`	This subroutine initializes all required ports to drive the LEDs on the Dragon12 board. At the end of this function all LEDs are turned off.

`setLED`	This subroutine has an 8 bit calling parameter in register B. The bits in this parameter define, which of the 8 LEDs are turned on (1) or off (0).
`getLED`	This subroutine returns an 8 bit value in register B. The bits indicate the current state of the 8 LEDs: on (1) or off (0).
`toggleLED`	This subroutine has an 8 bit calling parameter in register B. The bits define, which of the 8 LEDs shall be toggled. Note: Toggling can be achieved by XOR-ing the current state of the LEDs with the parameter.

From the perspective of the calling code, all functions must not change CPU register D, X and Y (with the exception of register B which holds the return value for function `getLED`.
All code driving the LEDs in the main program shall be removed and substituted by calls to above subroutines. Test with the simulator/debugger.

Files `delay.asm` and `led.asm` are the first functions of a code library, which you may reuse in all further projects. To reuse the files, simply copy them to the source subdirectory of the new project and then add them to the project (in CodeWarrior's directory view right-click directory *Sources* – select menu *Add File*). *Hint: Soft-Links (`ln -s`) under Linux & MacOS allow sharing the same file.*

Assembler Utility functions

Preparation Task 2.1:

Write a subroutine in HCS12 assembler based on template `lab1c` into a new file `toLower.asm`:

```
void toLower(char * string)
```

This subroutine shall convert a null-terminated ASCII string from upper case to lower case characters. The string may contain a mix of upper and lower case characters as well as numbers and special characters like commas and dots:

- The conversion algorithm may use a trick: in ASCII, the upper and lower case alphabets are arranged in two blocks, where the only difference is bit5, being 0 for upper case and 1 for lower case characters. The conversion can be done using a bitwise OR operation:
  $\texttt{ASCII-Code}_{\texttt{Lower\_case\_character}} = \texttt{ASCII-Code}_{\texttt{Upper\_case\_character}} \mid (\texttt{0x1} << \texttt{5});$
- Alternatively one could use the fact, that upper and lower case character ASCII code values use a constant difference:
  $\texttt{ASCII-Code}_{\texttt{Lower\_case\_character}} = \texttt{ASCII-Code}_{\texttt{Upper\_case\_character}} + 32_{D};$
- The subroutine gets a pointer to the string in register D when called. The string itself is in a global array in RAM and shall be modified by the subroutine via the pointer in register D.
- From the perspective of the calling code, the function must not change CPU registers D, X and Y and must not use any global variables.

Write a second subroutine in HCS12 assembler in `toLower.asm`:

```
void strCpy(char * destination, char * string)
```

which copies and ASCIIZ (zero/NUL-terminated) string from memory location pointed to by parameter `source` (in register X) into memory location pointed to by parameter `destination` (in register Y). The memory area of the destination in RAM must provide enough space to hold the string including the terminating zero.
Write a test program main.asm for both subroutines. Use an initialized string in ROM and copy it to RAM using `strCpy`, before calling `toLower`.

Carefully design the program using flow-charts for the `toLower` subroutine and for the main test program, before you start writing code. Use the simulator / debugger to test your program.

Preparation Task 2.2:

Write a subroutine in HCS12 assembler into a new file `hexToASCII.asm`:

```
void hexToASCII(char * string, unsigned int val)
```

This subroutine shall convert a 16 bit hexadecimal value in parameter `val` (passed in register D) into a NUL-terminated ASCII-string. Example: For value $F018, the subroutine shall return a string consisting of ASCII values “F”, “0”, “1” and “8”, i.e. 46_H, 30_H, 31_H and 38_H as well as a terminating 00_H. The four hex digits shall be preceded by “0x” resulting in the string “0xF018” followed by NUL, aka in total 7 bytes.

- The 16 bit variable `val` is passed in register D
- The pointer `string` is passed in register X and points to a location in RAM, large enough to hold the result
- The conversion of each hexadecimal digit (4bit, aka nibble) into its ASCII-code may use a constant table, aka in ROM:
  ```
  H2A: DC.B  "0123456789ABCDEF"
  ```
  The table is read via the hexadecimal digit as index into the array. The constant H2A shall be defined as global constant in the same file as the subroutine.
  The C-style pseudo code for this algorithm could be written as:
  ```
  string[0] = '0';                   // Place “0x” at beginning
  string[1] = 'x';                   // Place “0x” at beginning
  string[2] = H2A[(val >> 12) & 0xF]; // 16³ hex digit → ASCII
  string[3] = H2A[(val >> 8)  & 0xF]; // 16² hex digit → ASCII
  string[4] = H2A[(val >> 4)  & 0xF]; // 16¹ hex digit → ASCII
  string[5] = H2A[ val        & 0xF]; // 16⁰ hex digit → ASCII
  string[6] = 0;                     // Terminating NUL character
  ```
- From the perspective of the calling code, the subroutine must not change CPU registers D, X and Y. The subroutine must not access global variables, but may access global constants.

Write a `main.asm` to test the subroutine `hexToASCII()` by incrementing a 16 bit counter in a loop, calling `hexToASCII()` with the counter.
In the debugger, check for the “corner cases” 0, 1, …, 15, 16, …, 255, 256, 257, … , 65535 and finally the wrapping into 0, so that proper ASCII strings are generated.

Preparation Task 2.3:

Write a subroutine in HCS12 assembler into a new file `decToASCII.asm`:

```
void decToASCII(char * string, int val)
```

This subroutine shall convert a 16 bit signed decimal value in parameter `val` (passed in register D) into a NUL-terminated ASCII-string, which represents the decimal value:

- The 16-bit variable val is passed in register D
- The pointer string is passed in register X and stored in RAM, passed by the calling program,. Don’t forget to reserve enough space for the sign, all digits and the terminating zero.
- Negative values shall be preceded by a ‘-‘ character, positive values by a ‘ ‘ (space) character. Values shall be right aligned.
  The C-style pseudo code for this algorithm:

```
if (val >= 0)        { string[0] = ' '; }  // Check sign of val
else                 { string[0] = '-';    // Convert negative to
                       val = -val;         // positive value
(Q,R) = val / 10000; string[1]=Q+'0';      // 10⁴ digit→ASCII
(Q,R) = val / 1000;  string[2]=Q+'0';      // 10³ digit→ASCII
(Q,R) = val / 100;   string[3]=Q+'0';      // 10² digit→ASCII
(Q,R) = val / 10;    string[4]=Q+'0';      // 10¹ digit→ASCII
                     string[5]=R+'0';      // 10⁰ digit→ASCII
                     string[6]=0;          // Terminating NUL
```

Note: (Q, R) = A / B → Q = string result of division A/B, R = remainder (german Rest) of integer division. Use the HCS12 instruction `IDIV` for the integer division. The `R+'0'` is adding the ASCII character `'0'` (aka 0x30, aka 48_D) to the remainder. This is an alternative method to convert a decimal digit rather than addressing the array via `H2A[R]`.

- From the perspective of the calling code, the subroutine must not change CPU register D, X an Y. The subroutine must not access global variables.

Write a `main.asm` to test the subroutine `decToASCII()` by incrementing a 16 bit signed counter in a loop, calling `decToASCII()` with the counter. Carefully design the program using program flow charts for the subroutine and for the test program before you start writing code.
In the debugger, check for the “corner cases” -32768, -32767, …, -1, 0, 1, 2, 100, 255, 256, …, 32767 and the wrap-around to -32768!

Do test both subroutines hexToASCII and decToASCII carefully. These subroutines will be used heavily in later programs. Bugs detected early will save you a lot of time and annoying debugging in later labs.

Interfacing with the real world

Lab Task 3.1: Printing to the LCD display on the Dragon12 board

In project template **lab1-lcdVorlage** you find a program, which outputs a NUL-terminated ASCII string to the Dragon12’s LCD. Unfortunately, the function `writeLine` in this template has at least 2 bugs. Run the program, identify the problems and fix the bugs.
Modify the main program so that it additionally outputs the following strings by sequentially calling writeLine:

```
msgA: DC.B "ABCDEFGHIJKLMnopqrstuvwxyz1234567890", 0  → in LCD line 0
msgB: DC.B "is this OK?", 0                           → in LCD line 1
msgC: DC.B "Keep texts short!", 0                     → in LCD line 0
msgD: DC.B "Oh yeah!", 0                              → in LCD line 1
```

Run the program step by step with the debugger and make sure, that each text appears correctly in the specified line of the LCD display and that you don’t see any funny characters or leftovers by the previous output. Again: Thoroughly test, as you will use the LCD driver code in later programs.

Make sure, that the text appears left aligned in the display and that you always output 16 characters. If the original string is shorter, you must pad the rest of the line with spaces.

Lab Task 3.2:

Use a copy of the project lcd1-Vorlage. Copy your files `led.asm`, `delay.asm`, `hexToASCII.asm` and `decToASCII.asm` into the source folder and add them to the CodeWarrior project. Write a main program, which increments a 16 bit counter variable `i` into a loop and output `i` on the LCD display:

- The decimal value of `i` shall be display on LCD line 0. The hexadecimal value of `i` shall be displayed on LCD line 1. Both values are left aligned. Use `decToASCII()` and `hexToASCII()` to convert `i` an `writeLine()` to output the strings to the LCD display.
- Use your `led.asm` functions to write the lower 8 bits of `i` to the LEDs.
- The loop shall count with approx. 0.5s, i.e. 2 increments per second to allow visual debugging.

Debug your program. When your program works, add the following features:

- When no button is pressed, the counter shall count up in steps of 1
- When the button at port PTH.0 is pressed, the counter shall **increment** in steps of 16
- When the button at port PTH.1 is pressed, the counter shall **increment** in steps of 10
- When the button at port PTH.2 is pressed, the counter shall **decrement** in steps of 16
- When the button at port PTH.3 is pressed, the counter shall **decrement** in steps of 10

Note: Unfortunately, the simulated buttons in the debugger operate inverse to the buttons on the real Dragon12 hardware, i.e. on the Dragon12 board, when you click a button a logic 0 is generated at the respective pin on port PTH and logic 1 is generated when not pressed. In the debugger, when you mouse click a simulated button, it generates logic 1, while inactive buttons read out as logic 0. Additionally on Dragon12 board you can click multiple buttons simultaneously, while the simulator only allows one button at a time. To write a program, which can be easily adapted for the simulator and the Dragon12 hardware, use the HCS12 instruction BRSET to test buttons with the simulator and substitute these instructions by BRCLR to work with the real board. E.g. for the simulated buttons in the debugger:

```
      BRSET PTH, #$01, button0pressed     ; check if button on port PTH.0
      …                                   ; continue here if not pressed
button0Pressed: …                         ; continue here if is pressed
```

When you have thoroughly tested your program, make sure, that your program code is accurately commented and that all your subroutines have an interface description as described in appendix A of this laboratory sheet.
Then please upload your CodeWarrior project for lab task 3.2 as ZIP-file onto Moodle.

Appendix A: Code Comments and Interface Descriptions for Subroutines

Your code must have a reasonable amount of comments. Comments shall describe what the code is **supposed to** do on a per-block basis, i.e. describe complete code blocks but not individual statements.

	Bad style:	**Good style:**
`LDAB 0, X`	`; load byte into B`	`; check for end of string`
`TSTB`	`; Test if B is 0`	
`BEQ cont`	`; Branch if equal`	`; if no, goto cont`
`STAB 0, Y`	`; Store Byte`	`; if yes, store next charac-ter`
…		
`cont:`	`; continuation`	`; handle next character`

Additionally to commenting code blocks, you shall provide an interface description for all subroutines. The interface description **must** contain:

- A short description of the purpose of the function
- For assembly functions:
  - A list of all calling parameters (including their type and value range), where the parameter is placed (register or stack?)
  - Return parameter(s) of the subroutine (register?)
  - Which registers are modified by the subroutine (from the caller's perspective)
- For C functions:
  - Meanings of special values (e.g. int interpreted as Boolean, 0 being false and 1 true)
  - Range limits for function parameters, e.g. `int hours // 0 .. 23 only`
- Parameter and error checks done inside of the function, if any.

The interface description is inserted as comments directly above the subroutine's code in the program module. For many programming languages like Doxygen or JavaDoc extract comments and automatically generate suitable and portable documentation.

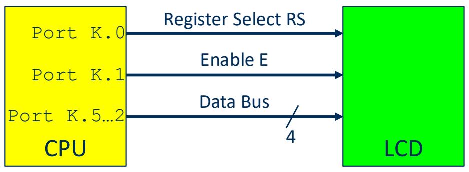
Examples:
```
; Public interface function: initLCD
;    Initialize LCD (called once before using the LCD).
; Parameter: -
; Returns: -
; Registers: Unchanged (e.g. upon return, D, X and Y are restored)
…

; Public interface function: writeLine
;   Write zero-terminated string to LCD
; Parameter:
;   X     pointer to zero-terminated ASCII string
;   B     row number (0 or 1)
; Returns: -
; Registers: Unchanged
; Error checks:
;   Only the first 16 characters of a string will be display.
;   Shorter strings will be padded by blanks.
```

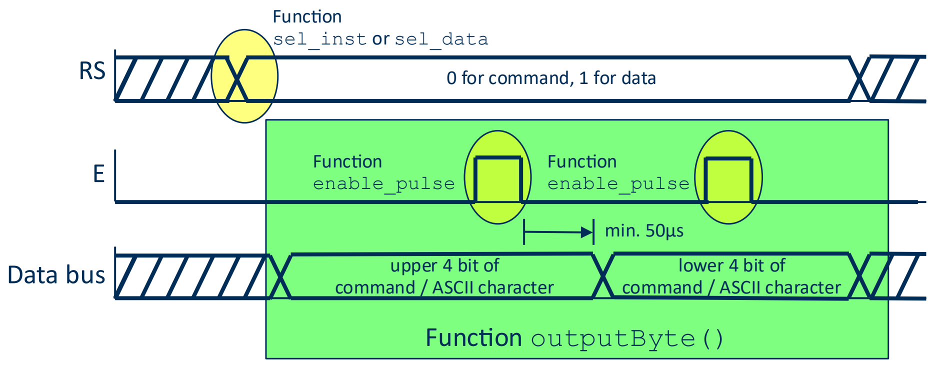
Appendix B: Interface to the LCD Display

For those interested how the interface between the microcontroller and the LCD display works, here are some details.
Note: This is the hardware interface on the Dragon12 board. The LCD interface in the debugger works slightly different, but the software driver `LCD.asm` should work for both.

- **Hardware:**



- **Output an 8 bit command or an ASCII character in `lcd.asm`:**


Function sel_inst or sel_data
RS
0 for command, 1 for data
E
Function enable_pulse
Function enable_pulse
min. 50µs
Data bus
upper 4 bit of command / ASCII character
lower 4 bit of command / ASCII character
Function outputByte()

Example for command output:	80_H … Move cursor to line 0
	$C0_H$ … Move cursor to line 1
Example for data output:	Display ASCII character at current text position

Instruction sequence in driver program `lcd.asm`:

1. Set port K.0 (aka Signal RS) to select between command and data
2. Put upper 4 bit (nibble) of command or ASCII character on port K.5…2 (aka data bus) to LCD
3. Set port K.1 (aka signal E) to 1 and back again to 0 to store the data into the LCD
4. Repeat steps 2 and 3 for the lower 4 bit of the command or character.

Step 1. is implemented in functions `sel_inst()` and `sel_data()`. Steps 2 to 4 are executed in function `outputByte()`. Step 3 uses subroutine `enable_pulse()`.