# Lab 2: Clock – HCS12 Interrupts & I/O

**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.

## Analysis of a Timer Interrupt Program

**Preparation Task 2.1:**

In template **lab2-tickerVorlage** you'll find a test program for our ticker driver, which uses one of the Enhanced Capture Timer (ECT) channels. The program shall toggle an LED on port B of the Dragon12 board once per second. The LED is controlled in the Interrupt-Service-Routine `isrECT4` (in module `ticker.asm`). The ISR shall be triggered periodically every 10ms. The 1s tick will be generated via a software counter in variable `ticks`. The timer is initialized in function `initTicker`. The code assumes a CPU bus clock of 24 MHz. Via a divider (prescaler factor 128) a clock frequency of 187,5 kHz is generated for the timer. Based on this frequency the ECT generates a 10ms interrupt period. Unfortunately, the code in `ticker.asm` has (at least) 2 bugs. Thus, the timer interrupt and the LEDs will not work as expected.
Test the program with the simulator/debugger and remove the bugs. Please note: The debugger does not run in real-time, so you need to measure the timer period via watching the CPU clock cycles in the register window of the debugger (1 CPU clock cycle = 1/24 MHz)

## Design of the Clock program

**Requirements**

The requirements for the clock and the thermometer are:

- The current time shall be displayed on the LCD in format HH:MM:SS and updated once per second (range for hours 0 … 23, range for minutes and seconds 0 … 59). The LED on Port B.0 shall toggle once per second. For more details see chapter A.2 in the appendix.

- On program start, the time shall be initialized to 11:59:30. By shortly pressing button SW2, the clock shall be switched from *Normal Mode* to *Set Mode*. In *Set Mode* by pressing buttons SW3, SW4 and SW5 the user can increment the hours, minutes and seconds to set the time. The LED on port B.7 shall be turned on when *Set Mode* is active.
- Additionally to the time, the LCD shall display the temperature (e.g. 25°C). The temperature is measured via the temperature sensitive resistor (simulated by a potentiometer in the lab) on analog port AD.7. The voltage range is 0 … 5V representing a temperature of -30 … +70°C. Your program must measure the analog voltage and convert it into temperature value in "°C". The temperature shall be updated periodically once per second. For more details see chapter A.2 in the appendix.
- The first line of the LCD display shall periodically toggle between the name of all group members and the text "© IT W2021/22", see chapter A.2 in the appendix for details.

The software shall be done according to these requirements:

- Use the 1s tick generated by module `ticker.asm` from prep task 2.1. For the LCD and the LED use modules `LCD.asm`, `decToASCII.asm` and `LED.asm`.
- Split your program in several modules. E.g. for the ADC converter write a module with driver functions. Combine the temperature calculation in one module and the clock function in another module. Keep *Normal Mode* and *Set Mode* separate in your code.
- Keep your main program `main()` as short as possible. The main program shall call initialization functions for all hardware components and the run in a loop, calling the clock and the temperature functions once per second. Your interrupt routine(s) shall only deal with the interrupt hardware, but not include more complex operations. Communication between the ISR(s) and the other parts of the program shall be done via global variables. Especially, your interrupt routines must not directly access the LCD display.
- To test the state of the buttons, use polling, not interrupts.

## Choice of programming language

In this lab, you may write your program in a mix of C and HCS12 assembler. See appendix B in this lab manual for a tutorial, how to call assembler functions from C and vice versa and how to exchange data between the two programming languages.

**Preparation Task 3.1:**

Read the tutorial in appendix B and run the CodeWarrior project `lab2-MixedCode.mcp` to learn how to mix C and assembler code.

## Design and Implementation

**Preparation Task 3.2:**

Template project **lab-2-Uhr-C-Vorlage** may be used as the basis for your clock program. The following implementation hints need to be considered:

- The program must start with `void main(void)` written in C. Within `main`, your code shall call the initialization functions for the hardware peripherals, e.g. `initLCD`, `initClock`, etc., and implement an infinite loop which checks the `clockEvent` variable. See appendix A.5.1 for details.
- You must not use any of C's standard library functions, i.e. if you need a function, you have to code it yourself in C or assembler.
- The template project **lab2-Uhr-C-Vorlage** contains buggy dummy routines for `ticker.asm` and `lcd.asm`. Overwrite `ticker.asm` with the file you debugged in your Prep Task above

and `lcd.asm` with the file you debugged in lab 1. Also please copy file `decToASCII.asm` which you developed in lab 1 into your source directory and add the file to the project. Wrapper functions for the LCD driver and `decToASCII` are provided (see `main.c`). Implement more wrapper functions yourself, if needed.

Before you start coding, design your program using a module plan, flow charts for all functions, a data dictionary and an interface description for all subroutines. At the end of the lab you have to deliver a **complete documentation for your program**, see appendix A of this manual for details. Your documentation may be written in either English or German, but not in a mix of languages.

Based on your design **write and debug your program**. As the simulator/debugger has some limitations, e.g. it does not simulate the ADC, you have to substitute several functions with dummy code. E.g. instead of reading the analog value via the ADC use a fixed value when testing the subroutine, which converts the analog value into the temperature in "°C".

Coding and debugging should be done incrementally, i.e. write and debug one function at a time. To test each function, you may need to write additional test functions. Then combine the subroutines required for the *Normal Mode* of the clock. Then add the *Set Mode*. When everything works, add the temperature measurement and display etc. Writing and testing everything in one step will surely fail.

## Testing your Program on the Dragon12 Board

**Lab Task 4.1:**

In the lab run your program on the Dragon12 board. Add and debug those functions, which you could not test with the simulator/debugger. Manually set the clock to 23:59:00. Is setting the clock easy to use? If not, modify your strategy.

Check, if the clock handles the overflow at midnight (23:59:59 → 00:00:00) correctly in *Set Mode* and in *Normal Mode*.

**Lab Task 4.2:**

Modify your program in such a way, that the time display can be configured to a 12h mode rather than the 24h mode. In the 12h mode, the times from midnight to noon shall be displayed with "am" attached, times from noon to midnight with "pm".

Whether the 12h or the 24h mode is used, is defined via conditional compilation. You shall define a symbol `SELECT12HOURS`, which is set to 0 or to 1. Code sections, which are required for the 24h mode or for the 12h mode exclusively, shall be encapsulated with assembler directive "`IF SELECT12HOURS==` …" (similar to `#ifdef` … `#else` … `#endif` in C). See the HCS12-Assembler's online help in CodeWarrior, chapter "Conditional Assembly Directives" for details.

Check if the clock handles overflows: 11:59:59am → 12:00:00pm
12:59:59pm → 01:00:00pm
11:59:59pm → 12:00:00am
12:59:59am → 01:00:00am correctly.

See https://en.wikipedia.org/wiki/12-hour_clock for details about the 12 hour clock.

Then please upload your CodeWarrior project for lab task 4.2 as ZIP-file onto Moodle including the program documentation as PDF.

# Appendix A: Elements of Your Program Documentation

### A.1 Functional Requirements

See the chapter above in this lab manual!

### A.2 User Interface of the Program

LCD display

- 1st line: Names of all group members or "© IT WS2021/2022" alternating every 10s
- 2nd line:23:55:31 25oC Current time in format HH:MM:SS decimal, left aligned
  Temperature in degrees Celsius decimal, right aligned

  Numbers < 10 shall be display without leading zeros, no "+" sign for positive values!

LED display:

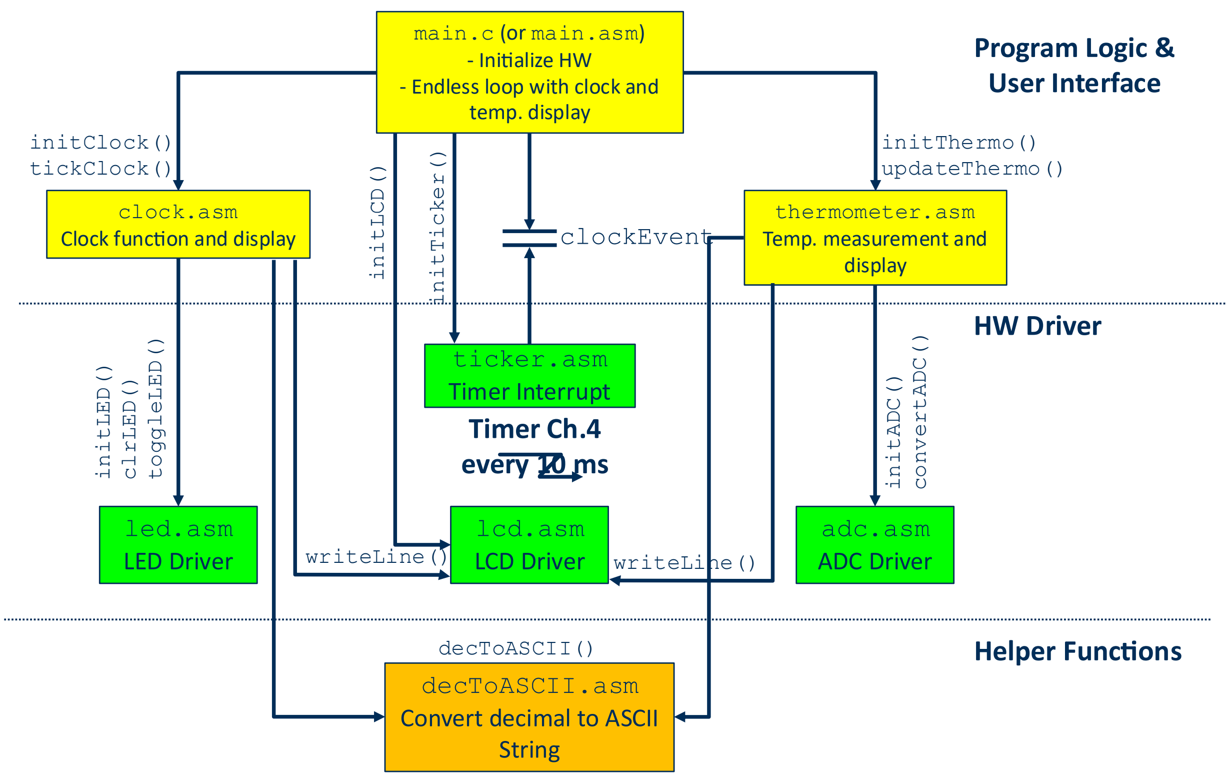
- LED0: toggles once per second (in *Normal Mode* and in *Set Mode*)
- LED7: off in *Normal Mode*, on in *Set Mode*.

Operating Buttons:

- SW2: Short press to switch to *Set Mode*, press again to switch back to *Normal Mode*.
- SW3: Press to increment the hours (in clock *Set Mode* only)
- SW4: Press to increment the minutes (in clock *Set Mode* only)
- SW5: Press to increment the seconds (in clock *Set Mode* only)

### A.3 Module overview

Overview of all program modules and their call interfaces:



The module overview shows all modules (files with source code) of a project plus all function calls **between** modules. Functions which are only used within a module are not shown.

## A.4 Data dictionary

### A.4.1 List of all global variables (incomplete)

| **Module**, where declared (in [] other modules, where used) | **Variable name** | **C-like datatype** | **Purpose** |
|---|---|---|---|
| main [ticker] | clockEvent | Boolean (unsigned char) | Periodically indicates one second has elapsed |
| clock | hrs | unsigned char (0..23) | Hours (as number) |
| | mins | unsigned char (0..59) | Minutes (as number) |
| | … | | |
| | time | unsigned char[9] | Time as string |
| thermometer | Temperature | unsigned char[6] | Temp. as string -XX°C\0 |

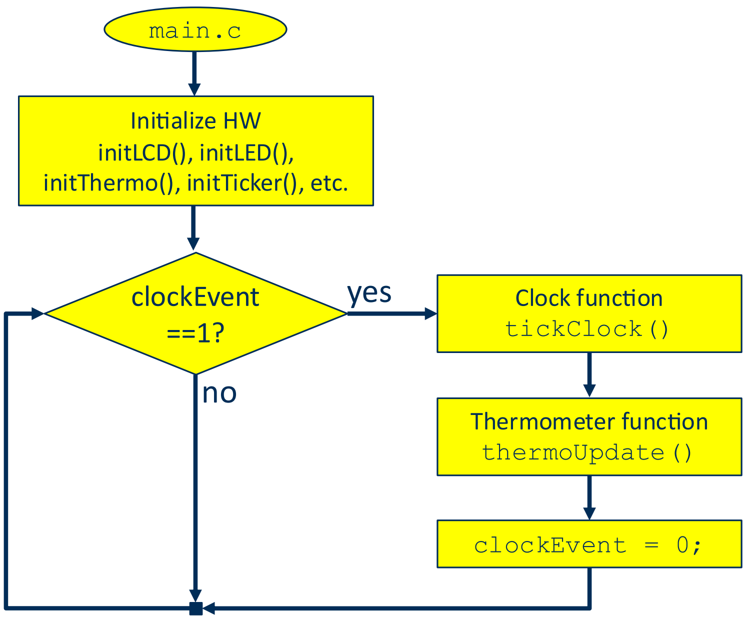
For variables, which do not use the full range of the data type, the acceptable range shall be specified, e.g. hours = 0 … 23.

### A.4.2 Hardware Resources

| **Module** | **HCS12 or Dragon12 HW resource** | **Purpose** |
|---|---|---|
| Clock Thermometer | CPU Port K LCD Display | Display: Line 1: Text Line 2: Time & Temperature |
| LED [Clock] | Port B, Port J.1 LEDs | Various status signals |
| AD [Thermometer] | ATD0 Channel 7 | Temperature Sensor |
| Clock | Port H.3…0 Buttons SW2 … SW5 | Set the time / mode |
| Ticker [Clock] | Enhanced Capture Timer Channel 4 | 10 ms ticker |
| … | … | … |

## A.5 Flow chart for all modules

### A.5.1 Main Program


main.c
Initialize HW initLCD(), initLED(), initThermo(), initTicker(), etc.
clockEvent ==1?
yes
no
Clock function tickClock()
Thermometer function thermoUpdate()
clockEvent = 0;

### A.5.2 Clock Module

Etc.

**Note**: Don't forget to document module `LED.asm` and `ticker.asm`! The module `LCD.asm` need not be documented.

**A.6 Interface Description for all Subroutines**

You must provide an interface description for all subroutines. The interface description must contain:

- A short description of the purpose of the function,
- A list of all calling parameters (including their type and value range), for assembler functions: where is the parameter placed (register or stack?)
- Return parameter of the subroutine (register?)
- Parameter and error checks done inside of the function, if any.
- Which registers are changed by the subroutine (for assembler functions)

The interface description is inserted as comment directly above the subroutine's code in the program module. Tools like Doxygen or Javadoc may extract such comments and automatically generate HTML or PDF document (not used in the lab).

Example:

```
; Public interface function: initLCD ... Initialize LCD (called
;                                        once before using the LCD)
;  Parameter:    -
;  Return:       -
;  Registers:    Unchanged (when function returns)

; Public interface function: writeLine ... Write zero-terminated
;                                          string to LCD
;  Parameter:
;          X ... pointer to zero-terminated ASCII string
;          B ... row number (0 or 1)
;  Return:       -
;  Registers:    Unchanged (when function returns)
```

# Appendix B: Mixed Programming in C and HCS12 Assembler

When writing a modular program in a mix of different programming languages, one must understand how to call a subroutine from another programming language and how to exchange data. This appendix describes the most important issues to be used in this lab. All program executables may be found in CodeWarrior project **lab2-MixedCode**.

### B.1 Cross-module Data Exchange via Global Variables

The simplest way to exchange data between C and assembler modules is via global variables. They may be declared on either side and used on the other one, provided that the data types are compatible.

| C module | HCS12 ASM module |
|---|---|
| `// C data types` | `; Compatible assembler data types` |
| `char                        var;` | `var: DS.B 1      ; in ASM signed &` |
| `unsigned char               var;` | `var: DS.B 1      ; unsigned vars` |
| `const char                  var:` | `var: DC.B 1      ; do not differ!` |
| `const unsigned char         var;` | `var: DC.B 1` |
| `int / unsigned int          var;` | `var: DS.W 1` |
| `const int / unsigned int    var;` | `var: DC.W 1` |
| `long / unsigned long        var;` | `var: DS.L 1` |
| `const long / unsigned long var;` | `var: DC.L 1` |
| `char                        var[4];` | `var: DS.B 4      ; We need NUL!!!` |
| **Use C variables from assembler code:** | |
| `// Declare C variables` | `; Import C variables_` |
| `int var1C;` | `XREF var1C, var2C` |
| `char var2C;` | `; Use C variables:` |
| | `LDD var1C` |
| | `STAA var2C` |
| **Use assembler variables from C code:** | |
| `// Reference ASM variables:` | `; Export ASM variables` |
| `extern int var1A;` | `XDEF var1A, var2A` |
| `extern char var2A;` | `; Declare ASM variables` |
| | `.data:     SECTION` |
| `// Use ASM variables` | `var1A:     DS.W 1` |
| `var1A = 2 * var2a;` | `var2A:     DS.B 1` |

### B.2 Functions without parameters and return values

When data exchange is handled via global variables (see above), functions may be defined on either side and called directly from the other one.

| C module | HCS12 ASM module |
|---|---|
| **Call assembler function from C code:** | |
| `// ASM function prototype` | `; Export ASM function` |
| `void function1A(void);` | `XDEF function1A` |
| `// Call ASM function from C` | `; Program code for ASM function` |
| `function1A();` | `function1A: … ; Make sure, all` |
| | `RTS      ; regs are restored` |
| **Call C function from assembler code:** | |
| `// Program code for C function` | `; Import C function` |
| `void function1C(void) {` | `XREF function1C` |
| `…` | `; Call C function from Assembler` |
| `}` | `JSR function1C` |

**B.3 Functions with parameters and/or return values**

Most assembler functions pass parameters and return values via registers, while C uses a more complex method (to be discussed later in chapter 4 of the lecture manuscript). To call such assembler functions from C and vice versa, so-called **wrapper functions** are required to handle parameter passing. For convenience, these wrapper functions should be defined in C using so-called inline assembler with keyword `asm`, which allows to embed assembler statements within C code to access CPU registers.

| C module | HCS12 ASM module |
|---|---|

**Call assembler function from C code via wrapper function:**

C module:

```
// Prototype for ASM function
void function2A(void);
// Program code of wrapper fct.
int function2A_wrapper(
                int param1,
                int param2) {
  int returnVal;
  // copy parameters to registers
  asm LDX param1;
  asm LDY param2;
  // Call ASM function
  function2A();
  // Store return value
  asm STD returnVal;
  return returnVal;
}
…
// Call ASM function via wrapper
val = function2A_wrapper(3, 9);
…
```

HCS12 ASM module:

```
; Export ASM function
     XDEF function2A
; Program code for ASM function
; takes two 16 bit parameters in
; regs X and Y and returns a
; 16 bit value in register D
function2A: …
     …
     RTS
```

**Call C function from assembler code via wrapper function:**

C module:

```
// Program code for C function
// with two parameters and return
int function2C( int param1,
                int param2) {
     …
}
// Wrapper function, expects
// parameters in regs X, Y and
// returns a value in reg D
void function2C_wrapper(void) {
     int param1, param2, retVal;
     // Get parameters from regs
     asm STX param1;
     asm STY param2;
     // Call C function
     retVal = function2C(param1,
                     param2);
     // Copy return value to reg
     asm LDD retVal;
 }
```

HCS12 ASM module:

```
; Import C function
     XREF function2C_wrapper




; Pass parameters via regs X, Y
     LDX param1
     LDY param2
; Call C function (via wrapper)
     JSR function2C_wrapper
; Store returen value (in reg D)
     STD val
```