Computer Systems 245 - Practical 1

31 July 2015

1 Goal

Get to know the Renesas e2 Studio IDE.

- Create a new project in e2 studio.
- Compile and program the project on to a microcontroller.
- Write basic C and assembly code and see how they integrate

All source code that is listed in this practical is also available online on the subject's webpage. It should help with copying the code to the IDE.
PS: All text in blocks like these are hints.

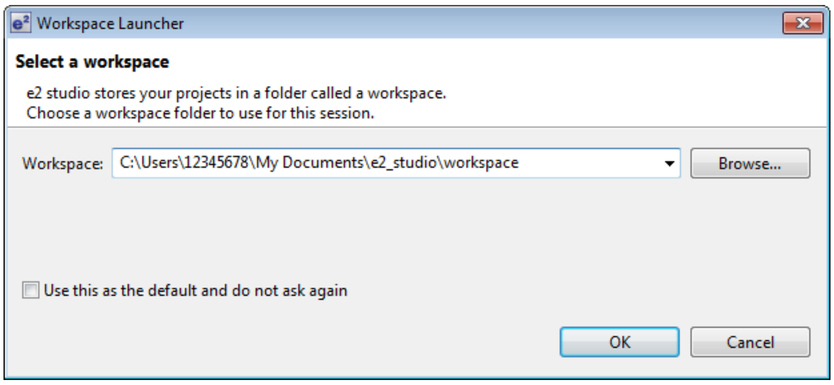
2 Setting up a project in e2 studio

Start by opening e2 Studio. It should be located on the lab pcs under
Start ⇒ All Programs ⇒ Renesas Elecronics e2 studio ⇒ Renesas e2 studio

When the IDE is opened for the first time it will ask you for a default workspace location. Change it to be a folder under your user's My Documents.

If you are asked for **Toolchain Registry**, choose GNURL78, and click Register.

After this you will be greeted by 'n fullscreen **welcome** page. **Close it.**


Workspace Launcher

Select a workspace

e2 studio stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\Users\12345678\My Documents\e2_studio\workspace

When you are coding on the lab PC's it is a good idea to have the workspace location on the local C: drive. If you use the H: drive for storage, the files need to be transferred every time you compile, causing delays. At the end of your practical you can copy the code to your H: drive or to a flashdrive. It will be useful to remember the path where e2's workspace is stored.

After you closed the welcome screen, go to **File ⇒ New C Project**. A new window will open with a wizard to create a new project. Note that we are using C, not C-plus-plus.

Project Name RSPrac1 (or any name you choose)

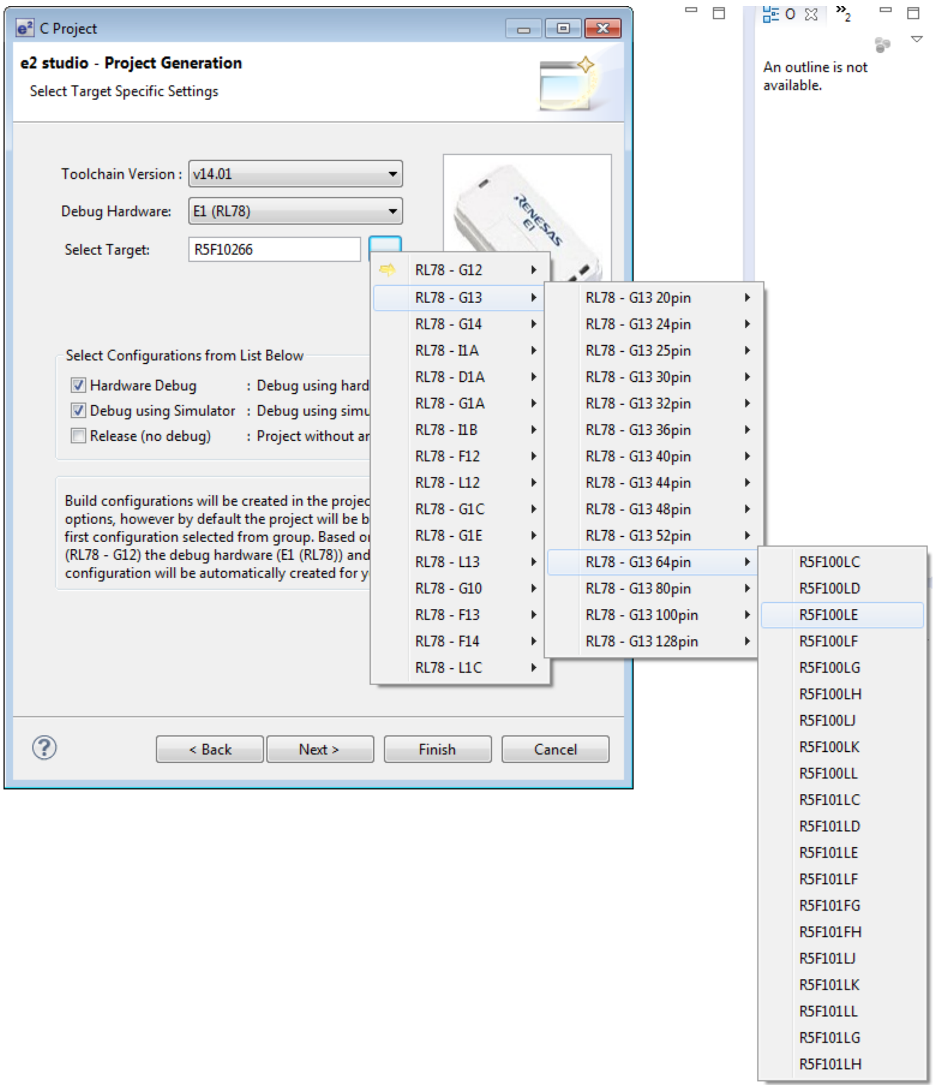
Project type Sample Project

Toolchain KPIT GNURL78-ELF Toolchain

Toolchain version v14.01 (or whichever is the newest)

Debug Hardware E1 (RL78)

Select Target R5F100LE (RL78-G13⇒RL78-G13 64pin⇒ RF5100LE)



C Project

e2 studio - Project Generation

Select Target Specific Settings

Toolchain Version : v14.01

Debug Hardware: E1 (RL78)

Select Target: R5F10266

Select Configurations from List Below

☑ Hardware Debug : Debug using hard

☑ Debug using Simulator : Debug using simu

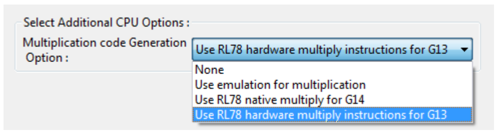
☐ Release (no debug) : Project without ar

Build configurations will be created in the projec options, however by default the project will be b first configuration selected from group. Based o (RL78 - G12) the debug hardware (E1 (RL78)) and configuration will be automatically created for y

An outline is not available.

Code Generator Settings Don't select the peripheral code generator. Simply click next. We will use this function in e-design next year.

Select Additional CPU Options Choose *Use RL78 hardware multiply instructions for G13*



Select Additional CPU Options :

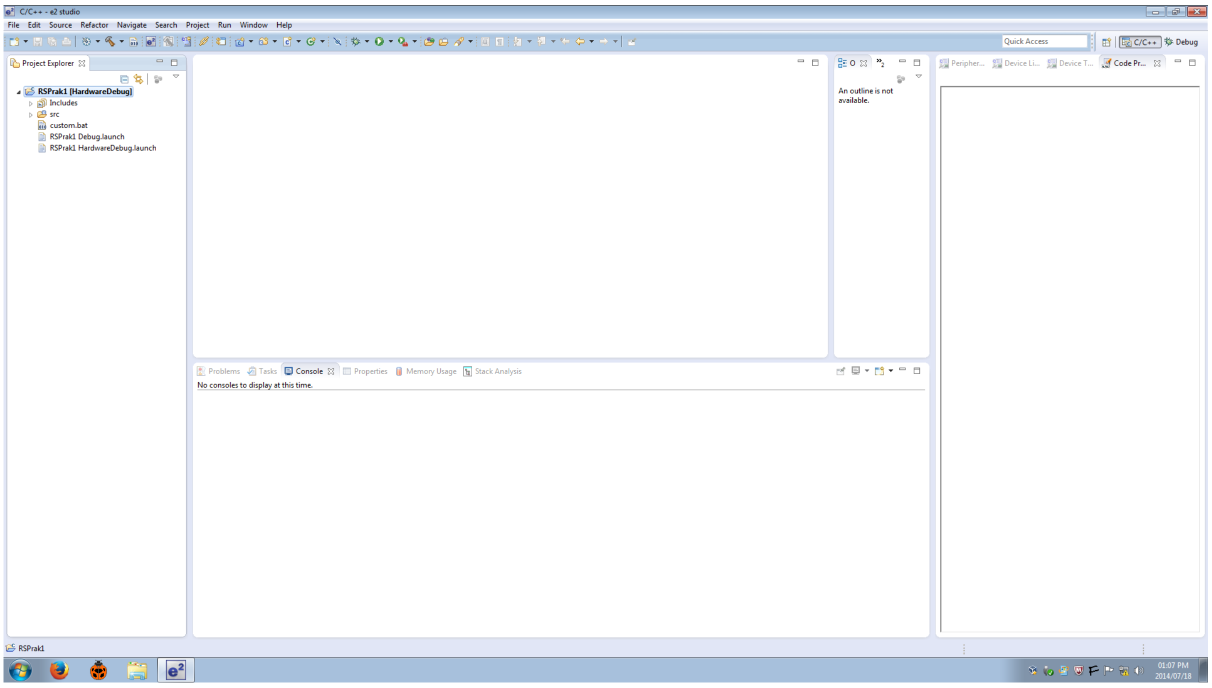
Multiplication code Generation Option : Use RL78 hardware multiply instructions for G13

None
Use emulation for multiplication
Use RL78 native multiply for G14
Use RL78 hardware multiply instructions for G13

Select the library source as **optimized**, and the type as **pre-built**. Click Finish, and click OK for the summary.

The code generator automatically generates the configuration for the peripherals of the microcontroller. This is useful if you do not wish to configure the microcontroller yourself, but we will be working with the registers directly.

The new project wizard will close showing the e2 IDE in front of you. If it does no look like this, go to **Window** ⇒ **Reset Perspective** ⇒ **Yes**. This will change the IDE to the standard layout.



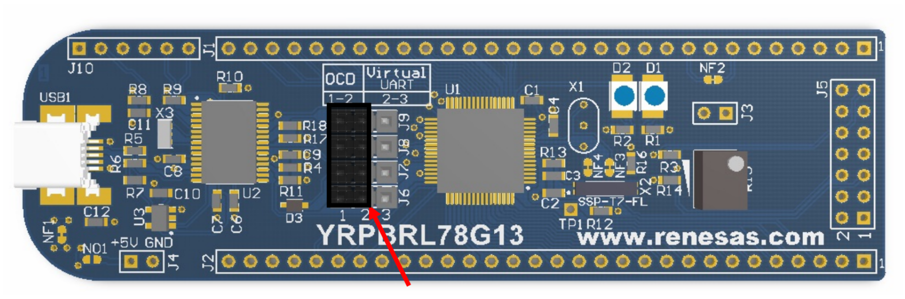
An outline is not available.

No consoles to display at this time.

Now your basic project should be configured. You should see the project explorer on the left, code editor in the middle and the console below it.

3 Program a microcontroller

On the left in the **Project Explorer**, go to the **src** directory, open the **RSPrac1.c** file. Note that this file will have the same name as your project. This is your main c file, and it contains your int main(void) function. Plug the RL78 development board into the computer's USB port. The jumpers on the development board should be in the 1-2 (OCD) position, to enable programming and debugging., Refer to the next table, from the development board's manual (page 16).



Jumpers	Configuration
J6	1-2
J7	1-2
J8	1-2
J9	1-2

Table 2: J6 to J9, On-Board Debug / Flash Programming Mode

When the mircocontroller starts, some startup code is run. This code configures all the required settings for the device to function. The code is generated for you by the IDE, and can be found in the reset_program.asm file. After the startup code is completed, the int main(void) function is called. This function should contain an endless loop, which keeps on running until the power is removed.

Copy the following code to your RSPrac.c file in e2 studio, replacing all the original code:

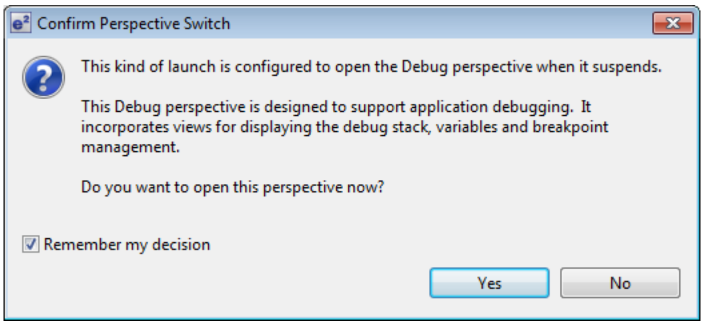
```
#include "iodefine.h"

int main(void)
{
    unsigned long counter = 0;

    /*
       Set port 7, pin 7 as an output
       See Hardware Manual page 803: 0=output, 1=input
    */
    PM7 = 0b01111111;

    // main loop
    while (1U)
    {
        /*
           Switch LED on
           Make port 7, pin 7 high
           See Hardware Manual page 187 -
           "These registers can be set by a 1-bit or 8-bit memory
            manipulation instruction."
        */

        P7_bit.no7 = 1;              // Shorthand to set pin high.
        //P7 = (P7 | 0b10000000);    // The better way of setting
                                     // a bit in a registry high.

        // delay
        for (counter=0;counter<800000UL;counter++);

        /*
           Switch LED off
           Make port 7, pin 7 low
           See Hardware Manual page 187
        */
        P7_bit.no7 = 0;              // Shorthand to set a pin low.
        //P7 = (P7 & 0b01111111);    // Or use the better way by reading,
                                // bitwise AND and saving the register again.

        // delay
        for (counter=0;counter<800000UL;counter++);
    }
}
```

This will turn the LED on the board off, delay for a while, turn the LED on again and delay again. This happens in an endless loop.

Now we need to compile this code. In the Project Explorer, **right click on the project** ⇒ **Build Configurations** ⇒ **Set active** ⇒ **Hardware Debug**. If you want to simulate the hardware you can choose the debug option.

Now build the project. Go to **Project** ⇒ **Build All**.

When compiling has finished, click the debug icon (The little bug next to the Run button). The IDE should now connect to the hardware and program it. If it gives a connection error, unplug the board and plug it back in. The IDE will switch to the debug perspective. If it asks for confirmation, select **Remember my decision** ⇒ **Yes**.


Confirm Perspective Switch

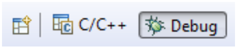
This kind of launch is configured to open the Debug perspective when it suspends.

This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

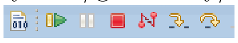
Do you want to open this perspective now?

☑ Remember my decision

Yes   No

From here you can step through the code, run it, or upload new code. To get back to the code editor, press the C/C++ button in the upper right hand corner. 

If you still get an error message, even after replugging the development board from USB, it might be that the drivers aren't loaded. Try the following steps: Start ⇒ Rightclick My Computer ⇒ Manage ⇒ Device Manager. Under ports there should be a COM port named Renesas. If not, there is likely an Unknown Device section with one device. Update the unknown device's drivers, manually chose driver, Ports, Renesas UART.

While in the debug perspective, click on the yellow/green Play/Pause button to start executing the code on the microcontroller, or to stop it again. 

Note that the IDE automatically adds a breakpoint at the start of your program, making the code Pause almost immediately after you tell it to execute. You should therefore click on the Play button a second time to make it execute indefinitely.

A breakpoint is a marker you add to a line of your code, telling the debugger to stop executing just before that line. This is useful to inspect the contents of variables and registers while debugging your code.

If your code has compiled and programmed successfully, and is executing on the microcontroller, the

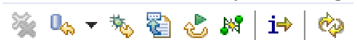
LED on the development board will be flashing. If the LED is not flashing, you will have to make sure all steps up to this point has been done correctly.

4 Functions in C

In C it is good practice to move code that is used a lot into functions. Because switching an LED on and off is one of these common things to do, we will move it into a function. This will help a lot for the readability and usability of our code.

Copy the following code into RSPrac1.c, compile, program and run it again. The LED on the development board should flash again.

```c
#include "iodefine.h"

void switchLEDon()
{
        P7=(P7|0b10000000);
}

void switchLEDoff()
{
        P7=(P7&0b01111111);
}

void delay(unsigned long time)
{
        unsigned long counter=0;
        for(counter=0; counter<time; counter++);
}

int main(void)
{
    /*
      This code is more readable than the previous one. Readability and
      reusability of the same code are two benefits of using funtions.
    */

    // Set Port 7, Pin 7 as output
    PM7 = 0b01111111;

    //main loop
    while (1) {
        switchLEDon();
        delay(80000UL);
        switchLEDoff();
        delay(80000UL);
    }
}
```

If you get an error message in the debug perspective, it might be necessary to reset the microcontroller. This can be done by clicking on the reset icon looking like the third icon from the left in this image:


5 In-line assembly

The task of the compiler is to translate C code into the correct assembly language for the microcontroller we are using. Assembly language is code words that can be directly interpreted by the logic gates on the microcontroller.

Assembly is complex and difficult to write by hand. Therefore higher level languages like C has been developed. The C compiler follows smart rules to translate C into assembly. These rules are quite generic so that it can be reused on a wide variety of devices. The assembly it generates are therefore not always the best.

In the following practicals we will be looking at writing assembly language by hand. It is a nice skill to have if you have to optimise code one day. It is also good to know if you have to debug a piece of code and need to interpret what the compiler did wrong.

As an example, the following three pieces of code do exactly the same thing:

C code	C compiler output	Handwritten assembly

`P7=(P7|0b10000000);`	`movw 0xffef0, #0xff07`
	`movw 0xffef2, #0xff07`
	`movw ax, 0xffef2`
	`movw 0xffef4, ax`
	`movw hl, 0xffef4`
	`mov a, [hl]`
	`mov 0xffef2, a`
	`and 0xffef2, #127`
	`movw ax, 0xffef0`
	`movw 0xffef4, ax`
	`movw bc, 0xffef4`
	`mov a, 0xffef2`
	`mov 0[bc], a`
		`movw bc, #0xff07`
		`mov a, [bc]`
		`or a, #10000000B`
		`mov [bc], a`

To see the assembly the C-compiler generated, one can rightclick on the number on the left of a line of C code and choose View Disassembly.

The following block of code does exactly what the previous C code you programmed onto the development board did, but the two lines of C that did the magic are now replaced by in-line assembly. Copy this code to your RSPRac1.c file, compile, program. Make sure the LED flashes again.

```
#include "iodefine.h"

void switchLEDon()
{
        /* The following block does the same as
                P7=(P7|0b10000000);
           but rewritten in assembly.
           | means bitwise OR
           To understand this piece of code you will need to
           search for the keywords (mov, or) in the Software Manual.

           0xFF07 is the memory mapped IO address of register P7.
           It is also called a special function register.
           See Hardware Manual page 138.
        */

        asm(";; Switch LED on         \n\t"
            "movw bc, #0xff07         \n\t"
            "mov a, [bc]              \n\t"
            "or a, #10000000B ;; Bitwise OR \n\t"
            "mov [bc], a              \n\t"
           );
}

void switchLEDoff()
{
        /* The following block does the same as
                P7=(P7&0b01111111);
           but rewritten in assembly.
           & means bitwise AND
        */

        asm(";; Switch LED off          \n\t"
            "movw bc, #0xff07           \n\t"
            "mov a, [bc]                \n\t"
            "and a, #01111111B ;; Bitwise AND \n\t"
            "mov [bc], a                \n\t"
           );
}

void delay(unsigned long time)
{
        /*
          We leave this piece of code in C for now,
          as it is complex to rewrite in asm.
        */
        unsigned long counter=0;
        for(counter=0; counter<time; counter++)
        {
                asm("nop"); // assembly for No Operation
        }
}

int main(void)
{
    // Set Port 7, Pin 7 as output
    PM7=0x7F;

    // main loop
    while (1) {
        switchLEDon();
        delay(80000UL);
        switchLEDoff();
        delay(80000UL);
    }
}
```

6 Seperate asm-file

Because writing in-line assembly makes your code very messy, it is better to have a separate file for only assembly language. The following two code listings show you how this is done and how functions written in ASM (assembly) can be called from within C.

You will learn this next week: All registers that are used inside your assembly function should first be pushed onto the stack to save any existing values inside them. At the end of your asm-function you should pop them from the stack again to restore the registers to their existing states. The reason for doing this is because your C compiler might be using the registers itself, and we do not want to interfere with it.

Rightclick on your project and create a new file. Call it LED.asm

Copy the following code to the new file.

```
.global _switchLEDon
.global _switchLEDoff

_switchLEDon:
;; Switch LED on

push bc           ;; Save the original contents of the two registers
push ax           ;; we use by pushing it to the stack.
                  ;; (push works with 16bit register pairs)

movw bc, #0xff07  ;; Save the 16bit address of P7’s register into
                  ;; 16bit register pair BC
mov a, [bc]       ;; Get the 8bit value stored at that address
                  ;; and store it in register A
or a, #10000000B  ;; Bitwise OR the value in A with 10000000
mov [bc], a       ;; Set the value in P7’s register to the value in A

pop ax            ;; Restore the contents of the two registers we used,
pop bc            ;; by popping it from the stack.
                  ;; (pop works with register pairs)
ret               ;; exit this function


_switchLEDoff:
;; Switch LED off

push bc           ;; Save the original contents of the two registers
push ax           ;; we use by pushing it to the stack.

movw bc, #0xff07  ;; Save the 16bit address of P7’s register into
                  ;; 16bit register pair BC
mov a, [bc]       ;; Get the 8bit value stored at that address
                  ;; and store it in register A
and a, #01111111B ;; Bitwise AND the value in A with 01111111
mov [bc], a       ;; Set the value in P7’s register to the value in A

pop ax            ;; Restore the contents of the two registers we used,
pop bc            ;; by popping it from the stack.
ret               ;; exit this function
```

Change your RSPRac1.c file by replacing the two LED functions with the following two lines:

```
//tell the compiler we defined these functions in another file
extern switchLEDon();
extern switchLEDoff();
```

Compile, program, and run your code. Does the LED still flash?

7 Homework Questions

Answer the following questions as good as you can and hand it in on `learn.sun.ac.za`. It should be a document saved as a PDF file. The deadline is 23:55 on 6 August 2015. If you can answer these questions, next week's practical will make much more sense.

You will likely need to reference chapter 1 of the RL78 Software Manual to answer all the questions.

1. Modify LED.asm to only contain one function which will toggle the state of the LED. In other words, if the LED is on, the function should switch it off, and if the LED is off the function should switch it on. To do this you can copy _switchLEDon, change the name to _toggleLED, and then only change the line saying "`or a, #10000000B`".

   Register P7 has 8bits. We want to change only bit 7 (most significant bit), without affecting the other 7 bits in the register. The two functions you were given to switch the LED on and off also took this into account.

   Hint: Look back at last semester's Computer Systems and specifically the boolean operators you used there. AND, OR, ...? Which one will do what we want in this case? (See "Wenk vir huiswerk 1.txt")

   You may provide only the modified line as answer to this question.

2. a) What type of architecture does the RL78 employ (Harvard, Von Neumann, ...)?

   b) What does this mean with regard to the memory space, number of system buses, and addressing on the RL78?

   c) Section 2.2.1 refers to mirror areas in memory. Is your answer in 2a still valid?

3. a) Does the RL78 employ reduced (RISC) or complex instruction set computing(CISC)?

   b) What does this mean with regards to number of clock cycles used per ASM mnemonic?

   c) For the following list of CPU's, do they use RISC or CISC instruction sets?

      i. AMD cpu's

      ii. Intel cpu's

      iii. ARM cpu's

4. What is the least number of clock cycles one ASM mnemonic will take on the RL78?

5. a) How many registers does the RL78 have that we can use to store temporary values during calculations?

   b) Why would we want to use registers to store temporary values, rather than RAM?

6. To answer this question you need to lookup what "little endian" and "big endian" is. Note that it is "**E**ndian", not "Indian". For a good description on these two, read "On Holy Wars and a Plea for Peace", which is available on `learn.sun.ac.za` under this week's practical.

   a) Is the RL78 little endian or big endian?

   b) Is your Windows desktop PC little or big endian?

c) Give an example where big endian would be the logical choice to use.

7. If the RL78 provides any, what are the ASM mnemonics for:

   a) adding two numbers?

   b) subtracting a number from another number?

   c) multiplying two numbers?

   d) dividing a number by another number?

   e) If the RL78 does not provide ASM mnemonics for all the actions listed above, how would you go about implementing it with the existing mnemonics.

8. What is the name of the CPU core used by the microcontroller on the RL78 development board we use?

9. a) According to the Hardware Manual, to set or clear the state of Port 7, Pin 7, we need to set or clear bit 7 in register P7. The Hardware Manual specifies that P7 has the address FFF07H (page 138), but in the LED.asm file you were provided we used the address FF07H. What happened to the higher 4 bits of the address?

   There might be a couple of reasons for this, but inspecting exactly what is done in the code points one to what is written in section 4.2.7. Note what register we used to store the address and why section 4.2.6 is not applicable.

   b) Why was it necessary to copy the data out of register P7 first before performing the AND and OR on it? Can't we just perform the operation directly on register P7? (The answer hides in the list of available formats for AND and OR on pages 47 and 48 of the software manual.)