Exercises 2IN62.1

Today you will get acquainted with the CodeWarrior development environment, which we will use throughout the course. It allows you to write code, compile the code for the MC9S12XF512 controller and then run the executable on either the EVB9S12XF512E development board or within the MC9S12XF512 simulator. Today we will work with the simulator and perform some measurements.

1.1 Installing CodeWarrior

First let us setup CodeWarrior.

1. Download and install the CodeWarrior IDE from `http://cache.freescale.com/lgfiles/devsuites/HC12/CW_HC12_v5.1_SPECIAL.exe`

1.2 Building and running a CodeWarrior project

To develop under CodeWarrior you first need to create a project.

1. Navigate to the `exercises1` directory.
2. Open the project file `EVB9S12XF.mcp`.

First we need to compile the program (illustrated in Figure 1):

3. Open the `main.c` file, which resides in the `Sources` group. The `main()` function contains the program. This is where you will be writing your code in later exercises.
4. Select the `Full Chip Simulation` as target.
5. Build and run the project.

At this point the debugger window will appear, shown in Figure 2. It allows you to control the execution of the program.

6. The MC9S12XF512 controller contains two cores: the main HC12 core, and an auxiliary XGATE core. We will be working only on the HC12 core.

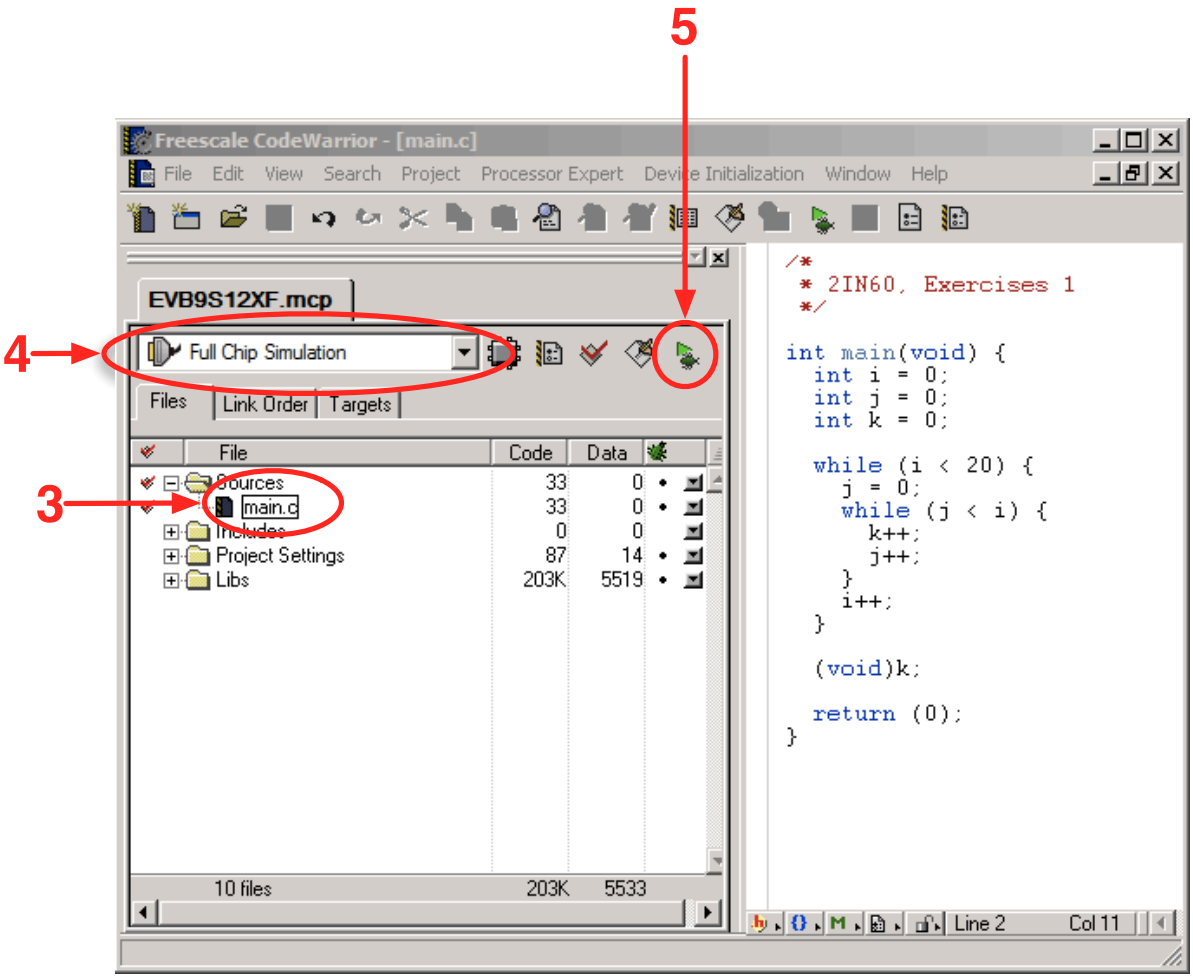

Figure 1: Building a project in CodeWarrior. The red numbers refer to the corresponding steps.

7. The control bar allows to control the execution of the program. The green arrow executes the program until the end, or until a breakpoint (more on these later). The black arrow in a red circle stops and resets the execution. The other buttons allow finer control, e.g. to execute a single instruction or to step out of a function call.

8. In the `Assembly` window you can see the low-level processor instructions which are currently executed.

9. In the status bar on the bottom of the window you can see the current cycle count or the current elapsed time (clicking on the frequency field next to the cycle count allows to switch between the two).

10. Start the execution by pressing the green arrow. The program will execute and stop almost immediately, since this particular program does very little. You will notice that the `Assembly` window is now pointing to a different instruction. Also, the cycle count has changed.

11. Reset the simulation and run it again a few times to see how the instruction pointer and the cycle count change.

1.3 Disassebly

CodeWarrior provides a handy view showing the disassembled C code, i.e. the mapping of the C code to the assembly generated by the compiler. It is very useful for analyzing the code to better understand which micro controller instructions will actually be executed. The assembly corresponding to the `main()` function is illustrated in Figure 3.

1. Disassemble the `main.c` file by right-clicking within its Source window and then choosing Disassemble.

1.4 Measuring the cycle count during runtime

Now that you are able to compile and run a program in the simulator, we will measure the execution time of the outer loop in the program. We will measure how long it takes to execute the outer loop by adding two breakpoints: one at the `int k = 0;` instruction and one at the `return (0);` instruction. When the execution reaches an instruction marked with a breakpoint it stops *just before executing the instruction*, allowing us to read the cycle count.

Note that you have to add the breakpoints in the debugger window. Clicking on a line in the CodeWarrior window does not add an actual breakpoint.

1. Make sure that the `main.c` file is shown in the `Source:1` window. If this is not the case, then right-click in the `Source:1` window, click on `Open Source File...`, and select `main.c`.

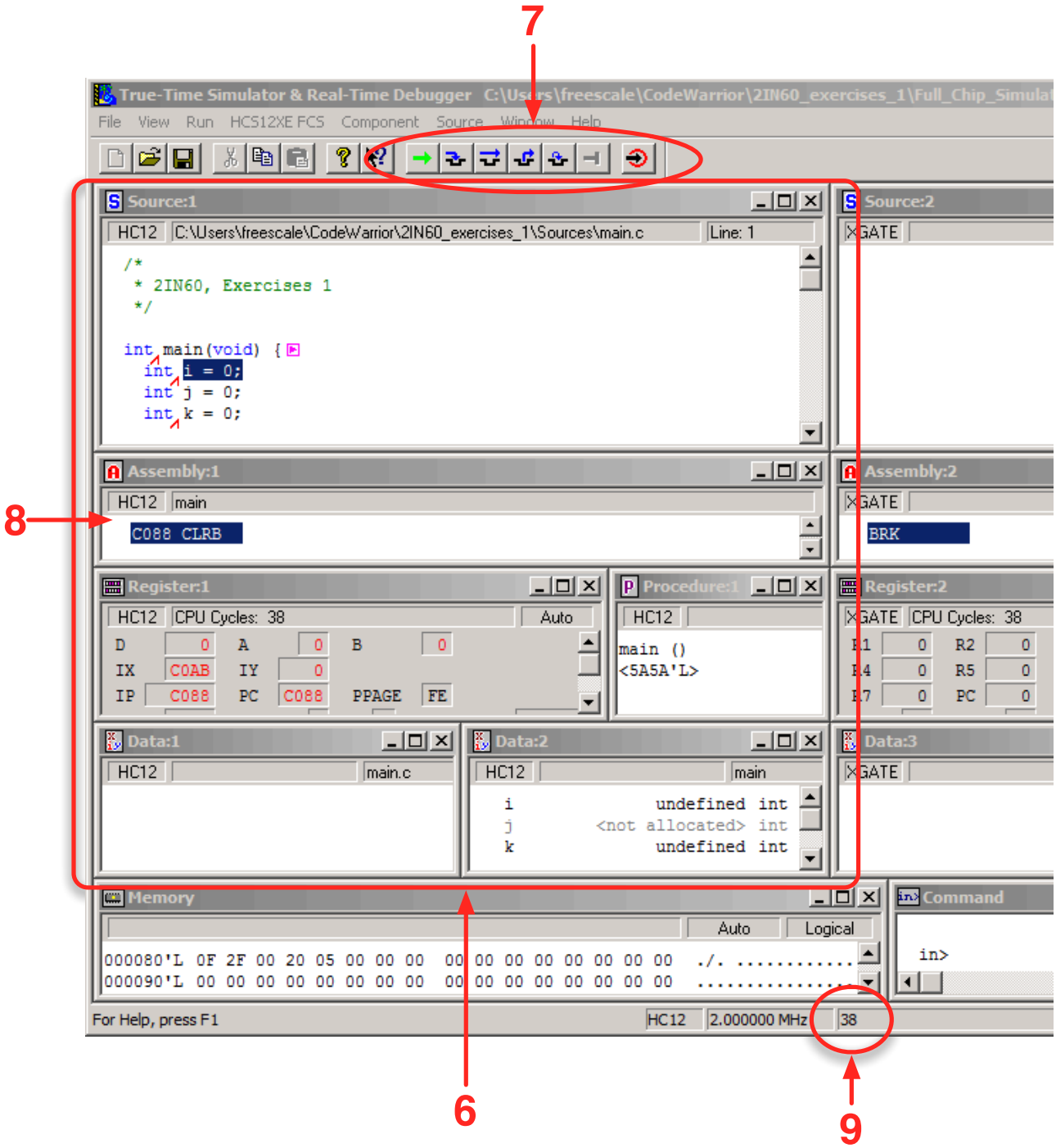

Figure 2: Debugging a program in the simulator.

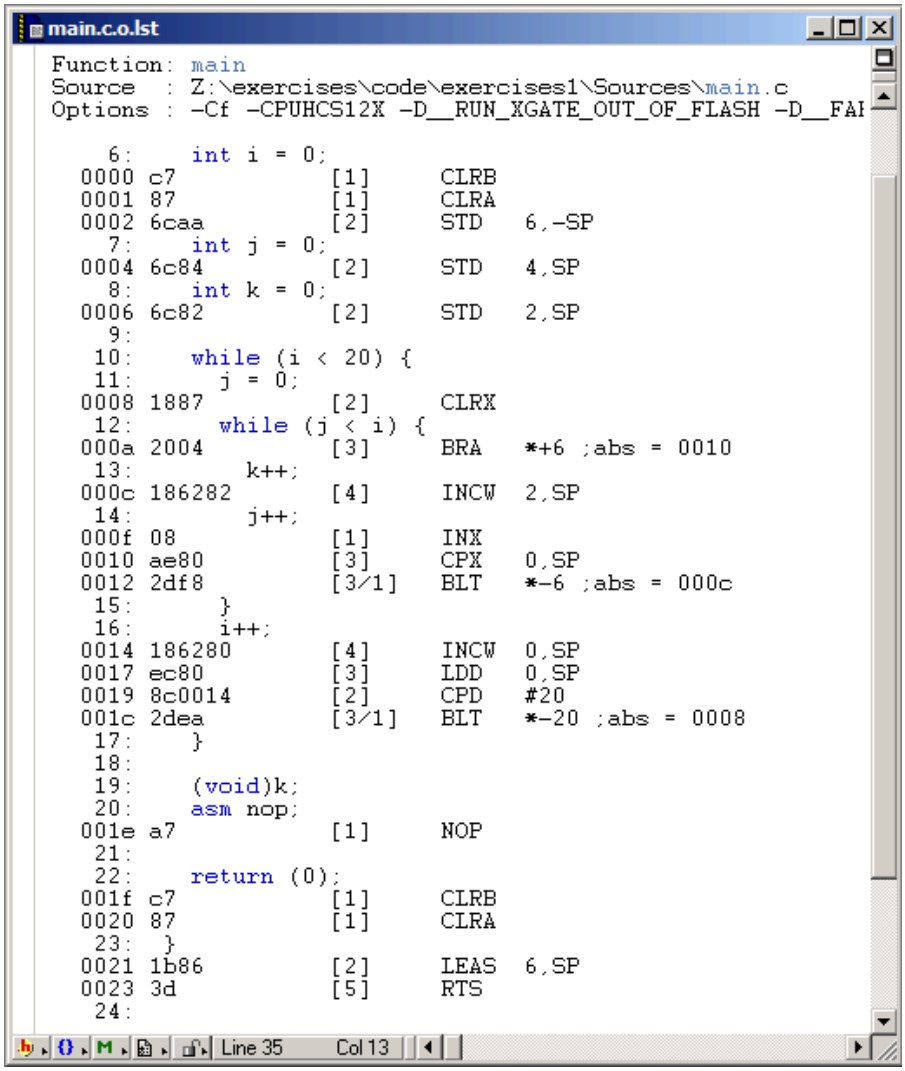

Figure 3: Disassembly of the `main.c` file.

2. Right-click on the `int k = 0;` line and click on `Set Breakpoint`.

3. Right-click on the `return (0);` line and click on `Set Breakpoint`.

4. Run the simulation. You will notice that the execution stops at `int k = 0;`. Write down the cycle count.

5. Press the green arrow to continue executing until the next breakpoint (or the end of the program). The execution stops at `return (0);`. Write down the cycle count. **How many cycles did it take to execute the loop?**

6. **Why did we place the breakpoint at `int k = 0;` and not at `while (i < 20)`? What are the consequences for the measurements?**

7. Rather than writing down the cycle count at the first breakpoint and then subtracting it from the cycle count at the second breakpoint, we can reset the cycle count at the first breakpoint by double-clicking on the cycle count.

1.5 Changing parameters

In this exercise we will modify the program.

1. Close the debugger.

2. Open the `EVB9S12XF.mcp` project (if it is not already open) and navigate to `main.c`.

3. Inside the `main()` function, the outer `while` loop will iterate 20 times. Change it so that it iterates only a single time.

4. Build and run the project.

5. **Measure the cycle count of the execution of the outer loop for different numbers of iterations, ranging from 1 to 100, in steps of 20 (i.e. 1, 20, 40, 60, 80, 100)**

1.6 Measuring the time during runtime

Next to measuring the cycle count, we can also measure the time.

1. In the status bar on the bottom of the debugger, click on the field on the left of the cycle count showing a frequency in MHz. A window appears in which you can select `True Time` rather than `CPU cycles`. The time is shown in milliseconds.

As mentioned earlier, the debugger allows also to step through the simulation by executing individual instructions.

2. Run the simulation and break at the first line of `main.c`.

3. You can perform a single step by clicking on the `Step Over` button (second to the right of the green arrow).

4. **Set the outer loop count to 2 and step through the entire program, writing down the cycle count and the time for each step (i.e. the execution of each instruction). When looking at your collected data, what do you observe?**

1.7 Inspecting variables during runtime

In this exercise we will inspect the values of variables during runtime.

1. **Assume the outer loop iterates n times (initially we had $n = 20$). Give an equation for the number of inner loop iterations in terms of n.**

2. You can inspect the value of any variable during runtime by setting a breakpoint and reading it in the `Data:1` window (for global variables) and `Data:2` window (for local variables).

   Note that the compiler, which translates your program into machine code, will try to optimize the generated code. In particular, if it observes that the value assigned to a variable is not used anywhere, or if the value of another variable can be reused, then assignments to that variable are removed from the code. Consequently, you may find that some variables are not visible in the `Data:1` and `Data:2` windows.

3. **Which variable should you inspect to see the total number of inner loop iterations?**

4. **Where should you place the breakpoint?**

5. **Compare your measured count of inner loop iteration with those you obtain using your equation from step 1, for $n \in \{1, 20, 40, 60, 80, 100\}$**