Objectives

- To write a small application for the Motorola MC9S12DP256B/C microcontroller that reads the A/D converter (ADC) and displays the converted value on the on-board LEDs of the Dragon12 development board
- To learn about interrupts and the use of interrupt service routines (ISR)

Introduction

Digital signal processing (DSP) and digital control both rely on the timely evaluation of the system equations (e.g. the difference equation of a filter or any other transfer function). A typical cycle consists of

(1) Reading the ADC
(2) Evaluation of the system equations
(3) Writing the result(s) to the DAC

The implement of such a cycle on a microcontroller or a digital signal processor requires some form of timing engine. This can be achieved in a variety of ways. Advanced systems often employ the services of a real-time operating system (RTOS) which usually allows for multi-tasking and the scheduling of tasks at predefined sample rates. Real-time operating systems are particularly useful when working with multi-rate control systems. However, many digital control systems are designed for a single sample rate. In this case, a simple *timer interrupt service routine (ISR)* can be used to provide the required time base.

With this laboratory session we begin to investigate how this can be done. We will write a number of small test applications to learn about A/D converters, interrupts and how these interrupts can be serviced. All programs will be simulated, followed by a run on the Dragon12 development board.

Note: During this exercise you will frequently have to find information from sources such as the user manual of the MC9S12DP256B/C or the data sheet of the D/A converter. It may be a good idea to open this user guide now…

Accessing the A/D converter unit of the MC9S12DP256B/C

Figure LM3-1 shows a block diagram of the 8/10-bit A/D converter unit (ATD_10B8C) of the MC9S12DP256B/C. Central to this unit are the *sample and hold amplifier (SHA)* and the *successive approximation register (SAR)*. The unipolar analogue voltage to be converted (V_{in}) is selected by the multiplexer from pins AN0 – AN7. V_{in} must fall within the range from (0 V =) $V_{SSA} \leq V_{in} \leq V_{DDA}$ (= 5 V). The digital value of the successive approximation register is converted to an analogue voltage (V_{DAC}). This D/A conversion is based on a reference voltage range from (0 V =) V_{RL} to V_{RH} (= 5 V). The voltage V_{DAC} is then compared to the output voltage of the sample and hold amplifier (V_{SAR}). Depending on the result of this comparison the value within the successive approximation register is doubled ($V_{DAC} < V_{SAR}$) or halved ($V_{DAC} > V_{SAR}$). Each comparison decides the state of one bit of the SAR. A total number of 10 comparisons are required to convert an analogue voltage with to an overall accuracy of 10 bit. Note that the ADC is synchronized with the bus clock of the microcontroller. This ensures that conversions always begin in synchrony with a CPU cycle.

Study the ADC user manual (S12ATD10B8CV2.pdf) to find the nominal conversion time of a single 10-bit A/D conversion (hint: you can *Edit → Search* for keywords such as *frequency, conversion time, sample time*, etc.). What is the maximum ATD clock frequency? What is the default ATD clock frequency (reset value)? How long would you expect a 10-bit conversion to last for when the ADC unit is driven at the default ATD clock frequency? In which register can you expect to find the conversion result? What register(s) is/are used to configure the ADC for a single conversion? How can you check if a conversion is still on-going or has finished?

Note: The bus clock of the MC9S12DP256B/C is controlled by a built-in *Phase-Locked Loop circuit (PLL)* to a constant 24 MHz.

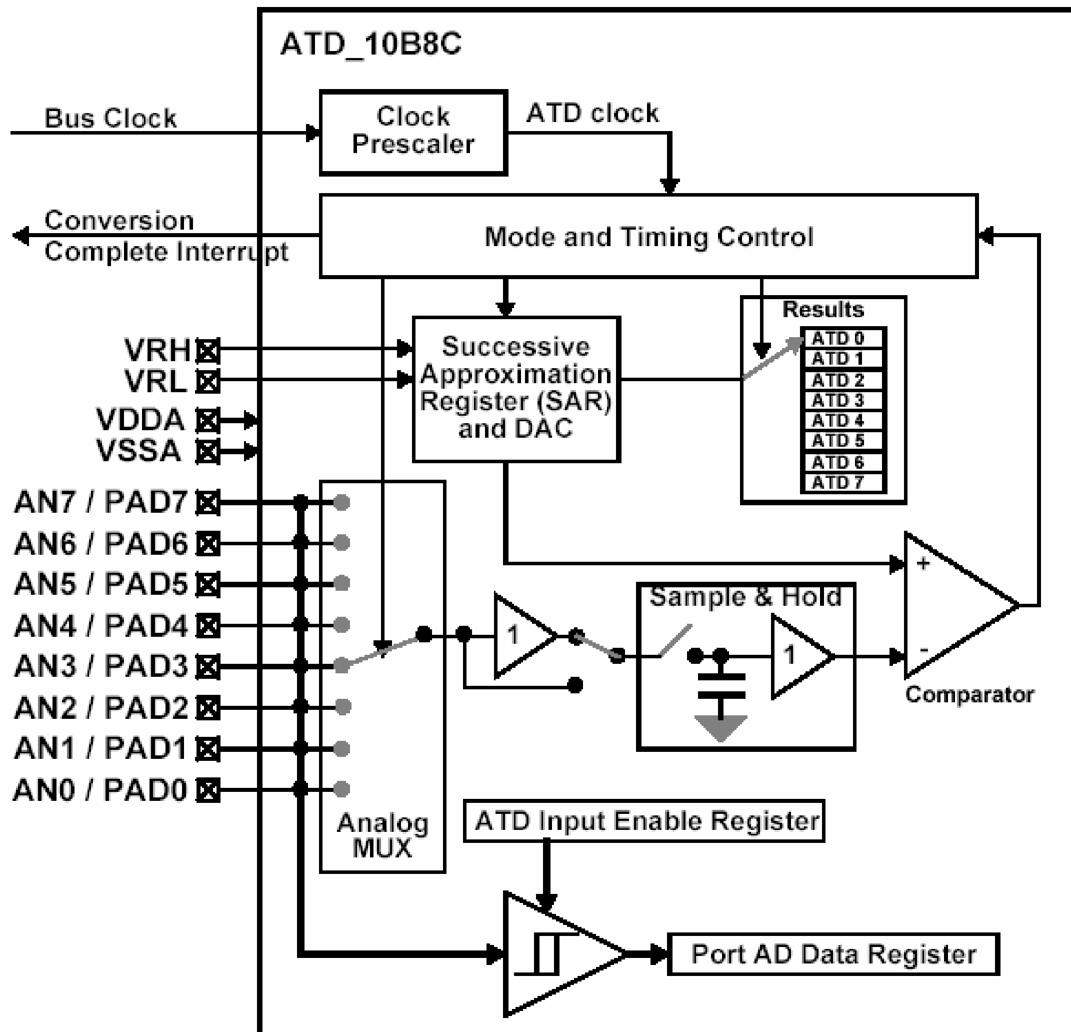

Figure LM3-1 ADC unit of the MC9S12DP256B/C

Now that you have familiarised yourself a little with the function of the ADC unit you should be able to write a small program that reads an analogue voltage from channel AN07 and displays the result of the conversion on the LEDs connected to port B. Channel AN07 has been connected to the on-board potentiometer (0 V … 5 V). The display should be implemented as *linear level gauge:* At the minimum conversion value (0) all LEDs should be off, at the maximum conversion value (0x3FF = 1023) all LEDs should be on, in between: number of LEDs lit is proportional to converted value.

Hints:

(1) Begin with creating a new project based on stationery *Dragon12_flat*. Save this empty project under the name *myADC.mcp*.

(2) This exercise will make use of 3 special function registers: A control register (select channel, configure ADC), a status register (test for end-of-conversion – don't use interrupts yet) and a data register (conversion result).

(3) It is good practice to write a separate source code file for each hardware unit you use (e.g. one for the ADC, one for serial communication, one for the timer, etc.). Functions which are defined in any of these source files can be made available to the *main* module via an associated header file (see Figure LM3-2). It is recommended that you create the following pair of files: *adc.c* contains the definition of all ADC related functions (e.g. *ADC_Init*, *ADC_Read*); the corresponding header file *adc.h* declares these functions by listing their *function prototypes*. You will have to modify your main program *(main.c)* to include the new header file.

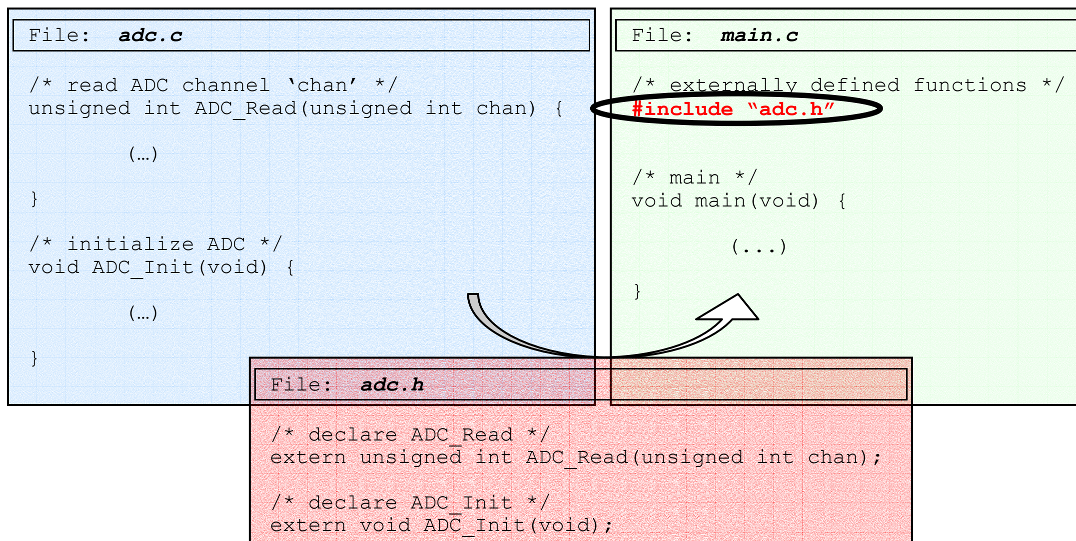

Figure LM3-2 Separating main program and user functions

(4) Having created *ADC.c* and *ADC.h* you will have to add both files to the *Sources* group of your CodeWarrior project. This can be done by right-hand mouse clicking onto *Sources* and selecting *Add Files…* . Upon adding the source files your project browser should look as shown in Figure LM3-3.

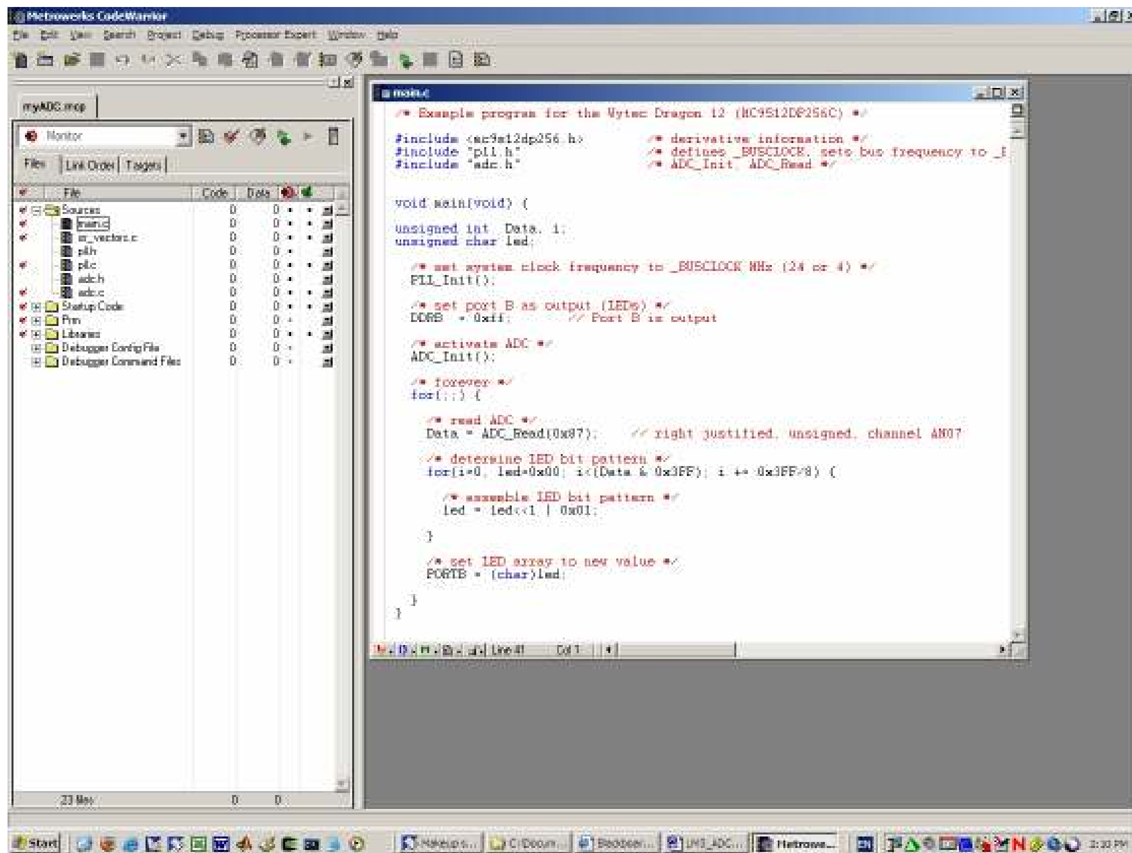

Figure LM3-3 Adding files *adc.c* and *adc.h* to project group *Sources*

Compile your code and download it to the Dragon12 board (target: *Monitor*). Step through your code to verify its proper operation. Click on the 'Start/Continue' button (green arrow) to run the program at full speed. Turn the potentiometer and observe that the converted value is instantly displayed on the LEDs. Click on 'Stop' to halt execution. The debugger might complain about the 'loss of communication'. Should this be the case, simply reset the board, close down the debugger and start it again.

Exit from the debugger. Set the *mode switch* (SW7) to *RUN* (the mode indicator LED should change from EVB to EEPROM). Operate the *RESET* button of the Dragon12 board. The program should now run uninterrupted.

ADC interrupt

In this section we will modify our ADC program to trigger an interrupt every time the ADC has finished converting a value. This way, we can keep the ADC running in the background and use an *interrupt service routine (ISR)* to store the result in a *global variable*. The main program will read this variable and display its value.

Close down the CodeWarrior IDE. Open the *Windows File Explorer* and go to the folder in which you stored your ADC project. Duplicate (copy) the entire ADC project folder *(myADC)* and rename the copied folder from *'Copy of myADC'* to *'ADC_interrupt'* (Figure LM3-4). For consistency, you may also want to change the project file name (*.mcp) of the newly created project (i. e. the one in folder *myADC_interrupt*) from *myADC.mcp* to *myADC_interrupt.mcp*. This is not really required, but it may help keeping things organised. Finally, delete the sub-folder *myADC_Data* of the new

project (i. .e *myADC_interrupt\myADC_Data*). The new project is now ready to be worked on.

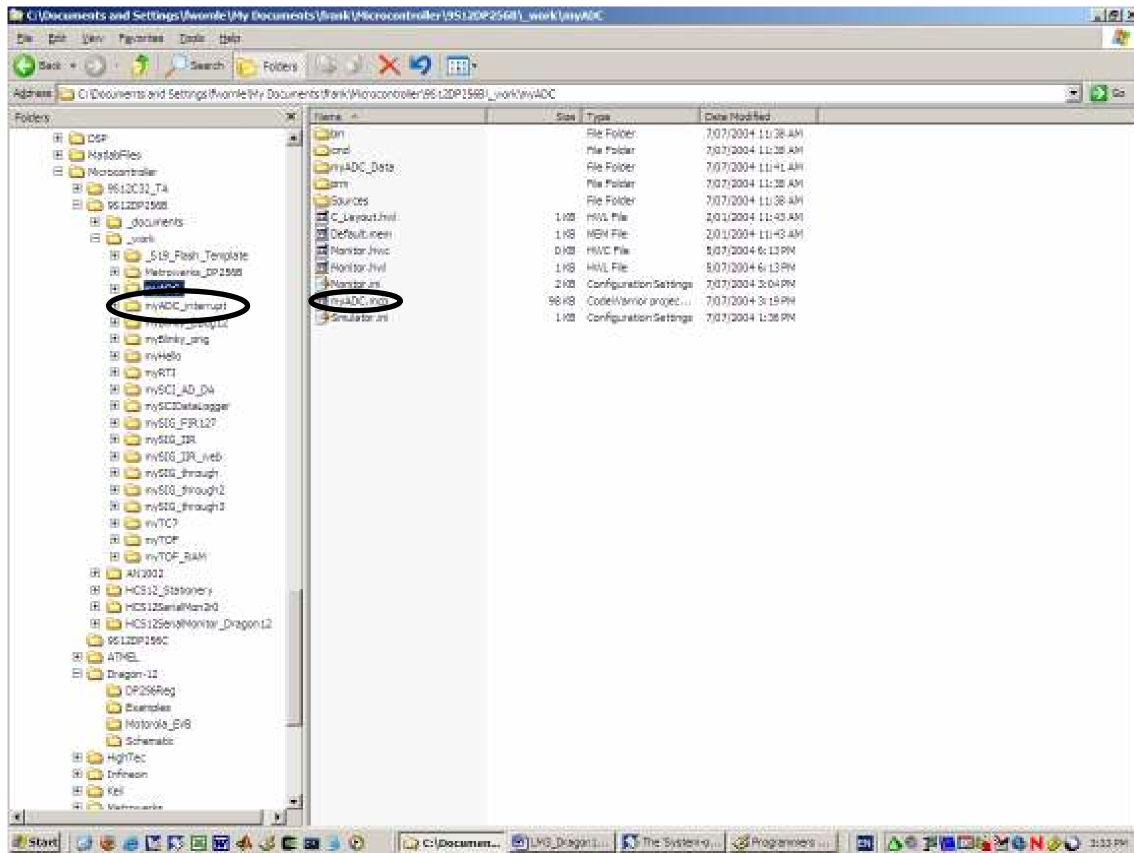

Figure LM3-3 Duplicating a project

Launch CodeWarrior by double clicking onto the project file *myADC_interrupt.mcp*. Open the source file of the A/D converter *(adc.c)* and find the initialisation function *ADC_Init*.

In its present form, *ADC_Init* consists of 2 instructions:

```
ATD0CTL2 = 0x80; // enable ADC
// bit 7 ADPU=1 enable
// bit 6 AFFC=0 ATD Fast Flag Clear All
// bit 5 AWAI=0 ATD Power Down in Wait Mode
// bit 4 ETRIGLE=0 External Trigger Level/Edge Control
// bit 3 ETRIGP=0 External Trigger Polarity
// bit 2 ETRIGE=0 External Trigger Mode Enable
// bit 1 ASCIE=0 ATD Sequence Complete Interrupt Enable
// bit 0 ASCIF=0 ATD Sequence Complete Interrupt Flag

ATD0CTL4 = 0x05; // configure conversion
// bit 7 SRES8=0 A/D Resolution Select
//               1 = 8 bit resolution
//               0 = 10 bit resolution
// bit 6 SMP1=0 Sample Time Select
// bit 5 SMP0=0 2 clock period
// bit 4 PRS4=0 ATD Clock Prescaler divide by 12
// bit 3 PRS3=0 ATD Clock Prescaler
// bit 2 PRS2=1 ATD Clock Prescaler
// bit 1 PRS1=0 ATD Clock Prescaler
// bit 0 PRS0=1 ATD Clock Prescaler
```

To instruct the ADC to trigger an end-of-conversion interrupt we will have to set the *ASCIE* flag in ATD0CTL2 (ATD Sequence Complete Interrupt Enable). Make the required code changes to enable interrupts.

Find the end-of-conversion function *ADC_Read.* In its current form, *ADC_Read* first starts a new conversion sequence (by writing the channel number to ATD0CTL5). It then *waits* for the conversion to be complete (Conversion Complete Flag *CCF0* in ATD0STAT1 goes high) before returning the 10-bit result from data register ATD0DR0.

```
//******** ADC_Read ***************
// perform 10-bit analog to digital conversion
// input: chan is 0 to 7 specifying analog channel to sample
// output: 10-bit ADC sample (left justified)
// analog input    left justified   right justified
//  0.000              0               0
//  0.005           0040               1
//  0.010           0080               2
//  1.250           4000             100
//  2.500           8000             200
//  5.000           FFC0             3FF
// uses busy-wait synchronization
// bit 7 DJM Result Register Data Justification
//       1=right justified, 0=left justified
// bit 6 DSGN Result Register Data Signed or Unsigned Representation
//       1=signed, 0=unsigned
// bit 5 SCAN Continuous Conversion Sequence Mode
//       1=continuous, 0=single
// bit 4 MULT Multi-Channel Sample Mode
//       1=multiple channel, 0=single channel
// bit 3 0
// bit 2-0 CC,CB,CA channel number 0 to 7
unsigned short ADC_Read(unsigned short chan){

  ATD0CTL5 = (unsigned char)chan;           // start sequence
  while((ATD0STAT1&0x01)==0){};             // wait for CCF0
  return ATD0DR0;

}
```

Waiting for the ADC to finish a conversion seems a waste of precious CPU time. Bear in mind that the microcontroller is supposed to evaluate a filter equation or a transfer function and feed the result to an external DAC *in the shortest possible time*. So, the smaller the number of avoidable delays we built into this cycle (ADC → evaluation → DAC) the better. The ADC delay can be avoided using the end-of-conversion interrupt: Using this technique, the ADC unit can work *in parallel* with the CPU, only interrupting the main program for a very short time in order to transfer the conversion result from the ADC data register (ATD0DR0) to memory. This takes significantly less time than waiting for an entire conversion to be finished.

We now have to turn function *ADC_Read* into an *interrupt service routine (ISR)* and install its starting address in the interrupt vector table. An ISR differs from a regular sub-routine only by the additional declarative keyword *__interrupt* and the fact that there are no call-up parameters and no return values. Change the definition of sub-routine *ADC_Read* to the following ISR:

```
//******** ADC_ISR ***************
// store conversion result in ADC_Data and restart conversion
__interrupt void ADC_ISR(void) {

    (...)

}
```

From the user manual of the MC9S12DP256B/C find the vector number of the end-of-conversion interrupt of the ADC unit *ATD0* (see chapter 5 of the user manual). We will now have to install our ISR in the vector table of our project (source file *isr_vectors.c*). Open this file and replace the placeholder vector *UnimplementedISR* of the ATD0 interrupt by the name of our interrupt service routine *(ADC_ISR)*.

These two modifications turn a regular sub-routine into an interrupt service routine. However, if we were to compile this program right now, we would get a compiler error message claiming an *unresolved external reference* to ADC_ISR. This is because our ISR has been defined in source file *adc.c*, whereas we use it in *isr_vectors.c*. To avoid this error we will have to reassure the compiler that ADC_ISR is indeed defined elsewhere. This means that *isr_vectors.c* has to be extended to include an 'external declaration' of ADC_ISR:

```
extern void __interrupt ADC_ISR(void);
```

It is good programming practice to place this declaration in header file *adc.h* and include this file in *isr_vectors.c*. That way, all ADC related functions can be found in *adc.c* and/or *adc.h*, an advantage you will learn to acknowledge as soon as a project comprises of several 10s of source files…

In summary, your project should be structured somewhat like this now (Figure LM3-5):

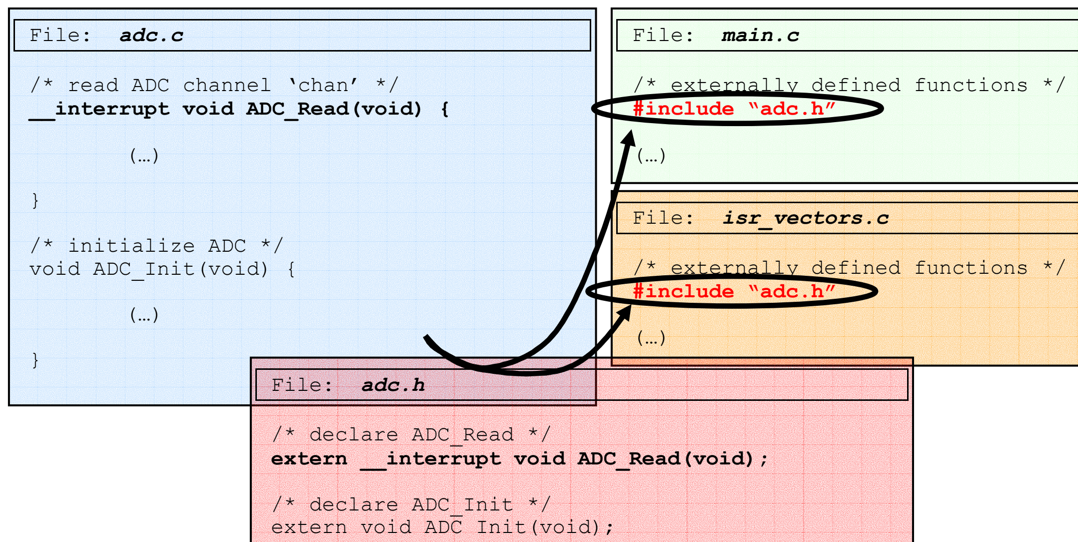

Figure LM3-5 Logical project structure

To complete the conversion from a regular sub-routine to an interrupt service routine we will have to modify the way data is transferred from the ADC data register to memory. Previously, *ADC_Read* took care of this by polling the conversion complete flag CCF0 (see code fragment above). In our new approach we have automated this process. We no longer have to check for an end-of-conversion flag – our interrupt service routine *ADC_ISR* is invoked automatically whenever a value has been converted. We therefore can (and have to) remove the test for CCF0 from the body of the ISR. Furthermore, we no longer *return* the conversion result to a calling function (remember: an ISR does not get called explicitly). Instead the contents of the data register (ATD0DR0) will have to be moved to a global variable. The reason why this variable has to be global is simple: We are accessing this variable from within 2

different functions. Values are written to it in *ADC_ISR*, while they are read from it in *main*. Defining the result variable in either one of these two functions would hide it from the other one. This data exchange via global variables is a general technique which is used with all interrupt service routines you may ever come across.

Complete the transition from *ADC_Read* to *ADC_ISR* by modifying your code in accordance to the explanations given above. Your ISR might end up looking somewhat like this:

```
//******** ADC_ISR ***************
// store conversion result in ADC_Data and restart conversion
__interrupt void ADC_ISR(void) {

  /* return value in global variable 'ADC_Data' */
  ADC_Data = ATD0DR0;
  ATD0CTL5 = 0x87;          // start sequence, channel AN07,
                            // result right-aligned

}
```

Note: As mentioned before, it is good programming style to define the global variable (say, '*unsigned int ADC_Data')* in source file *adc.c*. This maintains the strict separation of ADC related functions from general purpose functions. However, sticking *ADC_Data* in *adc.c* implies that we also have to extend the associated header file *adc.h* by an external declaration of this variable. That way the compiler won't freak out when it tries to compile *main.c* in which we use *ADC_Data* without explicitly defining it. Including header file *adc.h* now reassures the compiler that *ADC_Data* is defined elsewhere. It would also be good to initialise this variable to zero in function *ADC_Init*.

Confused? Well… have a look on myUni at:

Mechatronics IIIM → Course Documents → Tutorials → 9S12 → adc_interrupt

Compile your program and load it into the debugger/simulator (target: *Simulator*). Step over the first few instruction until you come to the display of the converted value. What do you observe? Does the program behave as expected?

Open file *adc.c* by right-hand mouse clicking inside the *Source* window of the debugger and selecting *'Open Source File… '* (Figure LM3-6).

Place a breakpoint onto the first line of the interrupt service routine *ADC_ISR*. This can be done from the *Debug* menu or by right-hand mouse clicking onto the first line of ADC_ISR and selecting *Set Breakpoint* (Figure LM3-7).

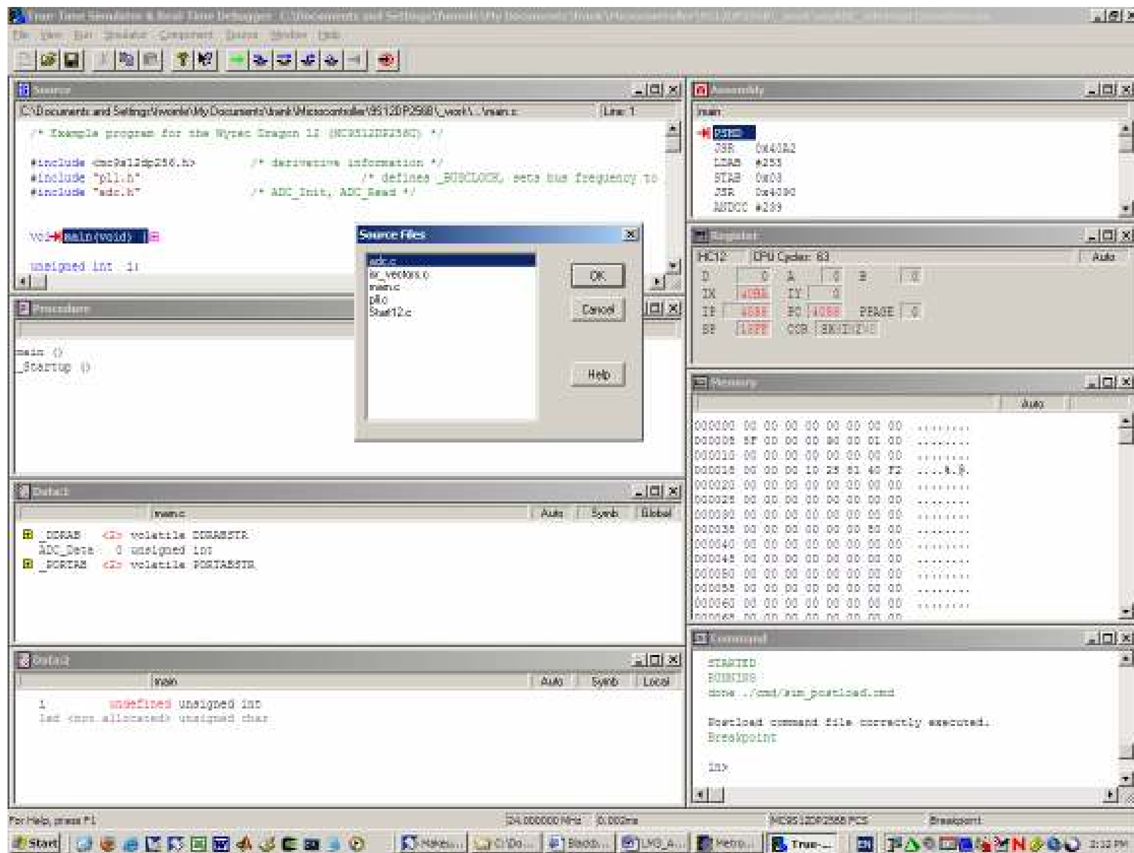

Figure LM3-6 Opening source file *adc.c*

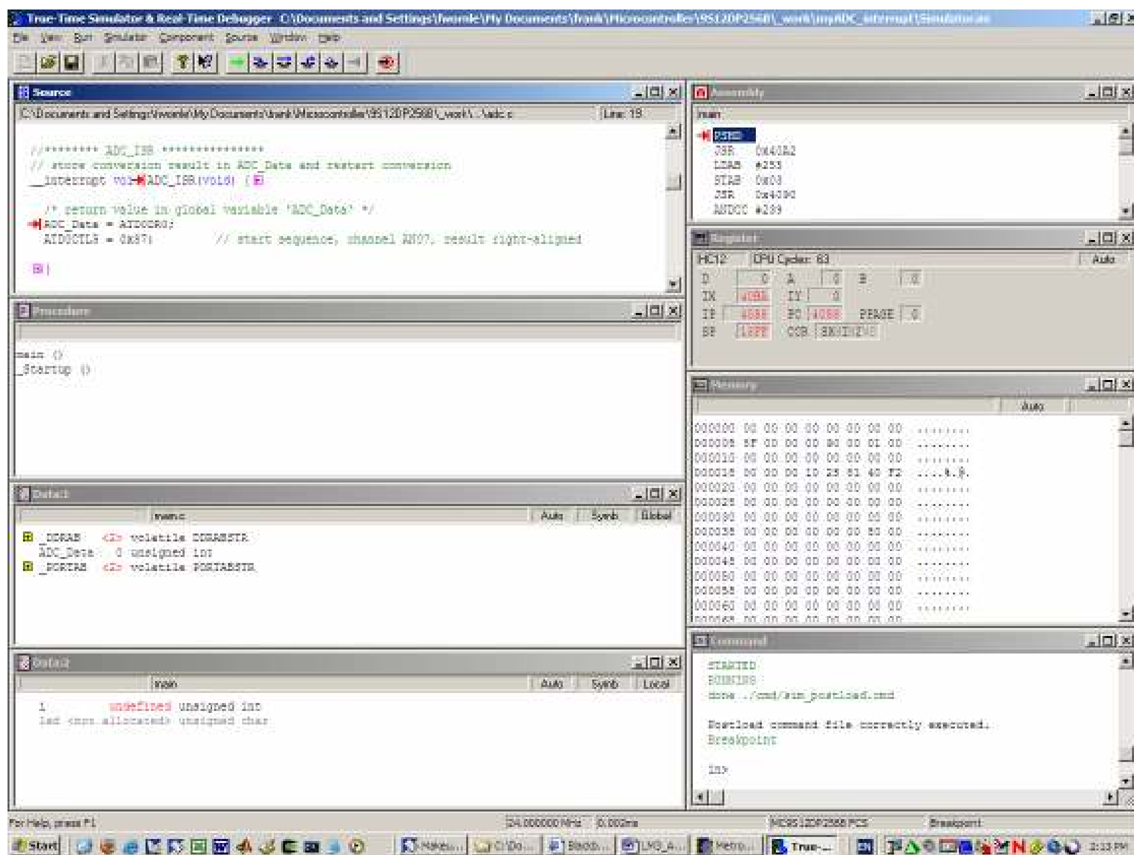

Figure LM3-7 Placing a breakpoint inside *ADC_ISR*

Run the program by clicking onto toolbar button 'Start/Continue' (green arrow). Observe that the CPU is running (CPU cycles are counting up in *Register* window). However, the program doesn't seem to do much – the breakpoint inside *ADC_ISR* is never reached. What could be the reason for this?

Halt the CPU (red 'Halt' button of the toolbar). Source file *main.c* reappears with the program stuck somewhere within the display. Re-open source file *adc.c* and scroll down to our interrupt service routine *ADC_ISR*. Notice the second line of this ISR. This re-starts the ADC after the previous result has been stored in *ADC_Data*.

```
/* return value in global variable 'ADC_Data' */
ADC_Data = ATD0DR0;
ATD0CTL5 = 0x87;         // start sequence, channel AN07,
                         // result right-aligned
```

Does this explain to you why this ISR is never triggered? Exit from the debugger and return to the CodeWarrior IDE. Modify function *ADC_Init* to ensure the ADC is actually running. Re-compile and re-start the debugger/simulator. Place a breakpoint inside *ADC_ISR* (as before) and run the code. What do you observe? Does the code behave as expected now?

It looks like we still haven't resolved the problem: The interrupt service routine remains inactive. Click on the 'Halt' button to stop code execution. Click on 'Reset', then 'Start/Continue'. The simulator should run through the code until it reaches the breakpoint at the beginning of *main*. Locate the *Condition Code Register (CCR)* of the CPU. What seems to be the state of the *Interrupt disable* flag? Double-click onto this flag (I) to toggle its current state, then continue running the code (green arrow). Does this resolve the problem?

Exit from the debugger and modify your source code to *allow interrupts to happen*. To alter the state of the *Interrupt disable* flag within the Condition Code Register of the CPU you can use the following inline assembler directives:

```
asm cli                  // clears the interrupt disable flag
asm sei                  // sets the interrupt disable flag
```

Which one of these instructions would you need to add to your code? Where would you place it? Once you have modified your code, re-compile and download to the Dragon12 board. Run your code at full speed ('Start/Continue', green arrow) and verify that the LEDs properly display the acquired value.

Concluding remark

The CodeWarrior simulator seems to struggle with the simulation of interrupts. You may have noticed that once an interrupt has been triggered once we immediately re-enter the ISR as soon as we get to its end. This strange behaviour might have to do with the way we have configured the simulator. Possibly we have stumbled across a bug in the program… Fortunately, the debugger seems to work when used in conjunction with the monitor program on the Dragon12. You should therefore always prefer the actual Dragon12 board over its simulation, at least when working with interrupts.

Summary – Programming interrupts

(1) Create an interrupt service routine (ISR). This is a sub-routine of type:

```
__interrupt void <ISR_name> (void)
```

(2) Install this ISR in the interrupt vector table. This can be achieved by including an explicit interrupt vector table in your project (as done here). Alternatively, you can include an interrupt vector number in the definition of your ISR. This causes the compiler to take care of the installation of the interrupt vector. Example:

```
__interrupt 0x16 void <ISR_name> (void)
```

(3) Allow all interrupts to happen by clearing the *CCR Interrupt disable flag*. A simple way to achieve this is by using the inline assembler instruction *cli (clear interrupt disable flag).*

In the next section we will set-up a short timer program. Everything we've learned about interrupts will be used in this program.