5. INTRODUCTION TO C WITH THE 9S12 MICROCONTROLLER

We will not have complete coverage of the C language in this course. Instead, we will focus on the tools needed for the microcontroller we use and some of the differences compared to the C# programming language.

Being C a procedural programming language, a C program consists of functions and variables. Therefore, unlike C#, classes and objects do not exist in C.

When we first create a project in Code Warrior, the default main.c file will contain the following:

```
#include <hidef.h>           /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */

void main(void) {
  /* put your own code here */

  EnableInterrupts;

  for(;;) {
    _FEED_COP(); /* feeds the dog */
  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Figure 5.1 – Basic main program generated.

5.1 Including Libraries

In order to use existing libraries, we use the *#include* clause followed by the library we want to add to our project, which is equivalent to the *using* clause in C#. In this example two libraries have been added automatically by the code warrior project creation. We notice the libraries included are referenced in two different ways, one using *<library.h>,* the other one using *"library.h".* When using *#include <filename.h>,* the preprocessor will look for the file in the directories normally pre-defined by the IDE, in our case inside the folder Code Warrior is installed in, whereas when using *#include "filename.h"*, the preprocessor will look for the file in the directories created in the project itself or the libraries that the programmer has manually added to the project. In other words, the libraries the programmer has created or have been added to the project either manually or automatically.

We have mentioned the word *preprocessor*, which may seem unfamiliar to some students. The preprocessor is part of the compilation process , but it is not part of the compiler itself. This means all the clauses that start with # are considered preprocessor directives, which are performed just before beginning compiling the code (converting the C code into machine code). In this case, the preprocessor directive that include these libraries, are adding the libraries so they can be compiled together with the code

written in the main file. We will discuss this in more detail when we start creating our first library.

One of the challenging aspects of working with a C project is to provide the proper path to these libraries. Sometimes errors occur when the compiler cannot find the libraries, especially the ones the programmer has created. That is normally because the project settings are wrong, or the libraries have not been added to the project properly. You may also see in old textbooks that libraries were included using an ***absolute path*** to avoid these errors, for instance, *#include "C:\Users\myUser\Documents \libraries\SD\SD.h"*. We consider this technique obsolete and not recommended since this path only exists in the computer that has the user "myUser". Nowadays, with the wide use of version control systems such as git and svn, multiple users contribute to the same code, or the same user contributes to the project from different machines. For instance, your home computer and the computer in the lab at school. We will then consider using ***relative path*** the appropriate technique for including libraries and configuring the projects.

If we have access to the main.c file through VsCode, we can hold the CTRL key and click in the derivate file (*#include "derivative.h")* to navigate through it. We will then notice this derivative file contains another #include clause that points to the specific library used for our particular micro: #include <mc9s12xdp512.h>. This process was done automatically by the IDE when we selected our specific micro in the project creation, as described in the following image.

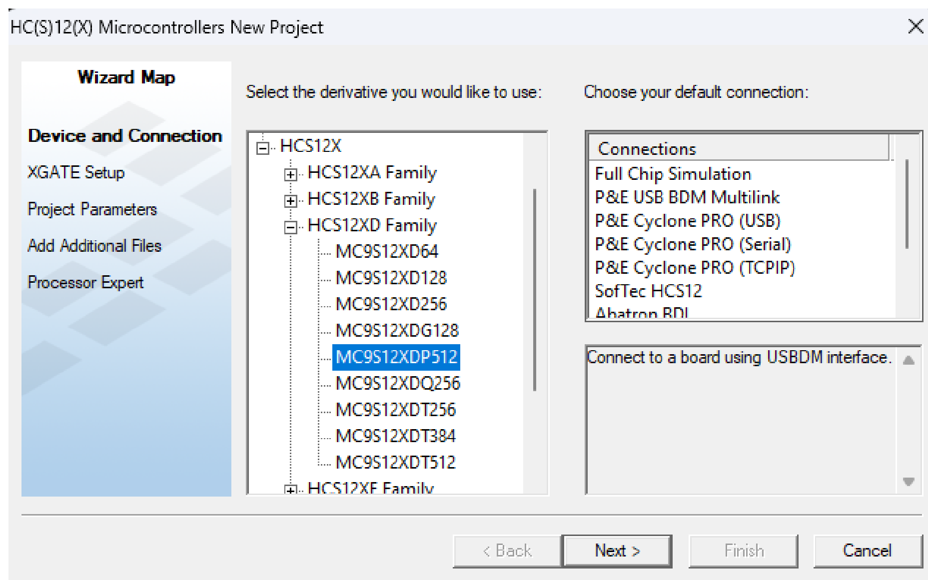

Figure 5.2 - Microcontroller selected.

This makes sense when we start looking into the micro registers and memory map. For now, you can imagine that the C code we write for one specific micro may look a bit different if it is written for a different one. In other words, our main code cannot know

which micro we are using unless we include the proper library for it, the derivative file. This particular library is provided by the microcontroller manufacturer, but some different IDEs may provide variations or extensions of it. The important thing to remember is that the derivative file is fundamental to any micro project we create.

5.2 Main Function

We notice that when the Code Warrior Project was created, the Main function was automatically generated. The main() function is required always, and it will be where the program starts. The main function generated is by default *void main(void),* which means the main function does not return anything and it does not take any parameters, which is expected in this environment, since our micro will be running one task always.

We also notice in main the macro *Enableinterrupts*, which is another preprocessor directive that uses the *#define* clause. In simple words, *Enableinterrupts* is replaced by the code that is placed after the *#define* clause, which exists in some other library provided by the manufacturer (*hidef.h*). We will discuss *Enableinterrupts* later in the course, for now we will keep that portion commented out.

The last part we notice in this simple main code is the infinite loop that is implemented using an unconditional for loop (*for(;;)*). This could also be implemented as a *while(1)* or something like that, but the reason for using the for loop is to avoid a compiling warning, considering we are dealing with a very old C standard (ANSI C / C89).

Lastly, inside the infinite loop we notice the `_FEED_COP()` statement, which is another macro call that feeds the watchdog timer, so it does not restart the program. This is a safety feature provided in case the program halts due to a bug, so it can restart and recover. For simplicity, we will disable this module in the main.c template that we will provide.

The default main file will be replaced by the template file we provide in every new project creation.

```
/*********************************************************************/
// HC12 Program:  YourProg - MiniExplanation
// Processor:     MC9S12XDP512
// Bus Speed:     MHz
// Author:        This B. You
// Details:       A more detailed explanation of the program is entered
here
// Date:          Date Created
// Revision History :
//  each revision will have a date + desc. of changes


/*********************************************************************/
// Library includes
/*********************************************************************/
#include <hidef.h>      /* common defines and macros */
#include "derivative.h" /* derivative-specific definitions */
```

```c
//Other system includes or your includes go here
//#include <stdlib.h>
//#include <stdio.h>


/********************************************************************/
//Defines
/********************************************************************/

/********************************************************************/
// Local Prototypes
/********************************************************************/

/********************************************************************/
// Global Variables
/********************************************************************/

/********************************************************************/
// Constants
/********************************************************************/

/********************************************************************/
// Main Entry
/********************************************************************/
void main(void)
{
  //Any main local variables must be declared here

  // main entry point
  _DISABLE_COP();
  //EnableInterrupts;

/********************************************************************/
  // one-time initializations
/********************************************************************/

/********************************************************************/
  // main program loop
/********************************************************************/
  for (;;)
  {

  }
}
```

```
/*******************************************************************/
// Functions
/*******************************************************************/

/*******************************************************************/
// Interrupt Service Routines
/*******************************************************************/
```

This will help us locate the sections where the code we implement need to be located. We must remember always entering the information in the Author, Details, date, and revision history sections to properly document our project and main file.

5.3 Data Types

The basic data types in C are *char, int, and float.* Considering this 16-bit architecture and the compiler it uses the size for these types are:

- char: 8-bits
- int: 16-bits
- float: 32-bits

All the other types are generated from these basic ones. For instance, *long* represents double the size of an int, which is 32 bits.

A custom data types can be also defined by the programmer or might be already present in some library provided by the manufacturer, which is done using the *typedef* keyword. For instance:

```
typedef  unsigned int    unit;
```

Then the variable declaration: ***uint myNumber*** would be equivalent to ***unsigned int number.***

5.4 Variable Declarations

Most variable declarations are very similar to the way we declare them in C#, except for the arrays. They may also be initialized immediately or later in the code. For instance:

```
unsigned int counter, index = 0;
char data;
float percentage = 0.5;
```

Let's look at the way we declare an array. In C#, this could be an array declaration:

```
int[] myArray = new int[10]; //This declares an array in C# and allocates the size of it.
```

In this case we may or may not specify the size of it as we can re-size it later using the new operator. This implies the use of dynamic memory which is something we want to avoid in the embedded systems environment. In C, we want to allocate the memory for the array immediately, even if we do not initialize it, and it can not be changed after. This means if we are not sure of how large the array needs to be, we need to estimate its size and select something larger to be on the safe side.

The syntax also changes a bit when declaring an array in C:

```
int myArray[10]; //declares array of size 10
int myArray2[] = {2,4,6,8,10}; //declares and initializes an array of size 5
```

5.5 Arithmetic Operators

These are the same as in C#:

- + add.
- - subtract.
- * multiply.
- / divide.
- % modulus (or remainder).
- ++ increment by 1. It can be pre-increment or post-increment.
- -- decrement by 1. It can be pre-decrement or post-decrement.

5.6 Logical Operators

These are also the same as in C#:

- == equal to (two "==" characters, one "=" is assignment!)
- != not equal to
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to
- && and
- || or
- ! not (one's complement)

5.6 Bitwise Operators

These operators are also present in C#, but they have probably not been used in introductory C# courses. The AND, OR operators may look like the logical versions of them, but they are totally different as the bitwise operators are done at the bit level:

- & Bitwise AND
- | Bitwise OR
- ^ Binary Exclusive OR (XOR)
- ~ Binary One's complement
- >> Binary right shift
- << Binary left shift

Understanding how these operators work is crucial in embedded systems, so we will cover them in more detail. It is also important to notice that can be used embedded in an assignment statement. For instance:

```
myVar = myVar << 2;  //this shifts myVar to the left by 2 bits and assigns the result back to myVar
myVar <<= 2;        //this does the same as the previous line
```