Using C with the 9S12XDP512

Table of Contents

Introduction 2
Datatypes 2
  Atomic Types 3
  No bool? 3
  Type Modifiers 3
  The const Modifier 4
  Numerical Notation - defaults and details 4
Working with bits 5
  Unary (1s) complement, or bitwise NOT (~) 5
  Bitwise AND & 6
  Bitwise OR | 6
  Bitwise XOR ^ 7
  Left Shift << (Logical Shift Left for unsigned values) 7
  Right Shift >> 7
The Mighty Derivative File 8
Functions 9
Decision-Making Statements & Looping 10
Interrupts and other Oddities 11
#define/#ifdef/#endif 12
static 12
User-Defined Types (enum & struct) 13
Creating Uncompiled Code Libraries 16
  The Header File (.H) 16
  The Implementation File (.C) 16

Introduction

This document is designed to transition students from C# to C. For students that are familiar with general C or C# programming, this document will also serve to point out items specific to the coding environment of CodeWarrior. Some important notes:

- Having watched the video “Tool Chain Tips” on Moodle is a prerequisite to this document.
- Create a default project to follow along (using the template main.c provided on Moodle).
- As with C#, you will create a complete project per assignment. You will submit this project and supporting libraries to Moodle as an archive when submitting your work. You will be given specific instructions on how to do this at a time when it is more relevant.
- Your main program must never exit, so keep that in mind as you write your code. An endless ‘for’ loop has been provided in the template for this purpose.
- You should get used to placing code elements in the appropriate sections of the template file to keep your code organized and tidy.
- All main.c files and library headers require documentation. More on this as we get into writing code for a specific task.
- C code is much more restrictive than C#, and in this environment, you’ll need to get used to the fact that there is very little library code to leverage – you are providing nearly all the code that executes on the device. The separate compiling and linking stages, along with a single-pass compiler make C an interesting experience.

While C is quite different from C#, there are many concepts that are transferrable.

Datatypes

This section lists the fundamental types of data used in C, as well as the modifiers that affect their declaration and the way they are stored and accessed in memory.

Variables have three basic characteristics:

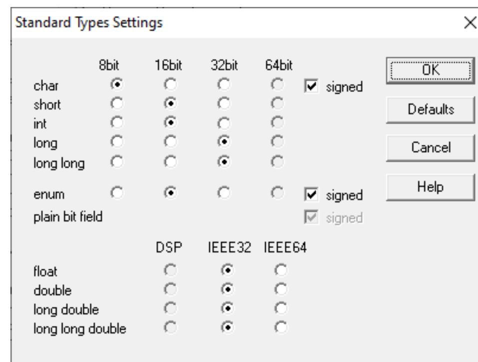
- location - the storage address for the variable
- size - the number of bytes of storage for the variable
- value - the actual state of the bits of the variable

The underlying type of the variable determines how the memory at the location is interpreted as the value. This involves manipulating memory at its location over its size. Memory is just a vast array of bytes; how that memory is interpreted is very important!

Atomic Types

There are several atomic types in C (actual allocation size can vary on different platforms, but the actual size can be obtained using the `sizeof()` operator):

- `char` - integral, often used to store ASCII codes, 8-bits (1 byte)
- `int` - integral, 16-bits (2 bytes)
- `long` - integral, 32-bits (4 bytes) [as a modifier]
- `float` - used to store single-precision floating point values (4 bytes)
- `void` - used to indicate things that have (or return) no value (no size, `sizeof` not defined)



> Unlike in C#, numeric values in C implicitly convert to `true` or `false` when used in a relational expression. All non-zero values evaluate to `true`, and only zero is `false`.
>
> ```
> if (-41)
>         printf("This is true!");
> ```

No bool?

After using a modern language like C#, it is also frustrating to discover the C does not contain the `bool` type. This will make expressions and return arguments require a little more thought, but when coupled with the rule above, we will settle into good patterns.

Type Modifiers

Most atomic types can be modified at the time of declaration with the type modifiers listed below:

- `signed` – Interpret type as being signed. Default for integer types like `int` and `char`.
- `unsigned` – Interpret type as positive only (no 2s complement interpretation).
- `short` – Halve the size of the variable type, where possible (but not here).
- `long` – Double the size of the variable type, where possible. Only used with int.

If no type specifier accompanies a modifier, `int` is assumed. This is why `long`, which is really `long int`, is often mistaken for an atomic type:

```
long long int AsBigAsItGets = 1;
```

The `const` Modifier

Another modifier that may be applied to a type at declaration is `const`. If an instance is declared `const`, the compiler will prevent any attempt to modify the contents of that variable. Values that should not change should be declared `const`. Because values declared as `const` cannot be changed, they must be initialized at the time they are declared:

```
int const iWidth = 800; // Initial CDrawer width
```

With the exception of identity values, meaningful literal numeric values should be declared as `const` values.

For reasons that will become clearer as we discuss pointers, you should always place the `const` modifier after the type (which is the reverse of the usual practice with modifiers). In this way, the const applies to 'the thing to the left'.

```
const int numA = 6;  // seems reasonable, perhaps even intuitive
int const numB = 6;  // actually preferred, because...
int const * const p = &numA; // this is where we are going... eventually...
```

Numerical Notation - defaults and details

- Integral values that begin with a `0` (zero) are interpreted as Octal (`007`)
- Values that begin with a leading `0x` are interpreted as Hexadecimal (`0xFE8D`)
- Undecorated values are treated as decimal (base 10)
- Values that begin with 0b are interpreted as binary `(0b11001110)`
- Values with an appropriately positioned '`E`' are in mantissa, exponent format (`12.3E+11`)
- Integral values default to `int`
- Postfix of a '`u`' specifies an unsigned literal integer. (`123u`)
- Floating point values default to `double, but this is same as float on our micro`
- Postfix of an '`f`' specifies a float literal (`12.234f`)
- A single character enclosed in single quotes (`'a'`) corresponds to an ASCII code value (still a number).
- A series of characters enclosed in double quotes (`"value"`) corresponds to a string literal, which is stored as a zero-terminated array of `char`. Also called "null-terminated string".
- Prefix an '`L`' to the front of a double quoted literal string to have the string interpreted as a Unicode string literal (`L"One Wide String"`), but this is not supported on our platform.

Other types, like `byte`, may be provided by included libraries (including the derivative), but be careful when relying on non-intrinsic types.

Variables may be declared in a new scope or globally in the variables section. Unlike in C#, variables in C must be declared ***before any code*** in the local scope. This includes constructs like `for` loops as well.

Working with bits

All of the addressable memory on a computing platform ultimately comes down to bits. An individual bit may have one of only two states: 1 or 0. In terms of logic, this equates to `true` or `false`. Electrically, it depends on the platform, but will normally be 0V for a logic 0 and > 0.7V for a logic 1 (typically 3.3V or 5V) when expressed in hardware. There are many terms used to describe the state of a bit, depending on context:

```
Logic 1: set, 1, high, true, 5V, etc.

Logic 0: clear, 0, low, false, 0V, etc.
```

You may hear these terms used interchangeably (incorrectly at times), particularly across contexts.

The smallest datatype that you will use to manipulate bits is the `unsigned char` (sometimes as `typedef byte`). An `unsigned char` consists of eight bits on this platform (our micro) but may vary in size on other platforms.

Each bit in the `unsigned char` interpretation carries a 2^n value, moving right to left:

Bit Name	B7	B6	B5	B4	B3	B2	B1	B0
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Value10	128_{10}	64_{10}	32_{10}	16_{10}	8_{10}	4_{10}	2_{10}	1_{10}

Signed integral types are interpreted as 2s complement and will be covered later. For our purposes, when performing bit manipulations, unsigned types will generally, if not always, be used.

Manipulating bits is essentially all that is happening when code is running. Bit manipulations are especially important when working with our micro, as the behaviors of many of the modules are managed through individual bits.

Bitwise operators operate exclusively on integral types and may perform automatic promotion to larger types to hold the result of an operation. Understanding the sizes of operands in bits is critical to making these expressions produce useful values.

You will need to learn techniques (and the associated operators) for manipulating the states of bits in integral types.

Unary (1s) complement, or bitwise NOT (~)

This operator returns the 1s complement of the operand. The operand is not changed, so the result will need to go somewhere. The result has every bit flipped, inverted, or toggled, as you may hear. A `0` becomes a `1`, and a `1` becomes a `0`:

```
unsigned short uiX = 0xA070;  // 1010 0000 0111 0000
uiX = ~uiX;                   // 0101 1111 1000 1111 //uiX is unchanged unless back-assigned
```

Bitwise AND &

The result of the operation is a bit-by-bit AND. When mixing different length arguments, the smaller is promoted and zero padded to match the larger operand (unsigned). Neither operand is modified.

```
unsigned short uiX = 0xA070;     // 1010 0000 0111 0000
unsigned short uiY = 0x559E;     // 0101 0101 1001 1110
//                               & --------------------
unsigned short uiZ = uiX & uiY;  // 0000 0000 0001 0000  //= 0x0010
```

The bitwise AND operation is often used to determine the set state of single or multiple bits in an operand by setting an operand to a specific bit pattern known as a ***mask***.

Set bits in the result indicate a 'on' bit that corresponds to a bit in the mask. For single bit masks, the return value may be evaluated directly as `true`/`false`. This is also true for multi-bit masks, but `0` or more bits may be on in the result.

```
unsigned short uiX =  0xA070;     // 1010 0000 0111 0000
unsigned short mask = 0x0020;     // 0000 0000 0010 0000  // is bit 5 ( 6th bit ) on in uiX?
//                                & --------------------
unsigned short uiZ = uiX & mask;  // 0000 0000 0010 0000  // True ! bit is on !
```

AND is also used to clear bits (set them to `0`). Once again, a mask is used as a pattern for bits to clear, and may be inverted before the AND operation. This looks complex, but is an easy way to self-document the bits you are clearing:

```
unsigned short uiX =  0xA070;     // 1010 0000 0111 0000
unsigned short mask = 0xF00F;     // 1111 0000 0000 1111  // force off upper/lower nibbles
//                                1s 0000 1111 1111 0000  // use 1s complement in and expression
//                                | --------------------
unsigned short uiZ = uiX & ~mask; // 0000 0000 0111 0000  // mask bits are set ON, other bits as they were
```

Bitwise OR |

The result of the operation is a bit-by-bit OR. Promotion and padding works the same as AND. Neither operand is modified.

```
unsigned short uiX = 0xA070;      // 1010 0000 0111 0000
unsigned short uiY = 0x5FF5;      // 0101 1111 1111 0101
//                                | --------------------
unsigned short uiZ = uiX | uiY;   // 1111 1111 1111 0101  // 0xFFF5
```

Bitwise OR is often used to set single or multiple bits in an operand. Once again, a mask is used as one of the operands, and is the pattern of bits that should be set.

```
unsigned short uiX =  0xA070;     // 1010 0000 0111 0000
unsigned short mask = 0x0A00;     // 0000 1010 0000 0000  // make sure these bits are ON (set)
//                                | --------------------
unsigned short uiZ = uiX | mask;  // 1010 1010 0111 0000  // mask bits are set ON, other bits as they were
```

Bitwise XOR ^

The result of the operation is bit-by-bit XOR (exclusive OR). Promotion and padding follow the same rules as above. Neither operand is modified. This is a useful operation when toggling bits in a mask pattern is desired.

```
unsigned short uiX = 0xA070;      // 1010 0000 0111 1010
unsigned short mask = 0x0FF0;     // 0000 1111 1111 0000 toggle middle nibbles
//                                ^ --------------------
unsigned short uiZ = uiX ^ mask;  // 1010 1111 1000 1010 0xAF8A
```

Left Shift << (Logical Shift Left for unsigned values)

This operator will shift the bits in the left-hand argument the number of positions indicated in the right-hand argument. Zeroes are inserted at the LSB position. Bits shifted out the MSB position are lost.

Shifting left results in a value that is twice (2x) the left-hand operand value for each bit shifted.

```
unsigned short uiX = 0xA071;      // 1010 0000 0111 0001
unsigned short uiY = uiX << 5;    // 0000 1110 0010 0000  // 5 zeros << in, bits lost
```

Right Shift >>

This operator functions much the same as left-shift, but bits are shifted right. The behavior of this operator is dependent on the sign of the presented type. For unsigned types it is zero in at MSB and LSB is lost. For signed types, the sign bit is preserved. Most operations in CMPE1250 would use unsigned types for this operator.

```
unsigned short uiX = 0xA070;      // 1010 0000 0111 0000
unsigned short uiY = uiX >> 5;    // 0000 0101 0000 0111  // 5 zeros >> in, bits lost

short uiX = 0xA070;               // 1010 0000 0111 0000 // signed value, negative MSB ON
short uiY = uiX >> 5;             // 1111 1101 0000 0111 // 5 ONES >> in, bits lost
```

Note the signed version, depending on the sign bit (MSB), will shift in ONES to maintain sign, and leave the number in the negative domain (-128_{10} `0b10000000` would become -64_{10} `0b11000000`)!

Finally, compound versions of all binary operators are supported: `&=, |=, ^=, <<=, and >>=`

```
unsigned char a = 0xFF;
a &= 0x10;  // same as a = a & 0x10;
a |= 0x10;  // same as a = a | 0x10;
a ^= 0x10;  // same as a = a ^ 0x10;

a <<= 1;    // same as a = a << 1;
a >>= 1;    // same as a = a >> 1;
```

Use these compound equivalents as you would the arithmetic versions for clarity and brevity.

You must now go and become an *expert* in:

- Knowing the size and types of your operands.
- Converting binary to HEX and back.
- Writing expressions that represent operand literals in the most informative base.
- Writing expressions to selectively turn on, turn off, and toggle specified bits in an operand, without affecting the other bits.
- Writing expressions to determine if specific patterns of bits are found in an operand.
- Selecting suitable types and modifiers for variables in your code.
- Writing expressions to control the flow of your code based on any set of variable states.

The Mighty Derivative File

The project will include derivative support files that contain definitions for port pins and other useful addresses. You may use these definitions in you code to manipulate ports and operate 9S12 modules:

```
// initialize indicator LEDs and switches for use
PT1AD1 &= 0x1F;      // turn off all LEDs so they start in a known state
DDR1AD1 = 0xE0;      // make all LEDs outputs, switches inputs
ATD1DIEN1 |= 0x1F;   // digital inputs on for switches (only applies to this port)

// turn RED indicator LED on
PT1AD1 |= 0x80;
```

Your instructor may discuss some of the details of this file with you now, and certainly will when we start operating modules on the micro. The document “Big Pink” has strong ties to the derivative file, so we can leverage this benefit when writing code.

Functions

Anything you do that could be used repeatedly either in this project or others should be placed in a function.

C supports functions just like C#, with a few minor things of note:

- Unlike C#, methods that take no arguments must explicitly be marked `void`.
- Functions in C must be *prototyped*. This means the compiler must see a function prototype or definition prior to its use. There are two ways to accomplish this – put the function definition above main (you will generally not be permitted to do this), or prototype the function with a function declaration. You will do the latter, and all it requires is that you place a copy of the function signature above main (in the correct comment section) with a semicolon after it. Later, you will see how to make libraries, and this mechanism will change slightly.

```
// a function prototype
void foo (void);

void main(void)
{
  // call to function
  foo ();
}

// function definition
void foo (void)
{

}
```

Decision-Making Statements & Looping

Much of what you have learned in C# with respect to decision-making statements, looping, and other constructs is directly applicable to C. The only caveat here is you can't declare variables after code, and the implicit conversion of an expression to `true`/`false` can impact how you write these constructs.

As a discussion with your instructor, consider the behavior of the following blocks of code:
(what do they mean / how would you comment the code?)

```
{
  int i;
  for (i = 0; i < 10; ++i)
  {
  }
}
```

```
{
  int i = 10;
  while (--i)
  {
  }
}
```

```
{
  if (!i || i%2)
  {
  }
}
```

```
{
  int i = 10;
  while ( i >>= 1)
  {
  }
}
```

Note: Variables can't be declared after code, but you can always open a new scope wherever you like!

Interrupts and other Oddities

The C implemented in CodeWarrior is pretty much hard-core ANSI compliant C, but it does deviate a little in some areas to make things a little easier to work with. For example, interrupts can be handled quite easily:

```
  // setup interrupts on OC0
  TC0 = TCNT + 625; // 0.01 sec
  TSCR1 |= 0x80;    // enable main timer
  TSCR2 = 0x07;     // bus / 128 for prescale (8MHz / 128 = 16us per tick)
  TIOS |= 0x01;     // enable IOS0 as output compare
  TIE |= 0x01;      // enable interrupt for IOS0

  for (;;)
  {
    asm wai;        // wait for an interrupt
  }
}

// ISRs //////////////////////////////////////////////////////////////////////////////////////
interrupt VectorNumber_Vtimch0 void INT_FFEE_ECT0 (void)
{
  TFLG1 = 0x01;     // ack interrupt

  asm sei;          // prevent additional interrupts

  TC0 += 625;       // next interrupt time

  // should make LED flicker like a candle
  if (rand() > 0x2000)
    Red_LED_On();
  else
    Red_LED_Off();

  asm cli;
}
```

To build an interrupt service routine, simply mark the handling method with interrupt. Next place the interrupt number. For this you may simply count the interrupt priority from the top and use the offset, or select the interrupt by definition from the derivative file:

```
#define VectorNumber_Vtimch1          9U
#define VectorNumber_Vtimch0          8U
#define VectorNumber_Vrti             7U
```

The rest of the handler will be a `void`/`void` signature with whatever name you want to call it. This mechanism will work for all 9S12 interrupts.

All standard interrupt handling methodologies still apply – this simply provides the handling function.

You will occasionally see other items in Visual Studio Code that report as errors. Some things are specific to the CodeWarrior environment, and will appear as errors. Don't panic, unless CodeWarrior reports it as a warning or error.

All code should be free of warnings and errors when handed in.

#define/#ifdef/#endif

Commenting in/out code in C is tiresome, as it needs to be done as individual lines, or as blocks (which don't properly nest).

If you have code that you wish to 'turn on' and 'turn off' at compile time, it is recommended that you achieve this with preprocessor directives.

A symbol that is created with a `#define` may be presence tested with an `#ifdef`. If true, the block is included until an `#endif` is encountered. You may then comment in/out only the `#define` to control the presence of entire sections of code:

```
#define use_this_block
#ifdef use_this_block

// this code is now switched on and off at compile time
//  by the presence or absence of the #define

#endif
```

```
//#define use_this_block
#ifdef use_this_block

// this code is now switched on and off at compile time
//  by the presence or absence of the #define

#endif
```

Note: #define makes definitions visible only to the compilation unit.

static

In C, you may use the static modifier on a variable. This has nothing even remotely to do with what static means in C#. In C, the storage of the variable is moved from the stack to global space. What does this mean? A static modified variable will maintain its value between calls to the function that contains it. This is a seriously powerful tool for some of the code you will write, and it will be discussed in the future, closer to when it matters.

User-Defined Types (enum & struct)

The C language supports enumerations and structures just as C# does, with subtle differences.

In C, an enumeration is a new, named type that defines a set of named constant values known as enumerators. Instances of the enumeration may only contain one of the enumerators, thus restricting the set of assignable values.

```
// unnamed + instance (single use)
enum { orange, grape, cherry } PopType;

//// named + instance (multi use)
enum Temps { Hot, Warm, Cold } sensorA;
enum Temps myTemp = Warm; // note: requires enum keyword to declare

//// not named + no instance (defines enumerators)
enum { Rock, Cotton };
```

Because the constant enumerators are regular identifiers, the code that uses enumerations is inherently easier to read:

```
enum eMachineStates { Running, Stopped, Broken };

void foo(enum eMachineStates state) // note: requires enum keyword to declare
{
	if (state == Broken)
		CallRepair();
}
```

Unlike C#, it is not necessary to 'dot' an enumerator from the type – the enumerators are constants declared in the scope of their creation.

You may suppress the need to repeat the `enum` keyword in declarations if you `typedef` the `enum` to a new name (shown below). Note: some purists are vehemently against `typedef`ing `enum`s and `struct`s.

Enumerators are automatically assigned increasing values, starting at `0`. You may override the values for selected enumerators during declaration:

```
typedef enum { Running, Stopped = 50, Broken } eMachineStates;

void foo(eMachineStates state)
{
	printf("\n%d", Running); // 0
	printf("\n%d", Stopped); // 50
	printf("\n%d", Broken);  // 51

	if (state == Broken)
		CallRepair();
}
```

You may also alter the backing type of the `enum`, but this is beyond the scope of CMPE1700.

C also supports aggregate user-defined types with `struct`.

A `struct` definition creates a new type that contains a sequence of members allocated in a block. Each member may be any type known to the compiler at the time of declaration, including pointers to the `struct` being defined. The purpose of the `struct` is to group like data into a single entity (a *record*).

The reported size of a `struct` will always be at least the sum of the sizes of its members, and may be more to accommodate alignment considerations.

Unlike enumerations, structures may be forward declared (could appear in a header):

```
struct SomeStruct;
```

The `struct` definition contains types and names for the members of the structure, in order of their memory footprint:

```
struct SomeStruct
{
	short i;
	float f;
	struct SomeStruct * p;	// note struct keyword required!
};
```

2 bytes	4 bytes	4 bytes	2 bytes
i	f	p	Padding

You might expect that the size of this type is 10 bytes, but with `int` alignment, it will be reported at size 12, and aligned at a 4 byte boundary in memory. Complier options may alter this.

Instances of a `struct` are declared with the `struct` keyword and a *tag name* for a defined `struct`:

```
struct SomeStruct myStruct;
```

Like with `enum`, the extra `struct` keyword may be omitted if the `struct` is `typedef`ed:

```
typedef struct
{
	short i;
	float f;
	struct SomeStruct * p;	// note struct keyword still required here!
} SomeStruct;

SomeStruct myStruct; // contents unknown!
```

A structure instance may be initialized with chicken lips:

```
SomeStruct myStruct = { 5, 2.6f, NULL }; // struct with initialization
```

Members of a `struct` are accessed with the direct member operator '.'

```
// members are accessed with the direct member operator '.'
printf("%d", myStruct.i);
```

You are used to this from C#, where everything you operate is a `class` or `struct` and always has members.

Structures may be passed to functions like any other type. The address of a `struct` yields a pointer to `struct`:

```
void foo(SomeStruct * pStruct)
{
	(*pStruct).i += 5; // must dereference pointer to get direct member operator
}

void main(void)
{
	SomeStruct myStruct = { 5, 2.6f, NULL }; // struct with initialization

	foo(&myStruct);
}
```

C offers a special operator to access members of a `struct` via a pointer: the indirect member operator: '`->`'

```
void foo(SomeStruct * pStruct)
{
	pStruct->i += 5; // indirect member operator (does the dereference for you)
}
```

If a struct contains an array, treat the member like what it is after the indirect member operator – nothing strange here…

```
void foo(SomeStruct * pStruct)
{
	printf ("%d", pStruct->stuff[5]);
}
```

Initialization with arrays will require an enclosing set of chicken lips:

```
typedef struct
{
	double d;
	int stuff[10];
	char q;
} SomeStruct;

SomeStruct temp = { 3.14, {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 } , 'Q' };
```

Creating Uncompiled Code Libraries

To gain the benefits of modularity and reusability, it will make sense to create compilation units for related code. For example, the indicator LED and switch code should be placed in a separate compilation unit so:

- You don't need to have it in your main program
- You can use it in subsequent programs
- You will only have one copy to maintain across all usage instances

Creating a compilation unit requires that you create two extra text files. You will create a header file that contain function declarations (prototypes), and a code file (implementation) that contains the function definitions. There are templates for these files on Moodle.

The Header File (.H)

The contents of the header file should not contain code implementations, but only prototypes and declarations necessary to operate the library.

The header will also include all documentation necessary to operate the library. In production code, headers are freely distributed, and implementations are pre-compiled. This means the user will not get to see the implementation, so the header is all they have as documentation.

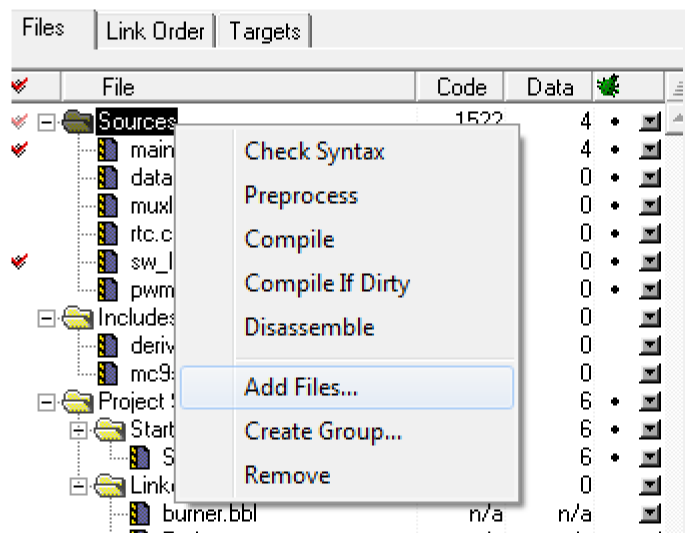
The Implementation File (.C)

The implementation file will include an include directive for the header and include directives for any other referenced compilation units – typically the derivative, but may include other units you have made. It will also include, of course, the matching function implementations for the prototypes in the header.

Implementation files include their header, and any other headers absolutely necessary for the code to compile. No more, no less.

It is recommended that you place these two files in a separate folder at the root of your projects folder, this way all of your projects can refer to the files. You should not keep separate library files in a particular project, copying them forward to new projects. Your instructor will discuss this with you, and there is a video on Moodle discussing the topic.

To use a library, you simply add the implementation file to the project 'Sources' folder:



Browse to your library folder and add the implementation (.c) file to the project.

In your main.c file, include an include directive for the header (.h) file:

main.c

Path: C:\Users\simonw\Dropbox\work\9S12\DPX512\projects\testC - simplified

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

#include "sw_led.h"

// variables
```

The 'include' directive will essentially copy and paste the code from the header into the position of the include directive, meaning you now have the function prototypes that you need in the main file. By adding the implementation file to the project, you've identified another code source that needs to be compiled and linked to your final output. This is all managed by the IDE, so as long as you follow these steps, you should be OK.