Firmware System Design:
Building : Compilation and Linking

1. Chip-Specific Header Files
2. Building : Code Compilation, Linking and Optimisation
3. Disassembly Listings as a Debug Tool

Firmware Development - Reflection

Firmware Systems Design will be used as an example.

Self-Reflection

What do you already know about writing portable code?

How does code in a text file get turned into firmware that runs on a microcontroller?

What do you already know about code optimisation?

Have you ever reconciled code with a Disassembly Listing?

Systems Design - Firmware Development

1. Chip-Specific Header Files
   - Introduction to Chip-Specific Header Files
   - Relating Concept to Hardware
   - Code Portability
2. Building : Code Compilation, Linking and Optimisation
   - Compiling into Seperate Object Files
   - Introduction to Linking
   - Optimisation
   - Finalised Executable
3. Disassembly Listings as a Debug Tool
   - Introduction to Disassembly Listings
   - Using Disassembly as a Debug Tool

Introduction to Chip-Specific Header Files

Chip-specific header files contain information that make it easier to write portable programs

Register addresses for peripheral features, pin names, and communication transciever hardware are presented in a more user-friendly manner. This means less referring back to the datasheet, and more focus on coding

Typically bit masks are saved to enable setting, resetting and toggling of individual bits

Alternatively, complete registers are named to facilitate easy reference

Families of chips follow the same naming conventions to allow cross-compilation of code

Chip-Specific Header Files : Connecting Concepts to Hardware

These header files are typically most helpful at the driver level (Between the firmware and the hardware)

The idea is to abstract the hardware functionality from the specific silicon implementation, and provide a more unified interface across a chip family

Snippets from pic18f452.h

```
extern _at(0x0F80) volatile _PORTA_t PORTA;

#define _PORTA_RA0                      0x01
#define _PORTA_AN0                      0x01
#define _PORTA_RA1                      0x02
#define _PORTA_AN1                      0x02
#define _PORTA_RA2                      0x04
#define _PORTA_AN2                      0x04
#define _PORTA_VREFM                    0x04
#define _PORTA_RA3                      0x08
#define _PORTA_AN3                      0x08
#define _PORTA_VREFP                    0x08

typedef union
  {
  struct
    {
    unsigned TMR3ON                     : 1;
    unsigned TMR3CS                     : 1;
    unsigned NOT_T3SYNC                 : 1;
    unsigned T3CCP1                     : 1;
    unsigned T3CKPS0                    : 1;
    unsigned T3CKPS1                    : 1;
    unsigned T3CCP2                     : 1;
    unsigned RD16                       : 1;
    };
```

Chip-Specific Header Files : Writing Portable Code

Each microcontroller vendor has their own specific silicon implementation and peripheral features

This means that drivers will typically only work for a family of chips. To achieve portability, low level drivers need to be written with a common interface.

Low-level programming languages intrinsically generate less portable code. Higher level programming languages achieve portability through abstraction of the underlying functionality. (Eg. Provide a communications "tunnel" for the application layer)

"Building" Code - Introduction

"Building" a program solution can mean many things depending on the programming language, and its architecture. (Eg. C code is compiled to an executable, whereas Java is assembled into blocks that are ready to be interpreted at run-time)

In the context of microcontrollers, "building" code refers to the process of compiling individual program files, linking them together, and finally generating a single program file that can be executed by a processor.

https://upload.wikimedia.org/wikipedia/commons/1/18/Legobr%C3%BCcke_Wuppertal_2.jpg

Building Code - Introduction

Compiling

https://upload.wikimedia.org/wikipedia/commons/3/32/Lego_Color_Bricks.jpg

Separate code files are compiled into separate object files

Functionality is still distributed and requires further assembly

Linking and Assembly

https://pixabay.com/static/uploads/photo/2013/10/04/18/09/lego-190704_640.jpg

The linker interprets the header file references and instructs how code should be assembled together

Blocks of code can be abbreviated / optimised

Building Code - Compiling Code into Seperate Object Files

Preprocessing aside, compilation is the first step in the build process

Compilation of code is performed file by file. C files are translated into object files

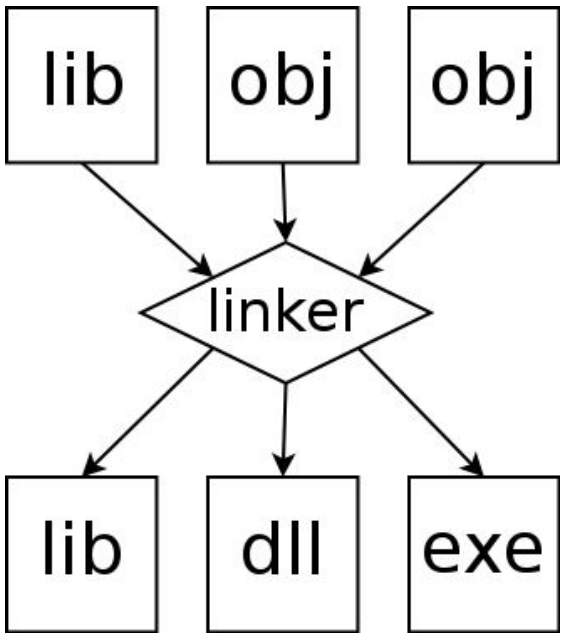
Object files contain machine code, but are not organised to do anything meaningful. The code still needs to be linked, assembled, and deployed

At different stages along this process, code can be optimised for speed / memory use

Building Code - Linking

Linking is the process of drawing connections between code from distributed object files and libraries to realise the final software / firmware solution

The assembler then organises the code to fit the chip memory architecture based on these linked relationships



https://en.wikipedia.org/wiki/Linker_%28computing%29#/media/File:Linker.svg

Building Code - Code Optimisation

Code optimisation can happen at many different stages in the building process

Typical methods are applied at the compilation, linking, and assembly stages

"Optimisation" can mean many things, depending on the scarcity of resources. Perhaps the code needs to run faster (optimise for speed), or the application is pushing the limits of data memory (optimise for memory)

Some optimisations, such as leaving out superfluous library functions may seem obvious, but deeper computer science methods can be used to find alternative processing avenues.

Building Code - Code Optimisation

http://programmers.stackexchange.com/questions/8487/compiler-optimization-examples

Example: How could the following code be optimised?

```
main() {
      long i;
      double x;
      for( i=0; i<100000; i++) {
          x = sin(.3)
      }
      printf("%ld\n", i);
}
```

Building Code - Code Optimisation

http://programmers.stackexchange.com/questions/8487/compiler-optimization-examples

Example: How could the following code be optimised?

```
main() {
      long i;
      double x;
      for( i=0; i<100000; i++) {
         x = sin(.3)
      }
      printf("%ld\n", i);
}
```

This could perhaps be shortened to:

```
main() {
      long i = 100000;
      printf("%ld\n", i);
}
```

Further reading:

http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization

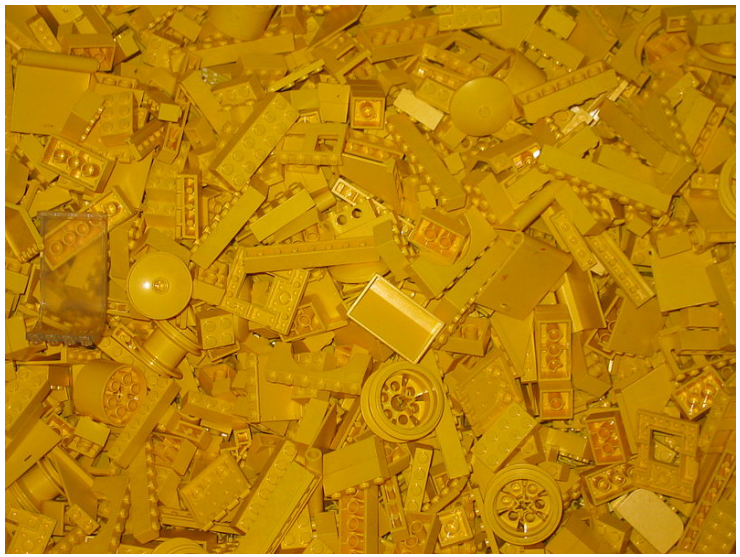
https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Building Code - Code Optimisation

Of course the example on the previous slide had immediately obvious optimisations

Others are significantly more difficult to spot or require significantly more background in the field of application.

All the following bricks have the same yellow colour "functionality" though are optimised for different layouts



https://commons.wikimedia.org/wiki/File:Yellow-lego.JPG

Building Code - Finalised Hex Files

Once code has been built, by compiling individual files, linking the files together, and assembling the program it is stored in a format that the processing machine understands Eg. Binary (or more human readable in Hexadecimal format.)

http://i4.ytimg.com/vi/Hb6IBcjlwZI/hqdefault.jpg

This is the file containing the 1's and 0's that are flashed into the microcontroller program memory that are interpreted by the processor to undertake their function.

Building Code - Debugging Build Errors

When a firmware project does not build completely and correctly (Eg. build failed) this can be due to different types of coding errors

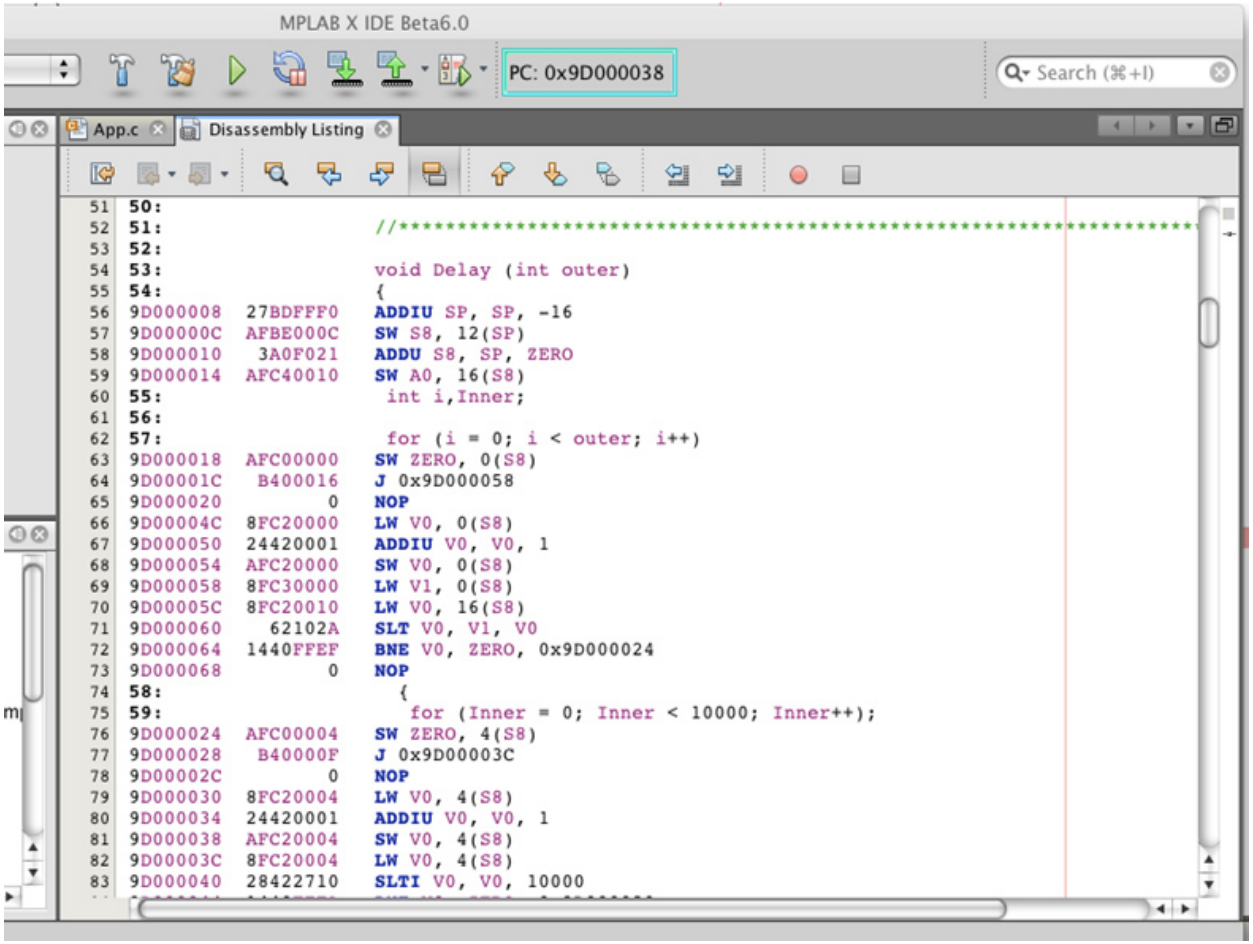
Understanding the build process helps to understand where these errors arise, and how they can be fixed

Generally a compiler throws errors when there is something wrong with syntax within a program file

Conversely, the linker throws errors when different files inadvertently recycle variable names (eg. They are not unique), or when files / libraries are missing or not in the correct directories

Building Code - Introduction to Disassembly Listings

Disassembly Listings are a side-by-side or line-by-line breakdown of higher level code (Eg. C Language) and the assembly operations that the code compiles to



http://www.microchip.com/_images/mplabx/mplabx_large17b.jpg

Building Code - Using Disassembly as a Debug Tool

Disassembly Listings can also be created from machine code to assembler, though they are difficult to follow without knowing what the code does

They can be used as a useful tool to check that C Language code is having the intended consequences at the hardware level. This is particularly true when writing drivers for peripherals

Assembly code has a 1:1 relationship with machine instructions, and is thus the best place to start when troubleshooting low level features

What to Take Away From This Lecture

The Main Points

- Chip-specific header files are a useful way to make driver code portable within a processor family
- Higher level programming allows for more portable code as the low level drivers become abstract
- Building a program solution involves compiling, linking, and assembling the project
- Code can be optimised at the compiler, linker, and assembler levels
- Disassembly Listings are a useful way to reconcile optimisations and eliminate bugs