Firmware System Design: Languages and Code Structure / Architecture

1. Programming Languages
2. Data Flow Diagrams
3. Code Architecture and Libraries
4. Real Time Operating Systems (RTOS)

Steven Dirven, Massey University, 2015

Firmware Development - Reflection

Firmware Systems Design will be used as an example.

Self-Reflection

What programming languages are you familiar with?

What do you already know about how a programming language is executed on a CPU?

How do you currently structure your code for microcontrollers?

What do you know about the costs of abstraction in programming languages?

Systems Design - Firmware Development

1. Programming Languages
    - C language variants and Arduino
    - Assembly language
    - Machine Language
2. Program Flow
    - State Machines
    - Flowcharting
3. Code Architecture and Libraries
    - Header and Code files
    - Defining Constants, Structures
    - Using Existing Libraries
4. Real Time Operating Systems (RTOS)
    - Introduction to RTOS concept (FreeRTOS)
    - Processor Time Slicing

Firmware Development - Programming Lanugages

Example of programming language levels:

Arduino (C/C++ Wrappers)
C Language
Assembler (Machine Instructions)
Machine Code (Hexadecimal / binary 1's and 0's)

Higher level languages abstract the CPU tasks into more programmer-friendly conceptual formats.

The higher level the language, generally the more processor intensive.

Lower level languages are more directly coupled to assembly, and instruction cycles that the machine needs to undertake.

Programming Languages : C Language Variants and Arduino

Arduino sketches are typically written in a mixture of C++ and C with wrapper functions that take care of the low level functionality.

C++ is a super-set of C, so C++ compilers can also compile most C. Code should be platform independent.

C Language is considered a low-level language, which is why it is used in microcontrollers which have limited processing capability.

When the code is built, it is transcompiled into assembler, which is finally translated into machine code.

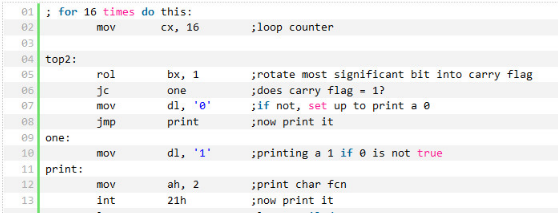
The transcompiled assembler can usually be reviewed in a disassembly listing

Programming Languages : Assembly Languages

Assembler refers to literal instructions that the CPU needs to complete.

Assembler code frequently translates line-by-line to single machine instructions, though some assembly languages and machine architectures can have a 1:many relationship.

It is the lowest level, readily human-readable programming format.



```
; for 16 times do this:
        mov      cx, 16        ;loop counter

top2:
        rol       bx, 1        ;rotate most significant bit into carry flag
        jc        one          ;does carry flag = 1?
        mov       dl, '0'      ;if not, set up to print a 0
        jmp       print        ;now print it
one:
        mov       dl, '1'      ;printing a 1 if 0 is not true
print:
        mov       ah, 2        ;print char fcn
        int       21h          ;now print it
```

Programming Languages : Machine Languages

Machine language is the translation of opcodes that the CPU interprets to determine the fetching, moving, and calculating of processes.

It is stored in binary format (which can be read in hexadecimal) but is still user-unfriendly to read.

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

Programming Languages : Summary

The higher level the programming language, the more human-readable code becomes.

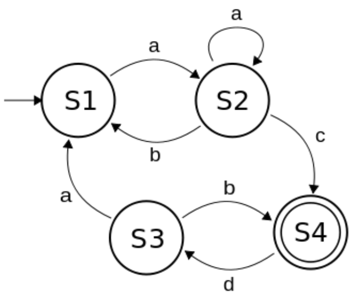
Readability comes at a conceptual level, meaning that a single line, or handful of lines, of high level code, may translate into a few / many / hundreds / thousands of lines of machine code.

Microcontrollers are typically programmed at the C Language level, or below, as program memory and processor capability are limited compared to PCs for example.

Program Flow : State Machines

A state machine is a useful method of breaking code down into modules and dictating a fixed relationship between how different program modules are run after one another.

The architecture can make code easier to follow than a very complex sequential logic flowchart.



https://upload.wikimedia.org/wikipedia/commons/thumb/2/2a/CPT-FSM-abcd.svg/326px-CPT-FSM-abcd.svg.png

Program Flow : Switch / Case State Machine Method

Simple state machines can be organised in a switch case.

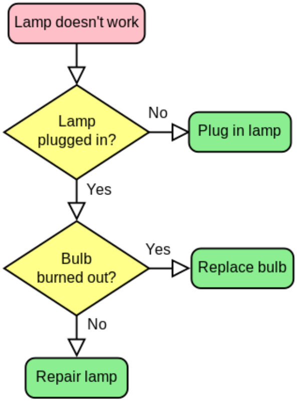
A variable (Eg. "state") holds the current state of the machine, and can be altered from within,or returned from, other functions (Eg. Move to next state).

```
void main(void){
   while(1){
      switch(state)
      {
         case STATE_1:
            state = DoState1(transition);
            break;
         case STATE_2:
            state = DoState2(transition);
            break;
      }
   }
}
int DoState2(int transition){
   // Do State Work
   if(transition == FROM_STATE_2) {
     // New state when doing STATE 2 -> STATE 2
   }
   if(transition == FROM_STATE_1) {
    // New State when moving STATE 1 -> STATE 2
   }
   return new_state;
}
```

Program Flow : Flowcharting

- Visually display the algorithm
- Nodes represent functions or state machine modules.
- Arrows represent the decision made by the function. Eg. State transition

Observe: There is no transition from "Lamp plugged in?" to "Repair lamp". The only way to get there is with a "Yes" path transition to "Bulb burned out?" followed by a "No" path transition.



https://en.wikipedia.org/wiki/Portal:Discrete_mathematics/Selected_article#/media/File:LampFlowchart.svg

Code Architecture

Code Architecture

How is code structured to ensure that it can be ported to different processing systems, and what level language do you need to do this?

What is good practice for organising the layout of code?

How can naming conventions for files and variables be used to make code easier to read/write?

What needs to be considered when using existing libraries?

Code Architecture : Header and Code Files

In C Language, it is important to distribute code modules in header and program files to aid in the development process as well as maintenance and portability.

Header files (headerFile.h) contain function definitions, structure formats, enumerations, and constant definitions.

Program files (codeFile.c) contain variables and the implementations of functions.

It is useful to use a naming convention in all aspects within code, and for filenames. Typical techniques are to: capitaliseAllWordsExceptTheFirst or separate_words_with_underscores.

Code Architecture : Header and Code Files

Recall, C files include header files. A header file is like a table of contents and description of nomenclature.

Code portability (ability to make it run on different microcontrollers) can be achieved with the distribution of code modules in header and C files.

Only peripheral drivers will have to be written for the target platform to work in the same way.

Header files for individual chipsets abstract the concepts of pin connections through #defines, facilitating efficient deployment on new hardware.

Code Architecture : Header File Example

An example of a header file (AD7147.h) for SPI comms with a chip

```
/*
 * AD7147.h
 *
 * Created: 9/06/2015 9:10:22 a.m.
 *  Author: sdirven
 */

#include <avr/io.h>

#ifndef AD7147_H_
#define AD7147_H_

void AD7147_Init(void);

void AD7147_ColSelect(int colNum);

int AD7147_IsReady(void);

void AD7147_ReadRows(char* Data);

void AD7147_ReadID(char* StringID);

#endif /* AD7147_H_ */
```

Notice the consistent naming convention

Code Architecture : Code File Example

Snippet from supporting code file (AD7147.c)

```
/*
 * AD7147.c
 *
 * Created: 9/06/2015 10:35:44 a.m.
 *  Author: sdirven
 */
#include "AD7147.h"
#include "SPI.h"

void AD7147_Init(void) // Might be a good idea to disable calibration here too.
{
        PORTB &=0b11111110; // Select AD7147 chip
        // Setup Stage 0 : 0x E080 3FFE DFFF : Single Ended connection setup with CIN0
        SPI_SendByte(0xE0); // Stage 0
        SPI_SendByte(0x80);
        SPI_SendByte(0x3F); // CIN0
        SPI_SendByte(0xFE);
        SPI_SendByte(0xDF); // Single-Ended
        SPI_SendByte(0xFF);
        PORTB |=0b00000001; // Deselect AD7147 chip
        Delay_MS(20);       // Pause briefly
        PORTB &=0b11111110; // Select AD7147 chip
        // Setup Stage 1 : 0x E088 3FFB DFFF : Single Ended connection setup with CIN1
        SPI_SendByte(0xE0); // Stage 1
        SPI_SendByte(0x88);
        SPI_SendByte(0x3F); // CIN1
        SPI_SendByte(0xFB);
        SPI_SendByte(0xDF); // Single-Ended
        SPI_SendByte(0xFF);
```

Code Architecture : Defining Constants / Structures / Enums

The following types of definitions should be placed in header files.

Define a Constant	`#define CNAME value`
Define a Structure	`struct myStruct{ int one; int two; };`
Define an Enumeration	`typedef enum {RAND, IMM, SCH} Strat; Strat strategy = IMM;`

```c
#define CNAME value
```

```c
struct myStruct{
    int one;
    int two;
};
```

```c
typedef enum {RAND, IMM, SCH} Strat;
Strat strategy = IMM;
```

Code Architecture : Using Existing Libraries

Existing libraries make development faster, especially for low-level drivers

Read and understand the code to ensure that there are no bugs, that it is compatible with the chosen hardware, and does not have any unexpected requirements / dependencies. (Eg. That it does not inadvertently hijack a timer.)

Additionally, generic features may not always take advantage of hardware peripherals (Eg. bit-bashed PWM rather than hardware PWM)

Real Time Operating Systems (RTOS)

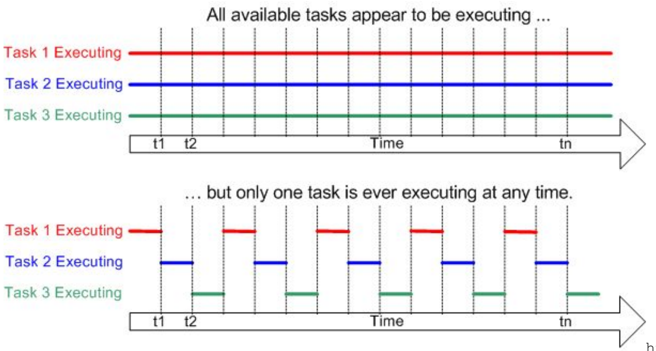
Real Time Operating Systems allow for multiple tasks to appear as if they are running at once.

RTOSs consist of tasks, which are prioritised by a scheduler. Information can pass between these tasks (or threads) by means of queues that can have messages added with differing priority.

Instead of polling, or waiting for something to happen, the CPU can process something else.

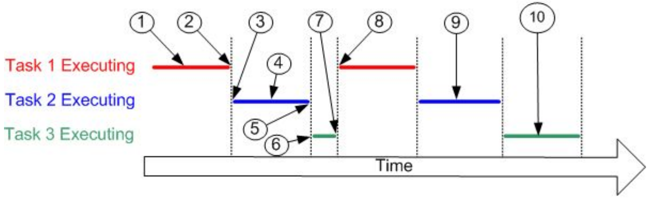
freeRTOS is an example of such a system.

RTOS : Processor Time Slicing

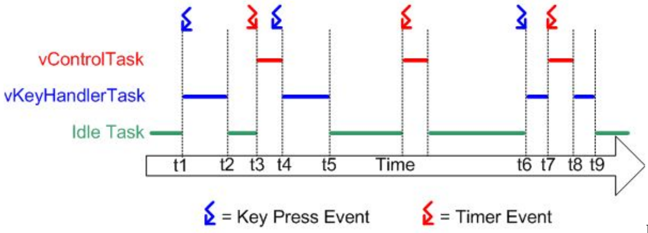


http: //www.freertos.org/implementation/a00004.html

RTOS : Processor Task Prioritisation



http://www.freertos.org/implementation/a00005.html



http://www.freertos.org/implementation/a00008.html

What to Take Away From This Lecture

The Main Points

- Different levels of programming languages have different strengths and weaknesses. Higher level is more conceptual and portable, but perhaps more processor intensive. Low level is more efficient, but more constrained to particular processors and harder to read / follow.
- It is important to determine algorithm, state machine, and file architecture before beginning coding
- Generic Libraries make coding easier, though can sometimes be refined to be made more efficient
- Real Time Operating Systems time-slice to allow multiple processes to run at a time. Have to consider thread-switching processor overhead.