Review of C-Programming:
C-Programming Basics (for Embedded Controller)

Young-Keun Kim

mod .2023.08.19

Firmware programming example

- Example of firmware programming in C

#Define

Pointer Typecasting

Enum

Structures

Bitwise

```
#ifndef __EC_GPIO_H
#define __EC_GPIO_H

#define INPUT 0x00
#define GPIOA  ((GPIO_TypeDef *) GPIOA_BASE)

typedef enum
{
    /******  Cortex-M4 Processor Exceptions Numbers ***                */
    RCC_IRQn = 5,      /*!< RCC global Interrupt                       */
    EXTI0_IRQn = 6,     /*!< EXTI Line0 Interrupt                       */
    EXTI1_IRQn = 7,     /*!< EXTI Line1 Interrupt                       */
}

typedef struct
{
    __IO uint32_t MODER;    /*!< GPIO port mode register,           Address offset: 0x00     */
    __IO uint32_t OTYPER;   /*!< GPIO port output type register,      Address offset: 0x04     */
    __IO uint32_t ODR;      /*!< GPIO port output data register,      Address offset: 0x14     */
} GPIO_TypeDef;


// GPIO Mode : Input(00), Output(01), AlterFunc(10), Analog(11, reset)
void GPIO_mode(GPIO_TypeDef* Port, int pin, int mode) {
    Port->MODER &= ~(3UL << (2 * pin));
    Port->MODER |= mode << (2 * pin);
}
```

Firmware programming

- We will concentrate on :

1) Structure and Enum

2) Pointer and Pointer typecasting

3) Bitwise operation

For assignment, you have to submit all the exercises in each topic.

Part 1: Structure and Enum

1) Study on Structure and Enum

2) Do Exercises.

3) Submit Assignment

Part 2: Pointer and Pointer Type Casting

1) Study on Pointer and Pointer typecasting

2) Do Exercises.

3) Submit Assignment

Part 3: Bitwise Operation

1) Study on Bitwise operation

2) Do Exercises.

3) Submit Assignment

Additional Notes on Firmware Programming

Notes on Firmware Programming

C-language has a lot of "built-in" functions

- Definition included in *header files* #include<header_file.h>
- You can use these functions on MCUs
- But not all can be used directly . e.g. "printf()"

Compiling process

- Compiler translates C code into assembly code
- Assembler (e.g. built into uVision5) translates assembly code into object code
- Object code runs on machine

Special function called *main* ()

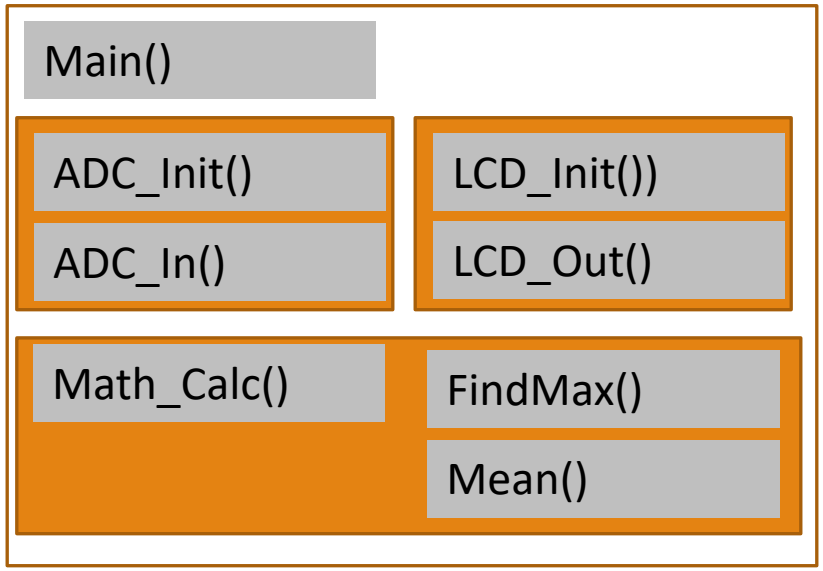
- Reset vector at ROM location 4 => PC
- Initializes global variables
- At reset of MCU, it jumps to main

Notes on Firmware Programming

- **TIP: Write structured Programs**
  - Draw Flowchart and analyze how the process work

  - Divide a SW project into tasks
    - Easier to understand
    - Increase number of modules
    - Decrease the interdependency

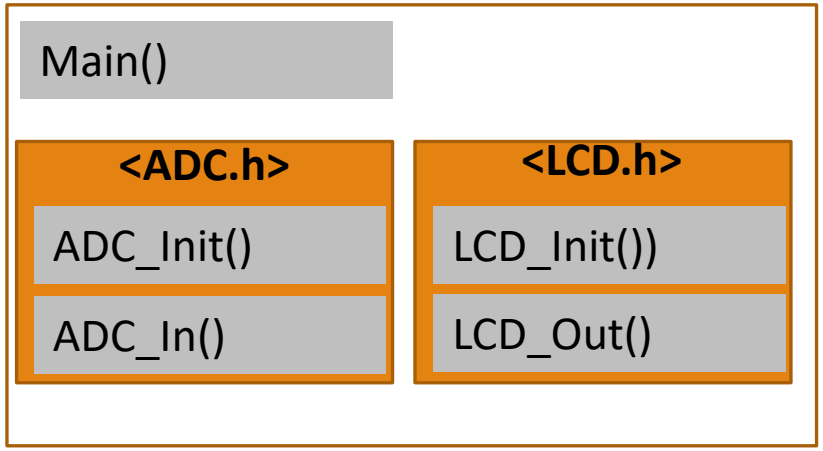
  - Divide a SW task into Modules
    - Easy to Debug/Understand
    - **Reusable and portable**




```
unit32_t MaxVoltage=0;
int main()
{
        // other initialization
        ADC_Init();
        LCD_Init();
        while(1)  {
                ADC_In(…);
                Math_Calc(…);
                // other subtask
                LCD_Out(MaxVoltage);
        }
        return 0;
}
```

TIP: Define Functions in Header file

- Grouping tasks in similar functions
  - Reusable and portable
  - Easy to Debug/Understand
- Header file (*.h)
  - Declaration of Functions
  - Constant, Variables, Class
- Header definition file (*.c)
  - Definition of Functions



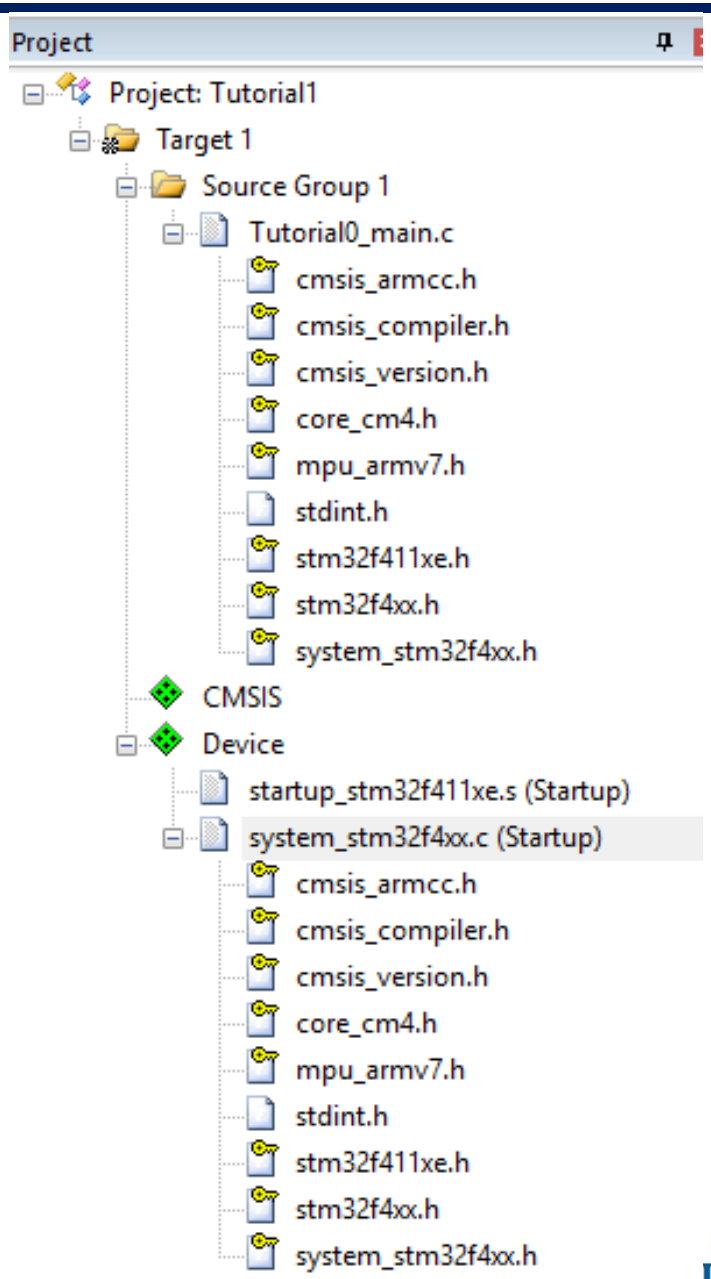

```
#include <ADC.h>
#include <LCD.h>

unit32_t MaxVoltage=0;
int main()
{
        // other initialization
        ADC_Init();
        LCD_Init();
        while(1)  {
                ADC_In(…);
                // other subtask
                LCD_Out(MaxVoltage);
        }
        return 0;
}
```

Notes on Firmware Programming

- **TIP: Define Functions in Header file**

Header files in MCU programming

- A typical project will include
  - Standard lib of 'C' (C99)
  - STM libraries
  - ARM(CMSIS) libraries
  - User defined Memory mapped I/O
  - etc



TIP: Define Functions in Header file

- Create a header file (*.h) and declare function in the header
- Create the definition file for the header (*.c) and define functions

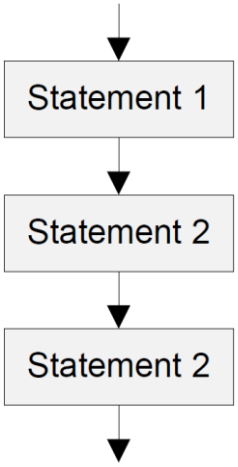
```c
#include <stdlib.h>                    // standard library ( malloc, calloc, free, system, etc )
#include <stdio.h>                     // standard input/output (scanf, puts, printf, etc)
#include "../Library/EC_HAL.h"

void main()
{
	float x[20] = { 0 };
	float y[20] = { 0 };
	float out[20] = { 0 };
	float out_dotProduct = 0;
	int vecLength = 0;


	….
	….


	addVector(x, y, out, vecLength);


	printVec(out, vecLength);
	system("pause");
}
```

User defined header EC_HAL.h

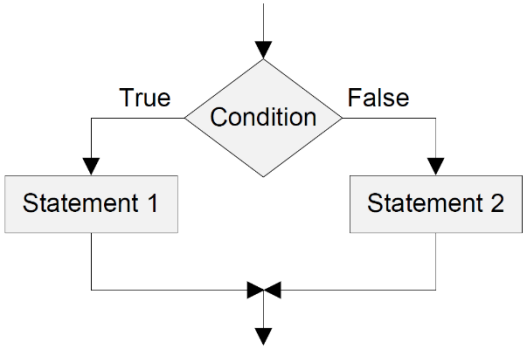
EC_HAL.c

```c
#include "EC_HAL.h"

void addVector(float _src1[], float _src2[], float _dst[], int _vecLength) {
		for (int i = 0; i < _vecLength; i++)
		_dst[i] = _src1[i] + _src2[i];
}
```

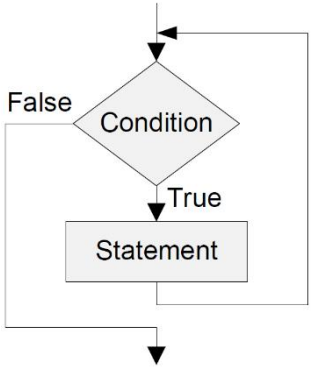
- TIP: Design your algorithm with Flow Chart



Sequence Structure

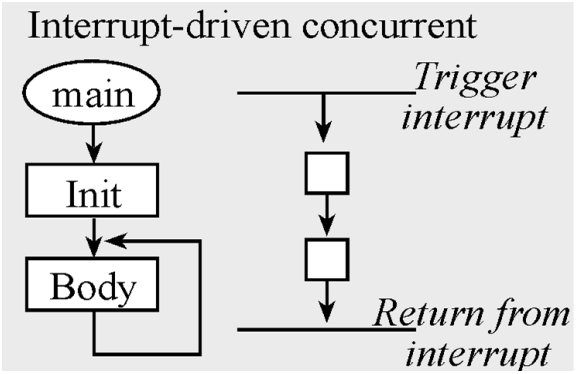


Conditional Structure



Loop Structure

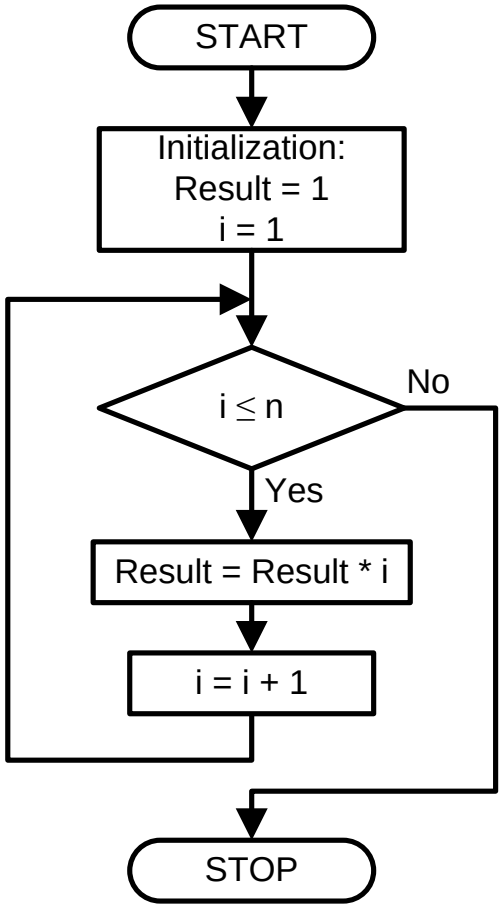
Interrupt Structure



TIP: Design your algorithm with Flow Chart

Example: Factorial Numbers

$$n! = \prod_{i=1}^{n} i = n \times (n-1) \times (n-2) \cdots \times 2 \times 1$$



C Program

```
int main(void) {
  int result, n, i;
  result = 1;
  n = 5;
  for (i = 1; i <= n; i++)
    result = result * i;



  while(1);
}
```

Notes on Firmware Programming

- **TIP: use specific variable type**
  - **int32_t** or **unsigned int32_t** instead of **int**
  - **uint8_t** for flag

Integer: C99 defined in <stdint.h>

C99	Data Type	Data Size (bits)	Data Range
int8_t uint8_t	Char unsigned char	8	-128 - 127(signed) 0 - 255(unsigned)
int32_t **uint32_t**	int/long int unsigned int	32	-2,147,483,648 - 2,147,483,647(signed) 0 ~ 4,294,967,296(unsigned)
int16_t uint16_t	short int unsigned short int	16	-32,768 - 32,767(signed) 0 ~ 65,536(unsigned)
int64_t uint64_t	long long int unsigned long long int	64	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807(signed) 0 ~ 18,446,744,073,709,551,616

Notes on Firmware Programming

- **TIP: General Guidelines for Naming conventions**
  - Make variable names to be easy to understand and consistent
  - Some helpful guidelines
    - Name should have meaning
    - Use capital letters to delimit words: maxTemp
    - Avoid ambiguities. e.g. temp
    - Give hints about the type
      e.g. Pointer variable: dataPt, timePt, ..
      e.g. Global var: VoltageIn
      e.g. Local var: voltageIn
    - Use Prefix to identify public objects
      e.g. 'underline char' for Public objects: LCD_Init
    - All capital letter for global and constant variables such as I/O port address

Object type	Names
Constants	GPIOA_PIN0
Local Variables	maxTemp, minVoltage
Private Global variables	MaxTemp, MinVoltage
Public global variable	ADC_MaxTemp, DAC_MinVoltage
Private function	ClearTime(), InChar()
Public function	Timer_ClearTime(), LED_Init()

Notes on Firmware Programming

- **TIP: Comments can help in understanding the SW**
  - Write concise comments
  - It is recommended to write a separate document file to explain the functions or your library

  - Beginning of every file
    - File name, purpose , hardware, programmer, date, copyright etc.

  - Beginning of every function
    - Function purpose, Input/output parameters, special conditions etc

```
/**
******************************************************************************
* @authorSSSLAB
* @Mod2021-8-12 by YKKIM
* @briefEmbedded Controller:  EC_HAL_for_student_exercise
*
******************************************************************************
*/


#include "ecGPIO.h"

/* Timer Configuration */
void TIM_init(TIM_TypeDef* timerx, uint32_t msec) {
    // 1. Enable Timer CLOCK
    if (timerx == TIM1) RCC->APB2ENR |= RCC_APB2ENR_TIM1EN;
     ...
}
```

Notes on Firmware Programming

- **TIP: Preprocessor and Macro**
  - What is preprocessor?
    - not a part of the compiler, a separate step in the compilation process
    - a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
    - **can create a macro**
  - Examples of preprocessor
    - #include
    - #define
    - #ifdef #if #elif #else #endif

```
#define PERIPH_BASE           0x40000000UL            /*!< Peripheral base address in the alias region  */
```

Header file

```
// stm32f4xx.h

#ifndef __STM32F4xx_H
#define __STM32F4xx_H
...
#endif /* __STM32F4xx_H */
```

Macro

```
// macro
#define SET_BIT(REG, BIT)      ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)    ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)     ((REG) & (BIT))
```

Notes on Firmware Programming

- **TIP: Volatile keyword**
  - **Volatile:** Do not do compiler optimization
    - that can change in ways that **cannot be determined by the compiler**
    - allows the variables to be changed from outside the program

```
volatile uint8_t variable;
// uint8_t volatile variable;

volatile uint8_t * variable;

uint8_t volatile * variable;
```

- Examples of declaring as volatile:
  1) Global variables modified by interrupt service routine (ISR)
  2) Global variables within a multi-threaded application / RTOS
  3) Memory-mapped I/O register variables

```
#define PORTB (*(volatile uint8_t*)(0x18 + 0x20))
```

Notes on Firmware Programming

- Example of 'volatile'

```
uint16_t x;

void main(void)
{
    x=10;
    x=30;
}
```

Optimized:
The compiler ignores 'x=10' to reduce codes

Not Optimized:
The compiler creates machine code for both 'x=10' and 'x=30'

Example : Interrupt variables

```
static int power;
void Power_ON_OFF()
{
   power = 0;
   while(power!=1);
}
```

After optimization
-> makes it inf loop

```
void Power_ON_OFF()
{
   power = 0;

   while(true);
}
```

But we may change 'power' by interrupt
→ Declare it as *'volatile static int power; '*

- **TIP: Static keyword**
  - Allocated memory in Data Segment (NOT Stack Segment)
  - Local static: preserving their value even after they are out of their scope
  - Global static: global variables but CANNOT be accessed by other files (as extern)

```
int fun(){
  static int count = 0;
  count++;
  return count;
}

int main(){
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output: 1 2

```
int fun(){
  int count = 0;
  count++;
  return count;
}

int main(){
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output: 1 1

Notes on Firmware Programming

- **TIP: Extern keyword**
  - Extern variables is declared as global variable which **CAN be** accessed in other source files
  - Should be defined only once, but can be declared number of times

extern uint32_t var ← declared but not defined (no memory allocated)
extern uint32_t var=0; ← declared & defined (memory allocated)

```
// system_Stm32f4xx.h
extern uint32_t SystemCoreClock;        /*!< System Clock Frequency (Core Clock) */
```

No error:
Declared only

```
int var;
int main(void)
{
  var = 10;
  return 0;
}
```

Compilation error:
Not defined

```
extern int var;
int main(void)
{
  var = 10;
  return 0;
}
```

No error:
Defined

```
extern int var = 0;
int main(void)
{
var = 10;
return 0;
}
```

No error:
If defined in a file in the project

```
#include "somefile.h"
extern int var;
int main(void)
{
var = 10;
return 0;
}
-------------------------
//”somefile.h"
extern int var=1;
```