Coding styles for EL402 Embedded Systems

Category	Item	Description	Rev
Project structure	1	To simplify use of a version control system, and to deal with unexpected programmer departures and sicknesses, every programmer involved with each project will maintain identical directory structures for the source code associated with the project.	
	1.1	The general “root” directory for a project takes the form: workspace/project-name/trunk/ project name must not have space or any special characters except underscore '_' or dash '-'	1
	1.2	Required directories: workspace/project-name/trunk/src - .c and .s source files workspace/project-name/trunk/inc - .h header files workspace/project-name/trunk/libs - source of external libraries workspace/project-name/trunk/test - .c and .h for test code	1
	1.3	Each project will have a special module "version.h" that provides information of development: - firmware version and date. - Toolchain version and settings. - List of external libraries and their version - All changes to the current version with the newest changes at the top of the file.	1

```
/**
* Project HMI
* Version 0.1
*
* Toolchain: Keil MDK-ARM 4.73
* Device:  STM32F051R8
* Xtal:  8 MHz
* Code generation:  Use MicroLIB
* Compiler directive: USE_STDPERIPH_DRIVER
* Debugger:  ST-Link Debugger
* Flash algorithm:  STM32F0xx 64k Flash
*
* 10/14/14 (v0.1) original version
*/
# undef VERSION
# define VERSION "HMI 0.1"
```

Category	Item	Description	Rev
Keil-specific project structure	2	In integrated development environments, specify a PROJECT file that is saved with the source code to configure all MAKE-like dependencies.	
	2.1	Required source group: startup - all startup code main - main program layer-name - source code for each layer library-name - source code of external libraries	1
Modules	3	A module is a pair of .c and .h source code that contains functions and related variables. Grouping functions as a module makes it easier to find relevant sections of code, and allows more effective encapsulation.	
	3.1	Keep module sizes under 1000 lines to enhance clarity.	1

	3.2	Prepare module templates named "template.c" and "template.h", stored in the source directory	1

```
/**
* @file      module.c
* @author    Supachai Vorapojpisut
* @version   1.0
* @date      October 14, 2014
* @brief     Template code for user source code.
*/
/* Includes --------------------------------------------*/
/* Private types ---------------------------------------*/
/* Private constants -----------------------------------*/
/* Private macro ---------------------------------------*/
/* Private variables -----------------------------------*/
/* Private function prototypes -------------------------*/
/* Private functions -----------------------------------*/
```

```
#ifndef __MODULE_H
#define __MODULE_H

#ifdef __cplusplus
 extern "C" {
#endif

/* Includes --------------------------------------------*/
/* Exported types --------------------------------------*/
/* Exported constants ----------------------------------*/
/* Exported macro --------------------------------------*/
/* Exported functions ----------------------------------*/

#ifdef __cplusplus
 }
#endif

#endif // __MODULE_H
```

	3.3	The template includes a section defining the general layout of functions.	1

```
/**
* @brief  Purpose of function.
* @param  arg  Purpose of arguments
* @retval Purpose of return value
*/
```

	3.4	Use "main.c" for main program.	1
	3.5	Never use project name as part of module name.	1
	3.6	For hardware-related modules, use "hal_" as prefix.	1
Naming convention	4	Reasons for using a naming convention are to reduce the effort needed to read and understand source code and also to enhance source code appearance.	
	4.1	Use long names to clearly specify the variable’s meaning. And separate words within the variables by underscores. For example, "network_status".	1
	4.2	Variable and function names are defined with the first words being descriptive of broad ideas, and later words narrowing down to specifics. For example, "network_status" "logging_debug()"	1

	4.3	Global variables are defined with first capital letter for each word. For example, "Port_Status". Local variables are defined with all small characters.	1
	4.4	Constants are defined with all capital characters.	1
	4.5	Identifiers should be prefixed by a module identifier. For example, "network_status" will be in "network.c".	
Functions	5	Keep function small and explicitly declare every parameter passed to each function.	
	5.1	Limit function by the length of two-page print.	1
	5.2	Define a prototype for every called functions.	1
Interrupt service routine	6	ISRs are often the hardest parts of firmware to design and debug. Long and slow ISRs are two major factors of system failures.	
	6.1	Keep ISRs short and fast.	1
	6.2	Avoid using loops in ISR.	1
Coding conventions	7	Programmers are highly recommended to follow guidelines to help improve the readability of their source code and make software maintenance easier.	
	7.1	Use integer types defined in stdint.h, i.e. uint8_t, int32_t.	1
	7.2	Put a space after every keyword, but never between function names and the argument list.	1
	7.3	Never use <magic numbers>. Always define constants with appropriate names.	1
	7.4	Use parenthesis to explicitly declare precedence.	1
	7.5	Place opening braces as the last on the line, and the closing brace first. Except the brace pair defining function scope.	1

```
void judge(int ref, int data)
{
  if (ref > data) {
    do_something();
  } else {
    do_another_thing();
  }
}
```

	7.6	Indent C code in increments of two spaces. Avoid using tab character as indent.	1
	7.7	Separate code block by a line feed. Use three line feeds to separate between functions.	1

Reference

A Firmware Development Standard, Jack G. Ganssle

C Coding Standard, Micrium