Hochschule Esslingen
University of Applied Sciences
Fakultät Informationstechnik

[HElikopter

Software Concept

Masterquad 2015

in the degree course ASM-SB
of the Faculty Graduate School
ASM2

Oliver Breuning
Martin Brodbeck
Jürgen Schmidt
Phillip Woditsch

Periode: Summersemester 2015
Professor: Prof. Dr. Jörg Friedrich

Contents

1 Software Layer Concept 1

2 Code documentation 3

3 Coding style guide 5
- 3.1 Naming style . . . 5
  - 3.1.1 Naming structure for defines & macros . . . 5
  - 3.1.2 Naming structure for variables . . . 6
  - 3.1.3 Naming structure for functions . . . 6
  - 3.1.4 Literals of scope . . . 7
  - 3.1.5 Literals of data types . . . 9
- 3.2 Comments . . . 11
  - 3.2.1 Type definitions and non-local variables . . . 11
  - 3.2.2 Interfaces . . . 12
  - 3.2.3 Functions . . . 12

List of Figures

1.1 Software layers with functional units of project MasterQuad 2015 2

List of Tables

1.1 Comparison between software layers and hardware modularity 2

3.1 Literals for the scope of variable naming 7
3.2 Literals for the scope of function naming 8
3.3 Literals for the data types of variable and function naming 9

1 Software Layer Concept

One of the core goals of the layering concept is the maximization of code re-usability. Since major parts of the software is intended to run on multiple platforms, it is mandatory to keep a strict separation between hardware-dependent and hardware-independent software.

Furthermore, since the hardware is highly modularized, the software shall reflect this modularity as close as possible to ease the interchangeability of sensors. In detail, the modularity encompasses the microprocessor platform (in this project: `Raspberry Pi B+`) and several extension boards equipped with sensors. In table 1.1, a detailed comparison between hardware modularity and software layering is shown.

To achieve the goal of a maximal code re-usability, the software shall be structured in four general layers (see fig.1.1). Within each ***Layer***, several ***Functional Units*** are defined in order to divide the software into logically separable modules.

1. **Application Layer (app)**
   This layer contains all high-level software that is necessary for the control of the quadrocopter. Control loops, position hold control, autonomous landing control and supervising functions are here located.

2. **Signal Processing Layer (sig)**
   In order to give a flexible framework for filtering and data fusion, a dedicated layer is introduced. The raw sensor data shall be filter (e.g. lowpass filtering). In a second step, the received data shall be fusioned in order to achieve an robust and reliable orientation representation of the quadrocopter.

3. **Hardware Abstraction Layer (hal)**
   Since all software of the `Application Layer (app)` and `Signal Processing Layer (sig)` shall be system independent, the `Hardware Abstraction Layer` provides all drivers for the used sensors and extension boards. The interface towards the `Signal Processing Layer (sig)` will generalize the data flow, independent of the used sensors.
   The interfaces towards the `Low-Level Driver Layer (LLD)` will be called **Low-Level Driver Interfaces (LLD_IF)**. These interfaces shall abstract the access the low-level drivers that are usually microprocessor-specific to the used hardware platform (here: Raspberry Pi B+). Thus, all

4. **Low-Level Driver Layer (LLD)**
   The `Low-Level Driver Layer` contains all drivers needed for low-level data communication like `UART`, `I2C` or `SPI`. For the scope of this project (MasterQuad 2015), all needed low-lever drivers are already provided by the chosen operating system.

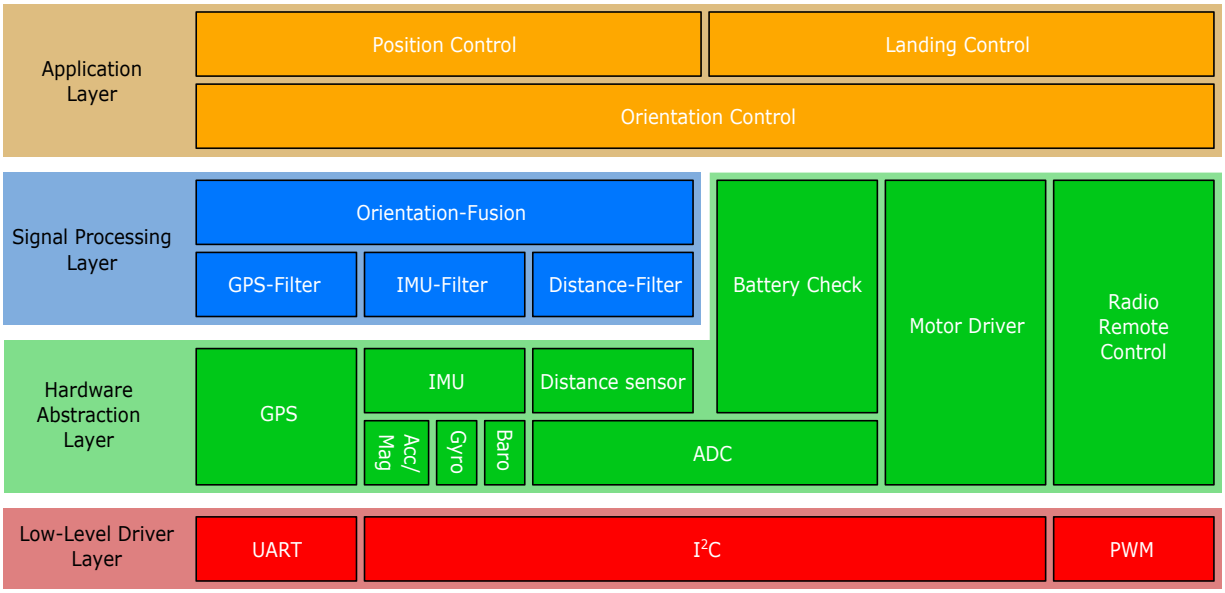

Figure 1.1: Software layers with functional units of project MasterQuad 2015

Software layer	**Equivalent interchangeable part**
Application Layer (APP)	Control loops and supervising/monitoring functions that are usually independent of the used hardware. ***Example:*** *GPS-Position Hold functionality*
Signal Processing (SIG)	**Signal conditioning** for used sensors. Additionally for sensor fusion (partly sensor-specific: parameterization of fusion algorithms are usually dependent on the signal/noise properties of sensors). ***Example:*** *Kalman-Filter (sensor fusion)*
Hardware Abstraction Layer (HAL)	**Drivers** for extension boards equipped with sensors for measurements of e.g. acceleration, gyro and distance-to-ground. ***Example:*** *GPS-Sensor (extension board)*
Low-Level Driver Layer (LLD)	Mircoprocessor platform incl. Operating System (if present) and drivers for bus communication (I^2C, etc). ***Example:*** *Raspberry Pi B+ Board*

Table 1.1: Comparison between software layers and hardware modularity

2 Code documentation

For documentation of source code, **doxygen** shall be used as an automatic code documentation tool. Doxygen is a special tool that scans C-Files for comments and creates a highly readable documentation in HTML or LaTex. In consequence, this also implies special commenting rules for the coding style guide (see 3.2).

Short guide for doxygen compliant comments

For every comment that shall be part of the automatic code documentation via doxygen, the respective comment (single line or block) has to be in the line directly above the relevant code. In order to make a comment available to doxygen, the C-comments have to be extended as shown below.
Example:

```
1 //! this is a single line comment (important: '//!' to use doxygen)
2 unsigned int g_sigGpsfilt_fooStorage_ui32;
```

It is also possible to give a short comment in the same line of the relevant code - behind the expression.
Example:

```
1 int g_halAcc_countNr_i32; //!< doxygen comment in same line as expression
```

Both, single line comments and block comments are supported by doxygen.
Example:

```
1 /*! this is a block comment that will be integrated to the code
2     documentation, created by doxygen. Similar to C-block comments
3     multiple lines are possible. Note the additional exclamation mark
4     in order to get a doxygen compliant block comment!
5 */
6 float g_appLanding_cumquatJuice_f32;
```

Important remark:
Only the **definitions of functions** (`.c` files) shall be commented in a doxygen compliant manner. The comments of function declarations (`.h` files) shall only enhance the readability and do not have to be doxygen compliant. All declarations of **custom data types** or **global variables** (typically in `.h` files) are **mandatory** to be commented in a doxygen compliant manner!

For a more detailed overview of the most important commands to produce a clean doxygen-conform documentation and commenting style, you can have a further look at **Doxygen: Documenting the code** and **Doxygen: Special Commands** . In addition, you can also use the short reference in `/doc/se/code_commenting/doxygen_quickReference.pdf`.

3 Coding style guide

3.1 Naming style

All variables and functions shall have meaningfull names that express the semantic and/or logical use in the code.

It is recommended (but not mandatory) to use the **Hungarian Notation** (see Wikipedia on Hungarian Notation). Especially the prefix-postfix notation style is helpful to clarify the scope and data type of the variable's content.

Explicitly forbidden function or variable names are such like `i`, `temp`, `foo`, `bar`, `test`, every combination of the given names and all similar words and expressions without descriptive manner.

3.1.1 Naming structure for defines & macros

The recommended naming structure for non-local defines (almost all relevant ones) is:

`[scope]_[layer]_[functionalUnit]_defName_[dataType]` (3.1)

For strictly local defines, the naming structure may be simplified to the pattern:

`[scope]_defName_[dataType]` (3.2)

Example of recommendation:

```
1 /*!
2 Pattern:  [global]_[hal]_[gps]_defName_[unsigned 32bit interger]
3 */
4 #define G_HAL_GPS_HWADDR_U32;
```

All letters of the define/macro name shall be capitals. The **scope** and the **data type** (only defines) shall be given according to table 3.1 (table for scope of variables) and table 3.3.

For macros, the data type can be omitted if not necessary.

3.1.2 Naming structure for variables

The recommended naming structure for non-local variables is:

`[scope]_[layer+functionalUnit]_varName_[dataType]` (3.3)

For strictly local variables, the naming structure may be simplified to the pattern:

`[scope]_varName_[dataType]` (3.4)

Example of recommendation:

```
/*!
Pattern:  [global]_[hal/gps]_varName_[unsigned 32bit interger]
*/
unsigned int g_halGps_gpsStateNumber_ui32;

/*
Pattern:  [local]_varName_[pointer of chars]
          local pointer on array of ASCII chars
*/
char* l_stringIndex_pch;
```

The **scope** and **data type** of variables shall be given according to table 3.1 and table 3.3.

Additionally to the scope and data type, for non-local variables, the **hierarchical position** of the variable (`layer + functionalUnit`) shall be given, as shown in fig.1.1. The respective layer has to be given first, followed by the name of the functional unit. The layer's name and the functional unit's name shall be separated by a capitalized first letter of the functional unit's name (see example above).

3.1.3 Naming structure for functions

The recommended structure for function naming is:

`[scope]_[layer+functionalUnit]_funcName_[dataType]` (3.5)

For strictly local functions (without interface declaration in the header file), the naming structure may be simplified to the pattern:

`[scope]_funcName_[dataType]` (3.6)

The **scope** and **data type** of functions shall be given according to table 3.2 and table 3.3. The data type shall represent the **data type of the return variable**.
Additionally to the scope and data type, for non-local functions, the hierarchical position of

the function (`layer + functionalUnit`) shall be given, as shown in fig.1.1. The respective layer has to be given first, followed by a the name of the functional unit separated by a capitalized first letter of the functional unit's name.

Example of recommendation:

```
/*
  Pattern:  [local]_[hal/gps]_funcName_[single byte character]
  ( remark: the scope of function parameters are given by the literal 'f'
            which equals a local scope, but indicates additionally its
            origin of the parameter list)
*/
char l_halGps_readByte_ch( int f_timeout_i32 );
```

3.1.4 Literals of scope

The scope of variables and functions differ in one level of hierarchy. As a consequence, both cases (scope of variables and scope of functions) are shown in detail below to clarify the tiny but relevant differences (see table 3.1 and table 3.2).

Scope	Scope description	Literal
Global (**variable**)	The scope of the **variable** comprises the whole software and/or the complete layer it belongs to (as shown in fig1.1). ***Example:*** *A variable that is used in at least two HAL drivers, e.g. in* `hal/gps` *and* `hal/imu`*.*	g
Module-wide (**variable**)	A **variable** that is used only within a functional unit. The functional unit equals the boxes within one layer, as depicted in fig.1.1. ***Example:*** *A variable used in at least two functions of the HAL-driver of the GPS sensor.*	m
Local (**variable**)	A **variable** is local if its scope is strictly limited to the function it belongs to. A **special case** are function parameters (local to the function's body) that shall be denoted with `f_...`! ***Example:*** *A variable declared in the body of a function, or a function parameter.*	l (f)

Table 3.1: Literals for the scope of variable naming

Scope	Scope description	Literal
Global (**function**)	A **function** is global if it shall be callable across the borders of the layer it belongs to (**interface between layers**). ***Example:*** *A function of the HAL Layer shall be callable by a function of the SIG Layer.*	g
Module-wide (**function**)	The scope of a **function** is module-wide if it shall be callable across the borders of the functional unit it belongs to. (**interface between functional units**) ***Example:*** *A function of GPS Filter in SIG Layer (`sig/gps_filt`) shall be callable by a function of the Orientation Fusion of the same SIG Layer (`sig/orient_fusion`).*	m
Local (**function**)	The scope of a **function** is local if it shall be callable only within a single functional unit it belongsto. No function declaration may be given in the header file! ***Example:*** *A function of the GPS driver in HAL shall be callable by another function in the same GPS driver in HAL.*	l

Table 3.2: Literals for the scope of function naming

3.1.5 Literals of data types

Data type	Size	Signedness	C-data type	Literal
integer	8bit	signed	`signed char`	i8
		unsigned	`unsigned char`	ui8
	16bit	signed	`signed short`	i16
		unsigned	`unsigned short`	ui16
	32bit	signed	`signed int`	i32
		unsigned	`unsigned int`	ui32
	64bit	signed	`signed long`	i64
		unsigned	`unsigned long`	ui64
float	32bit	-	`float`	f32
	64bit	-	`double`	f64
boolean	-	-	(e.g. `unsigned int`)	bl
ASCII byte	(8bit)	-	`char`	ch
enum	-	-	`enum strName{ ... }`	en
struct	-	-	`struct strName{ ... }`	st
union	-	-	`union unName{ ... }`	un
void	-	-	`void`	vd
array	-	-	`char arrayName[32]`	rg...
pointers	-	-	dataType*	p...
function pointer	32bit (address)	-	-	fp

Table 3.3: Literals for the data types of variable and function naming

Pre-allocated arrays

For pre-allocated arrays, the literal `rg` (short for 'range') plus the **size of the pre-allocated array** shall be given.
Example:

```
// local pre-allocated array with 255 elements of unsigned 8bit integers
unsigned char l_myArray_rg255ui8[255];
```

Pointers

The literal for **pointers** is given by `p`, followed by the literal of the referenced data type (as depicted in table 3.3). Nevertheless, a function pointer is indicated by the fixed literal `fp`. Examples:

```
// local pointer on unsigned 8bit integers (e.g. array of 8bit values)
char*   l_measurementArray_pui8;

// local pointer on ASCII bytes (e.g. string)
char*   l_measurementArray_pch;
```

```
6
7 //! global pointer on float (32bit)
8 float*  g_appLanding_floatReference_pf32;
```

3.2 Comments

All code listings of this section can be found in SVN `/doc/se/code_commenting` as text files for copy & paste purposes.

3.2.1 Type definitions and non-local variables

Type definitions as well as non-local variables (especially enums and structures and unions) shall be declared in the header file of the respective functional unit or layer. Additionally a doxygen-conform commenting is mandatory.

Template with exemplary content:

```
/*!*****************************************************************
 * \author    Juergen Schmidt (juscgs00)
 * \date      2014/04/01
 *
 * \brief   Enumeration of all possible GPS sensor states.
 * \details A detailed descripction about the purpose of the variable
 *          or typedef. Multiple lines of description is possible.
 *          Doxygen will recognize the format automatically.
 *
 * \internal
 * CHANGELOG:
 * 2015/04/02 olbrgs00:
 * Changed internal blaa production to foo creation
 *
 * \endinternal
 *****************************************************************/
typedef enum e_gpsStates{
  GPS_UNDEF_EN,              //!< undefined state
  GPS_INIT_EN,               //!< startup of sensor
  GPS_UNFIXED_SIGNAL_EN,     //!< sensor is searching for satellites
  GPS_FIXED_SIGNAL_EN,       //!< found enough satellites
  GPS_ERROR_EN               //!< non-functional operation
} gpsStatesType;

/*!
  the following line is intentionally commented improperly, since it shall
  only clarify the context of the typedef above. A complete commenting
  block - as depicted for the enum type definition above - would be
  required for the global variable below!
*/
gpsStatesType g_halGps_opState_en;
```

3.2.2 Interfaces

Interfaces (function declarations in `.h` files) shall be grouped to a reasonable set of interface declarations. Each set shall be commented by the following block. Due to readability, the maximal line width within a block comment shall not exceed 75 characters.

Template with exemplary content:

```
/* ---------------------------------------------------------------------------
 * AUTHOR: Juergen Schmidt (juscgs00)
 * DATE OF CREATION: 2014/04/01
 *
 * DESCRIPTION:
 * Interfaces to access all good foos and producting cumquats
 *
 * CHANGELOG:
 * 2015/04/02 olbrgs00:
 * Changed internal blaa production to foo creation
 * --------------------------------------------------------------------------- */
int    m_halGps_getFooToCumquat_i32(int, float);
int    m_halGps_SetFooToCumquat_i32(int, float);
int    m_sigGpsfilt_getCumquatToFoo_int(float, int);
void   g_sigGpsfilt_updateFooStorage_vd(void);
void   g_appOrient_updateCumquatStorage_vd(void);
```

3.2.3 Functions

Function definitoins, given in `.c` files, are commented by the following block. Input parameters are numbered in a left to right order. Due to readability, the maximal line width within a block comment shall not exceed 75 characters.

Important remark:
The definitions of functions (`.c` files) shall be commented in a doxygen compliant manner in order guarantee a comprehensive code documentation!

Template with exemplary content:

```
/*!*****************************************************************
 * \author    Juergen Schmidt (juscgs00)
 * \date      2014/04/01
 *
 * \brief   Brief description in one line about the function.
 * \details A detailed descripction about the function, here for creating
 *          good foos and much more. Multiple lines of description is
 *          possible. Doxygen will recognize the format automatically.
 *
 * \param[in]   f_paramNameOne is a 'call by value' parameter
 * \param[in]   f_paramNameTwo is a 'call by reference' parameter
 * \param[out]  return value of function
 *
 * \internal
 * CHANGELOG:
 * 2015/04/02 olbrgs00:
 * Changed internal blaa production to foo creation
 *
 * \endinternal
 *****************************************************************/
int m_sigGpsfilt_fooCreator_i32(float f_paramNameOne, int f_paramNameTwo)
{
  int l_fooValue_i32;
  int l_cumquatValue_i32;

  l_fooValue_i32      = 22;
  l_cumquatValue_i32  = 20;

  return (l_fooValue_i32 + l_cumquatValue_i32);
}
```