

INTERNSHIP REPORT

**EMBEDDED SYSTEM**  
**for**  
**QUADRICOPTER**

Bùi Nhã Đạt

Poitiers, spring 2014

# Abstract

This study carries out an general approach to the embedded system. The report will be presented in two main parts. The first part is the review of the data fusion theory that applied for an IMU sensor, and the general introduction of the real-time system design methods SA-RT and DARTS. In the second part, many technical issues concerning to embedded system, on both hardware and software aspect, can be found. The most common issues can be listed here, such as:

- Choosing a real-time operating system, a development environment (compiler, linker, loader);
- Acquiring sensor measurements;
- Designing a multitasking system for implementing a control system on the embedded system.

In additional, in this report, Chapter 6 will represent an implementation of a communication protocol; Chapter 8 will discuss about translating a control system designed in MATLAB/Simulink into C code and integrating it into the embedded system, by using a third-party software.

There are also the technical reports have been carried out together with this report, to give a more detailed look into the technical issues.

# Acknowledgements

The author would like to give special thanks  
to his *Family* for the love, support, and encouragement  
he has gotten over the years,  
to his *Advisor*,  
who never bothers spending the precious time  
answering all of his questions,  
correcting his mistakes,  
to his *homeland Professors*,  
who always encourages him,  
gives him the passion of studying,  
to his *Co-Worker*,  
who never bothers helping him  
with his language problems,  
to his *Best Friends*,  
who are always by his side.

Poitiers, winter 2013-2014

Bùi Nhã Đạt

# Table of Contents

<b>Abstract</b> .....	<b>ii</b>
<b>Acknowledgements</b> .....	<b>iii</b>
<b>Table of Contents</b> .....	<b>iv</b>
<b>List of Figures</b> .....	<b>vi</b>
<b>List of Tables</b> .....	<b>ix</b>
<b>Acronyms</b> .....	<b>x</b>

## PART 1. INTRODUCTION

<b>1. Introduction</b> .....	<b>3</b>
1.1 Background .....	3
1.2 Objectives .....	4
<b>2. Theory Review</b> .....	<b>5</b>
2.1 Data Fusion Algorithm .....	5
2.1.1 Introduction .....	5
2.1.2 Madgwick Data Fusion Algorithm .....	6
2.2 SA-RT Method Introduction .....	11
2.3 DART Method Introduction .....	13
2.3.1 Syntax Graph of the DARTS Method .....	13
2.3.2 SA-RT to DARTS Translation Simple Examples .....	16

## PART 2. CONTRIBUTION

<b>3. Development Environment and Real Time Operating System</b> ...	<b>19</b>
3.1 Integrated Development Environment .....	20
3.1.1 MapleIDE Installation .....	20
3.1.2 MapleIDE Testing .....	21
3.2 Real Time Operating System .....	23
<b>4. Device Preparation and Drivers</b> .....	<b>25</b>
4.1 System Overview .....	26
4.2 FlyMaple SDK and Available Resources .....	26

4.3	Integrated Sensors: the I2C Devices.....	28
4.3.1	I2C Communication .....	28
4.3.2	The Integrated I2C Devices .....	30
4.3.3	Measurement Resolution .....	30
4.4	GPS Shield.....	31
4.4.1	Hardware Preparation .....	31
4.4.2	UART Driver for GPS Shield.....	33
4.5	Wi-Fi Shield.....	36
4.5.1	Hardware Preparation .....	36
4.5.2	Network Configuration.....	37
<b>5.</b>	<b>Multitasking System Design.....</b>	<b>39</b>
5.1	System Specification .....	39
5.2	System Design.....	44
5.3	System Implementation.....	46
<b>6.</b>	<b>Communication Protocol.....</b>	<b>51</b>
6.1	Float Value Transmission.....	51
6.2	Ground-to-Air Protocol.....	53
6.2.1	General Syntax .....	54
6.2.2	Block Semantic .....	54
6.2.3	Summary.....	55
6.3	Air-to-Ground Protocol.....	56
<b>7.</b>	<b>Ground Station Design.....</b>	<b>59</b>
7.1	Overview.....	59
7.2	User Interface .....	60
7.3	Networking.....	64
<b>8.</b>	<b>Simulink Code Generation .....</b>	<b>66</b>
8.1	Preparation: Java Runtime Environment.....	67
8.2	Code Generating Procedure.....	67
8.2.1	Simulink Model Preparation.....	68
8.2.2	Code Generation .....	68
8.2.3	Code Integration .....	69
8.2.4	Summary.....	70
8.3	Test Case: the Very Simple Model.....	72
8.4	Quadricopter Flight Control System Implementation.....	73
8.4.1	Equivalent Models of Integrator and Derivative blocks.....	74
8.4.2	Quadricopter PID Flight Control System Implementation.....	79
<b>9.</b>	<b>Conclusion.....</b>	<b>91</b>
9.1	Achievements.....	91
9.2	Later Works.....	91
	<b>Bibliography .....</b>	<b>93</b>

# List of Figures

Figure 2-1 –	Block diagram representation of the complete orientation filter for an IMU implementation [2] . . . . .	10
Figure 2-2 –	Block diagram representation of the complete orientation filter for an MARG implementation including magnetic distortion (Group 1) and gyroscope drift (Group 2) compensation [2]. . . . .	10
Figure 2-3 –	Elements in SA-RT diagrams . . . . .	11
Figure 2-4 –	Generic context diagram of the specification method SA-RT [6] . . . . .	12
Figure 2-5 –	Generic functional decomposition [6]. . . . .	12
Figure 2-6 –	Representation of tasks [6]. . . . .	13
Figure 2-7 –	Representations of synchronizations between tasks [6]. . . . .	14
Figure 2-8 –	Representations of communication between tasks [6] . . . . .	14
Figure 2-9 –	Representations of physical task activations [6] . . . . .	15
Figure 2-10 –	Representations of software task activations [6]. . . . .	15
Figure 2-11 –	Representation of memory modules [6]. . . . .	16
Figure 2-12 –	Translation of functional data acquisition process into an periodic task [6] . . . . .	16
Figure 2-13 –	Translation of functional data acquisition process into an aperiodic task [6] . . . . .	16
Figure 2-14 –	Translation of a direct dataflow between two functional processes using mailbox [6]. . . . .	16
Figure 2-15 –	Translation of a direct dataflow between two functional processes using memory module [6] . . . . .	16
Figure 3-1 –	Working modes of FlyMaple. Left: perpetual boot loader mode – DFU driver, right: normal mode – serial driver . . . . .	21
Figure 3-2 –	Code uploading modes . . . . .	22
Figure 4-1 –	Block diagram of embedded system . . . . .	26
Figure 4-2 –	Code annotating loop . . . . .	27
Figure 4-3 –	Annotated code and corresponding generated documentation . . . . .	27

Figure 4-4 – I2C bus [6].....	29
Figure 4-5 – I2C bus on the FlyMaple.....	30
Figure 4-6 – I2C1 header on FlyMaple.....	32
Figure 4-7 – Remapping GPS shield TX pin .....	32
Figure 4-8 – Sample data returned from LEA-5H GPS shield in normal operating mode.....	33
Figure 5-1 – Complete block diagram.....	40
Figure 5-2 – Quadricopter context diagram .....	40
Figure 5-3 – Preliminary diagram of Quadricopter control system.....	41
Figure 5-4 – Simplified preliminary diagram of Quadricopter control system	43
Figure 5-5 – Decomposition of Process 1.....	44
Figure 5-6 – Quadricopter DARTS diagram.....	45
Figure 5-7 – Memory module definition and initialization.....	47
Figure 5-8 – Update a memory module .....	47
Figure 6-1 – Ground data structure .....	54
Figure 6-2 – “Orders” block.....	54
Figure 6-3 – “Gains” block.....	55
Figure 6-4 – “Mode” block.....	55
Figure 7-1 – Ground Station user interface – Monitor & Control tab .....	60
Figure 7-2 – Ground Station user interface – Network tab.....	61
Figure 7-3 – UI Updaters.....	62
Figure 7-4 – Event handlers – orders .....	62
Figure 7-5 – Event handlers – gains tuner .....	62
Figure 7-6 – Event handlers – mode.....	63
Figure 7-7 – Graphic rendered by the Artificial Horizon module.....	63
Figure 7-8 – Ground Station Networking Module.....	64
Figure 8-1 – Setting environment variables in Windows <sup>®</sup> OS .....	67
Figure 8-2 – JRE installed correctly (upper) and not correctly (lower).....	67
Figure 8-3 – Gen-Auto command lines.....	68
Figure 8-4 – Equivalent discreet derivative model .....	74
Figure 8-5 – Equivalent integrator model: Forward Euler method .....	75
Figure 8-6 – Equivalent integrator model: Backward Euler method.....	76
Figure 8-7 – Equivalent integrator model: Trapezoidal method.....	77
Figure 8-8 – Test bench for the equivalent derivative model .....	77

Figure 8-9 – Test bench for the equivalent integrator model . . . . .	78
Figure 8-10 – The complete Quadricopter simulating model with PID controller (some components removed) . . . . .	79
Figure 8-11 – The PID block, original design . . . . .	79
Figure 8-12 – PID controller for z-channel (“PID_z” sub-block), original design . . . . .	80
Figure 8-13 – The PID block, modified version . . . . .	81
Figure 8-14 – The “Controller” sub-block of the modified “PID” block . . . . .	82
Figure 8-15 – PID controller for z-channel (“PID_z” sub-block), modified version using alternative derivative and integrator blocks . . . . .	82
Figure 8-16 – Setting a block to be generated as separated module . . . . .	84



# List of Tables

Table 3-1 –	RTOSes that is compatible with FlyMaple.....	23
Table 4-1 –	Available resources for FlyMaple controller board .....	28
Table 4-2 –	Structure of GPGGA sentences.....	33
Table 4-3 –	GPGGA parsing method.....	35
Table 4-4 –	AT commands .....	37
Table 5-1 –	List of resources.....	46
Table 6-1 –	Methods for transmitting a float value.....	53
Table 6-2 –	Control characters used in communication protocol.....	55
Table 6-3 –	Three block types of ground data .....	55
Table 6-4 –	Air-to-Ground protocol semantic.....	56
Table 8-1 –	JRE environment variables in Windows <sup>®</sup> OS .....	67
Table 8-2 –	Three main steps of code generating procedure.....	70
Table 8-3 –	Comparison of equivalent models to the built-in blocks .....	78
Table 8-4 –	Controller I/O ports.....	83
Table 8-5 –	Generated modules .....	83
Table 8-6 –	Additional modules.....	84
Table 8-7 –	Additional I/O variables .....	85
Table 8-8 –	Code segment of the PID_<channel> blocks .....	85
Table 8-9 –	Code segment of the PID_<channel> blocks (treated as atomic units) .....	86
Table 8-10 –	Gene-Auto generated code metrics .....	87

# Acronyms

AHRS	Attitude and heading reference system
BAL	Mailbox ( <i>Boîte aux lettres</i> )
CG	Watch dog ( <i>Chien de garde</i> )
DCM	Direction cosine matrix
DFU	Device firmware upgrade
DHCP	Dynamic Host Configuration Protocol
DOF	Degree of freedom
E/D	Enable/disable
FIFO	First in first out
GPGL	Global positioning system fix data
GPIO	General-purpose input/output
GPS	Global positioning system
HTR	Real time clock ( <i>Horloge temps réel</i> )
I2C	Inter-integrated circuit
IDE	Integrated development environment
IMU	Inertial measurement unit
IT	Interruption
JRE	Java runtime environment
JST	Japan solderless terminal (connector)
MARG	Magnetic, angular rate, and gravity

MCU	Microcontroller
MDD	Memory module ( <i>Module de donnée</i> )
NMEA	National Marine Electronics Association
OS	Operating system
PID	Proportional-integral-derivative
PWM	Pulse-width modulation
RTC	Real time clock
RTOS	Real-time operating system
RX	Receive, receiver
SA-RT	Structured analysis for real-time systems
SCL	Serial clock
SDA	Serial data
SDK	Software development kit
T	Trigger
TCP	Transmission Control Protocol
TX	Transmit, transmitter
UART	Universal asynchronous receiver/transmitter
UDP	User Datagram Protocol
UI	User interface

Part 1

INTRODUCTION

---

• <i>Chapter 1</i> – Introduction . . . . .	3
• <i>Chapter 2</i> – Theory Review . . . . .	5



# Chapter 1

## Introduction

### 1.1 Background

Nowadays, embedded system can be found everywhere—may be home appliance (washing machines, microwave ovens...), portable devices (such as cell phones, MP3 players, digital cameras...), or in automotive (such as fuel injection system, anti-lock brakes), and many other devices.

It is not easy to define precisely the term “embedded system.” Simply say, embedded system is a system that do the dedicated functions, often with real-time computing constraints. It is embedded into a larger system that consists of other hardware and mechanical parts.

By implementing a flight control system on embedded system in a real application—the quadricopter, this study will solve many issues concerning to embedded system as well as real-time multitasking system.

## 1.2 Objectives

The Laboratory Informatics and Automation Systems (*Laboratoire d'Informatique et d'Automatique pour les Systèmes – LIAS*) has built a mini drone quadricopter equipped with a micro-controller, GPS, Wi-Fi connection and an inertial measurement unit.

The purpose of this project is to develop the embedded system on this quadricopter, so as to integrate a flight controller developed as a part of another PFE (*projet de fin d'études*). It comprises these parts:

- Choosing a real-time operating system, a development environment (compiler, linker, loader);
- Developing a library to read the sensor measurement and to control the actuators (the motors);
- Designing a multitasking system for implementing the flight control and navigation system;
- Proposing and implementing a communication protocol for communicating between the quadricopter and the ground station;
- Implementing a ground station;
- Performing the unit testing, integration testing...

Flight control and navigation algorithms to implement will be provided in the form of Simulink models, produced as parts of another PFE. Since then, these models need to be translated into the form of code, which can be integrated directly into the embedded system.

## Chapter 2

# Theory Review

## 2.1 Data Fusion Algorithm

### 2.1.1 Introduction

An inertial measurement unit, or IMU, measures accelerations and rotation rates, and possibly earth’s magnetic field, in order to determine a body’s attitude. Measurements from accelerator, gyroscope, and magnetometer if available, are needed to be fused into the more usable value—the attitude. There are several works with this issue have been carried out, and several algorithms have been developed. For example:

- The **Kalman filter** (*see* [1]): accurate and effective; however, it is complicated to implement, “demand sampling rates far exceeding the subject bandwidth [2],” require large state vectors so demand a large computational load.
- The **Complementary filter** (developed by Mahony, *see* [3], [4]): efficient and effective, simple to implement, low computational loading; however, its performance is only validated for an IMU (the magnetometer data is useless in this data fusion algorithm),
- **Direction Cosine Matrix (DCM) IMU algorithm** (developed by W. Premierlani, *see* [5]): only six degrees of freedom like complementary, and still in the developing phase—“actually, at this point, it is still a draft, there is still a lot more work to be done.” [5]
- The IMU+MARG (*magnetic, angular rate, and gravity*) filter, as known as the **Mahony & Madgwick filter** (developed by Madgwick, *see* [2]): “The MARG implementation incorporates magnetic distortion and gyroscope bias drift compensation. The filter uses a quaternion representation, allowing accelerometer, and magnetometer data to be used in an analytically derived [...]. The benefits of the filter include: (1) computationally



inexpensive; requiring 109 (IMU) or 277 (MARG) *scalar* arithmetic operations each filter update, (2) effective at low sampling rates; e.g. 10 Hz, and (3) contains 1 (IMU) or 2 (MARG) adjustable parameters defined by observable system characteristics.” [2]

Due to the scope of this project, we are not going to focus on analyzing each algorithm, but implementing it.

In addition to the accelerometer and gyroscope as in every 6 DOF IMU, our hardware also integrates a magnetometer to form a MARG sensor, so the Kalman filter and the Madgwick filter are the algorithms that can be applied for a 9 DOF MARG. One additional point, for implementing on embedded system with limited resources, the simpler the better; for this point, the Madgwick filter is the winner. Finally, there is an implementation of Madgwick filter that is ready-to-use in the FlyMaple SDK (*see Part 4.2*), we just need to tune some parameters. Since then, the data fusion algorithm used in this project will be the Madgwick filter.

A literature of this algorithm can be found very clearly in the material [2] of Madgwick. Here, we will not repeat it, but only describe the idea—the “philosophy”—of the algorithm.

## 2.1.2 Madgwick Data Fusion Algorithm

### 2.1.2.1 Approaching

The work of attitude estimation corresponds to evaluating (computationally) the kinematic equation for the rotation of a body:

$$\dot{\mathbf{R}} = \mathbf{R}(\boldsymbol{\omega}\mathbf{J}) = \mathbf{R} \begin{pmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{pmatrix}$$

where  $\boldsymbol{\omega} = (\omega_x, \omega_y, \omega_z)^T$  is the measure rotation rate vector, obtained from the gyroscope. The rotation  $\mathbf{R}$  represents the orientation of the body-fixed reference frame as observed in the earth reference frame. For any vector  $\mathbf{v}$ , the coordinates  $\mathbf{v}_{Earth}$  with respect to the earth frame become in the body-fixed frame  $\mathbf{v}_{body} = \mathbf{R}^T \mathbf{v}_{Earth}$ , which evolve as  $\dot{\mathbf{v}}_{body} = (\boldsymbol{\omega}\mathbf{J})^T \mathbf{v}_{body} = -(\boldsymbol{\omega} \times \mathbf{v}_{body})$  (the minus sign comes in here since  $\boldsymbol{\omega}$  represents the rotation of the body-fixed coordinate system).

This work can be divided into three main tasks:

- Integrate rate of change of the quaternion to represent  $\mathbf{R}$ .
- Apply “magnetic distortion compensation” and “gyroscope bias drift compensation.”

- Improve the attitude estimate by fusing gyroscope data with accelerometer and magnetometer data.
- Take care that the numerical representation of  $\mathbf{R}$  represents in fact a “real” physical rotation. In Madgwick’s algorithm, quaternion normalization is applied.

Madgwick calls the orientation estimated from gyroscope measurement the “orientation from angular rate,” and the orientation estimated from accelerometer and magnetometer measurements the “orientation from vector observations.”

### 2.1.2.2 Orientation from angular rate

A tri-axis gyroscope will measure the angular rate about the three axes of the sensor frame. Arranging this measurement in form of a vector as  ${}^S\boldsymbol{\omega} = [0 \ \omega_x \ \omega_y \ \omega_z]$ . Then, the quaternion derivative describing the rate of change of orientation of the earth frame relative to the sensor frame can be calculated by  ${}^S\dot{\mathbf{q}} = \frac{1}{2} {}^S\mathbf{q} \otimes {}^S\boldsymbol{\omega}$ , assuming that the current orientation  ${}^S\mathbf{q}$  is defined. The orientation of the earth frame relative to the sensor frame at time  $t$  can be computed by numerically integrating the quaternion derivative  ${}^S\dot{\mathbf{q}}_{\omega,t}$ .

$${}^S\dot{\mathbf{q}}_{\omega,t} = \frac{1}{2} {}^S\mathbf{q}_{est,t-1} \otimes {}^S\boldsymbol{\omega}_t \quad (2-1)$$

$${}^S\mathbf{q}_{\omega,t} = {}^S\mathbf{q}_{est,t-1} + {}^S\dot{\mathbf{q}}_{\omega,t} \Delta t \quad (2-2)$$

In these equations,  ${}^S\boldsymbol{\omega}_t$  is the angular rate measured at time  $t$ ,  $\Delta t$  is the sampling period,  ${}^S\mathbf{q}_{est,t-1}$  is the previous estimation of orientation.

### 2.1.2.3 Orientation from vector observations

A tri-axis accelerometer will measure the magnitude and direction of the field of gravity in the sensor frame compounded with linear accelerations due to motion of the sensor. Similarly, a tri-axis magnetometer will measure the magnitude and direction of the Earth’s magnetic field in the sensor frame compounded with local magnetic flux and distortions. In the context of an orientation filter, it will initially be assumed that an accelerometer will measure only gravity; a magnetometer will measure only the Earth’s magnetic field.

If the direction of an earth’s field is known in the earth frame, a measurement of the field’s direction within the sensor frame will allow an orientation of the sensor frame relative to the earth frame to be calculated. Using quaternions for representing the orientations, this may be achieved through the formulation of an optimization problem where an orientation of the sensor,  ${}^S\mathbf{q}$ , is that which aligns a predefined reference direction of the field in the earth frame,  ${}^E\mathbf{d}$ , with the measured direction of the field in the sensor frame,  ${}^S\mathbf{s}$ . Therefore  ${}^S\mathbf{q}$ , may be found as the solution to (2-3) where equation (2-4) defines the objective function [2]. The components of each vector are defined in equations (2-5) to (2-7).

$$\min_{\substack{S \\ E} \mathbf{q} \in \mathfrak{R}^4} \mathbf{f}(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d}, \substack{S \\ S} \mathbf{s}) \quad (2-3)$$

$$\mathbf{f}(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d}, \substack{S \\ S} \mathbf{s}) = \substack{S \\ E} \mathbf{q}^* \otimes \substack{S \\ E} \mathbf{q} \otimes \substack{E \\ S} \mathbf{d} - \substack{S \\ S} \mathbf{s} \quad (2-4)$$

$$\substack{S \\ E} \mathbf{q} = [q_1 \quad q_2 \quad q_3 \quad q_4] \quad (2-5)$$

$$\substack{E \\ S} \mathbf{d} = [0 \quad d_x \quad d_y \quad d_z] \quad (2-6)$$

$$\substack{S \\ S} \mathbf{s} = [0 \quad s_x \quad s_y \quad s_z] \quad (2-7)$$

In Madgwick's algorithm, he uses the gradient descent method to solve this equation. Equation (2-8) calculates the estimated orientation  $\substack{S \\ E} \mathbf{q}_{\nabla, t}$  computed at time  $t$  based on a previous estimate of orientation  $\substack{S \\ E} \mathbf{q}_{est, t-1}$  and the objective function gradient  $\nabla \mathbf{f}$  defined by sensor measurements sampled at time  $t$  [2].

$$\substack{S \\ E} \mathbf{q}_{\nabla, t} = \substack{S \\ E} \mathbf{q}_{est, t-1} - \mu_t \frac{\nabla \mathbf{f}}{\|\nabla \mathbf{f}\|} \quad (2-8)$$

$$\nabla \mathbf{f}(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d}, \substack{S \\ S} \mathbf{s}) = \mathbf{J}^T(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d}) \mathbf{f}(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d}, \substack{S \\ S} \mathbf{s}) \quad (2-9)$$

$\mathbf{J}$  is the Jacobian of the objective function  $\mathbf{f}$ . The sub-script  $\nabla$  indicates that the quaternion is calculated using the gradient descent algorithm. The form of  $\nabla \mathbf{f}$  is chosen according to the sensors in use.  $\mathbf{J}(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d})$  and  $\mathbf{f}(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{d}, \substack{S \\ S} \mathbf{s})$  can be simplified to the form of 3-row vector (*see [2] for the formulae*).

An appropriate convention would be to assume that the direction of gravity defines the vertical,  $z$  axis as shown in equation (2-10). If there is only the accelerometer available, gravity orientation  $\substack{E \\ S} \mathbf{g}$  and normalized accelerometer measurement  $\substack{S \\ S} \mathbf{a}$  will be substituted respectively for  $\substack{E \\ S} \mathbf{d}$  and  $\substack{S \\ S} \mathbf{s}$ .

$$\substack{E \\ S} \mathbf{g} = [0 \quad 0 \quad 0 \quad 1] \quad (2-10)$$

$$\substack{S \\ S} \mathbf{a} = [0 \quad a_x \quad a_y \quad a_z] \quad (2-11)$$

Since  $\substack{E \\ S} \mathbf{g}$  is fixed,  $\mathbf{J}_g^T(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{g}) = \mathbf{J}_g^T(\substack{S \\ E} \mathbf{q}) \mathbf{f}_g$  and  $\mathbf{f}_g(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{g}, \substack{S \\ S} \mathbf{a}) = \mathbf{f}_g(\substack{S \\ E} \mathbf{q}, \substack{S \\ S} \mathbf{a})$ .

If there are both accelerometer and magnetometer measurements available, the objective function  $\mathbf{f}$  will be defined in a different way. The Earth's magnetic field can be considered to have components in one horizontal axis and one vertical axis, which can be represented by equation (2-12). Now,  $\substack{E \\ S} \mathbf{b}$  and normalized magnetometer measurement  $\substack{S \\ S} \mathbf{m}$  will be substituted respectively for  $\substack{E \\ S} \mathbf{d}$  and  $\substack{S \\ S} \mathbf{s}$ . Now,  $\mathbf{J}_b^T(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{b})$  and  $\mathbf{f}_b(\substack{S \\ E} \mathbf{q}, \substack{E \\ S} \mathbf{b}, \substack{S \\ S} \mathbf{m})$  are defined.

$$\substack{E \\ S} \mathbf{b} = [0 \quad b_x \quad 0 \quad b_z] \quad (2-12)$$

$$\substack{S \\ S} \mathbf{m} = [0 \quad m_x \quad m_y \quad m_z] \quad (2-13)$$

However, the measurement of gravity or the Earth's magnetic field alone will not provide a unique orientation of the sensor [2]. To do so, the measurements and reference directions of both fields may be combined as described by equations (2-14) and (2-15) [2].

$$\mathbf{f}_{g,b}(\mathbf{q}_E^S, \mathbf{b}^E, \mathbf{a}^S, \mathbf{m}^S) = \begin{bmatrix} \mathbf{f}_g(\mathbf{q}_E^S, \mathbf{a}^S) \\ \mathbf{f}_b(\mathbf{q}_E^S, \mathbf{b}^E, \mathbf{m}^S) \end{bmatrix} \quad (2-14)$$

$$\mathbf{J}_{g,b}^T(\mathbf{q}_E^S, \mathbf{b}^E) = \begin{bmatrix} \mathbf{J}_g^T(\mathbf{q}_E^S) \\ \mathbf{J}_b^T(\mathbf{q}_E^S, \mathbf{b}^E) \end{bmatrix} \quad (2-15)$$

In short, equation (2-9) can be written as:

$$\nabla \mathbf{f} = \begin{cases} \mathbf{J}_g^T(\mathbf{q}_E^S) \mathbf{f}_g(\mathbf{q}_E^S, \mathbf{a}^S) \\ \mathbf{J}_{g,b}^T(\mathbf{q}_E^S, \mathbf{b}^E) \mathbf{f}_{g,b}(\mathbf{q}_E^S, \mathbf{b}^E, \mathbf{a}^S, \mathbf{m}^S) \end{cases} \quad (2-16)$$

#### 2.1.2.4 Filter fusion algorithm

An estimated orientation of the sensor frame relative to the earth frame,  $\mathbf{q}_{est,t}^S$ , is obtained through the fusion of the orientation calculations,  $\mathbf{q}_{\omega,t}^S$  and  $\mathbf{q}_{\nabla,t}^S$ , calculated using equations (2-2) and (2-8) respectively. The fusion is described by equation (2-17) where  $\gamma_t$  and  $(1 - \gamma_t)$  are weights applied to each orientation calculation.

$$\mathbf{q}_{est,t}^S = \gamma_t \mathbf{q}_{\nabla,t}^S + (1 - \gamma_t) \mathbf{q}_{\omega,t}^S, \quad 0 \leq \gamma_t \leq 1 \quad (2-17)$$

An optimal value of  $\gamma_t$  can be defined that which ensures the weighted divergence of  $\mathbf{q}_{\omega,t}^S$  is equal to the weighted convergence of  $\mathbf{q}_{\nabla,t}^S$ . This can be described as an equation of the convergence rate  $\frac{\mu_t}{\Delta t}$  of  $\mathbf{q}_{\omega,t}^S$ , and the divergence rate  $\beta$  of  $\mathbf{q}_{\nabla,t}^S$ .

By assuming that  $\mu_t$  used in the equation (2-8) is large enough, Madgwick shows that, equation (2-17) can be rewritten as equation (2-18).

$$\mathbf{q}_{est,t}^S = -\beta \Delta t \frac{\nabla \mathbf{f}}{\|\nabla \mathbf{f}\|} + (\mathbf{q}_{est,t-1}^S + \dot{\mathbf{q}}_{\omega,t}^S \Delta t) \quad (2-18)$$

By defining the estimated rate of change of orientation as  $\dot{\mathbf{q}}_{\epsilon,t}^S$  in (2-21), and the direction of the error as  $\dot{\mathbf{q}}_{est,t}^S$  in (2-20), equation (2-18) becomes equation (2-19).

$${}^S_E \hat{\mathbf{q}}_{est,t} = {}^S_E \hat{\mathbf{q}}_{est,t-1} + {}^S_E \dot{\hat{\mathbf{q}}}_{est,t} \Delta t \quad (2-19)$$

$${}^S_E \dot{\hat{\mathbf{q}}}_{est,t} = {}^S_E \dot{\omega}_t - \beta {}^S_E \dot{\hat{\mathbf{q}}}_{\epsilon,t} \quad (2-20)$$

$${}^S_E \dot{\hat{\mathbf{q}}}_{\epsilon,t} = \frac{\nabla f}{\|\nabla f\|} \quad (2-21)$$

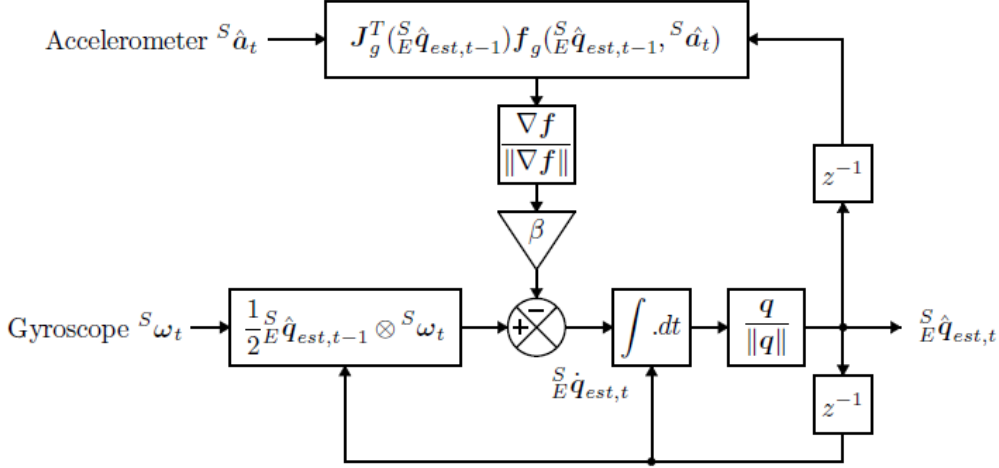


Figure 2-1 – Block diagram representation of the complete orientation filter for an IMU implementation [2]

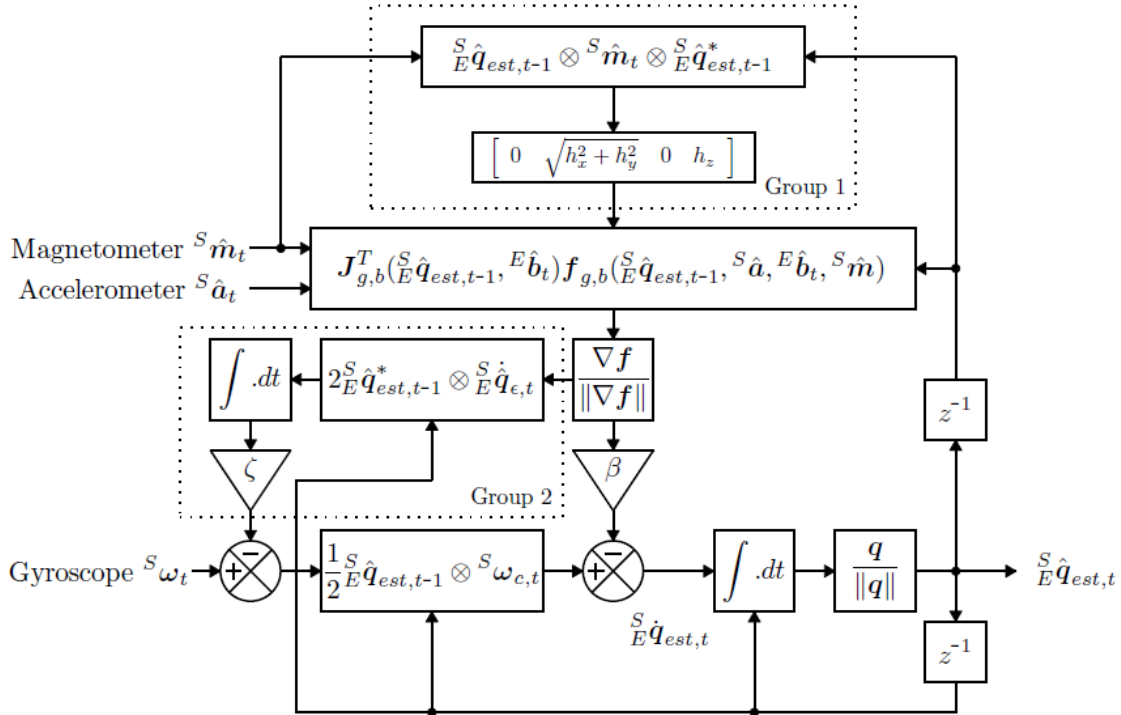


Figure 2-2 – Block diagram representation of the complete orientation filter for an MARG implementation including magnetic distortion (Group 1) and gyroscope drift (Group 2) compensation [2]

Figure 2-1 shows a block diagram representation of the complete orientation filter for an IMU implementation. When the magnetometer measurement is available, magnetic distortion compensation and gyroscope bias drift compensation

are also applied (*please refer to [2] for more details*), forming a complete filter implementation for a MARG sensor (Figure 2-2).

Further information about the formulae of objective function and its Jacobian as form of 3-row vector; the determination of the parameters  $\mu_t, \gamma_t$ ; as well as the filter gains  $\beta, \zeta$  can be found in [2].

## 2.2 SA-RT Method Introduction

SA-RT (Structured Analysis for Real-Time Systems) method is a method of functional and operational applications analysis of the control systems. This method allows a graphical and textual description of the application in terms of needs, that is to say, “what we do?” This shaping of the “specifications” of the application is formal in the sense that the methodology and expression (graphical syntax) are defined. [6]

The key word of these methods is “structuration” in the sense of a decomposition elements or functional blocks for a particular level of analysis and a coherent hierarchical decomposition between different levels of analysis. These methods of analysis or structured design naturally lead to structured programming. The second common feature of these methods is the description in the form of flows, such as data flow, control flow or otherwise. The operational aspect of the description is then visualized by the propagation of these flows. [6]

Specification of SA-RT method consists of a data context diagram, dataflow diagram and hierarchical control flow diagram, and a state-transition diagram. The elements in these diagrams are shown in the Figure 2-3.

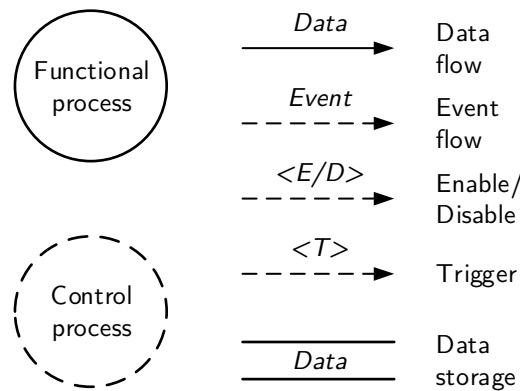


Figure 2-3 – Elements in SA-RT diagrams

The **context diagram** is an extremely important first step because it will define the context and the environment outside the system. The borders of the model or terminations will only appear in this diagram. A generic context diagram of the specification method SA-RT is shown in Figure 2-4. We can find only

one functional process, numbered “0,” which reflects the actual application carried out by the designer. Around this functional process, a set of terminators of the model delivers or consumes data or events. These terminators can be the following physical elements:

- Sensors (thermocouples, load cells, etc.);
- Actuators (motors, valves, etc.);
- Display systems (Computer screen, lamps, diode, etc.);
- Storage system (disk, tape, etc.);
- Print system (printer, paper dispenser, etc.);
- ...

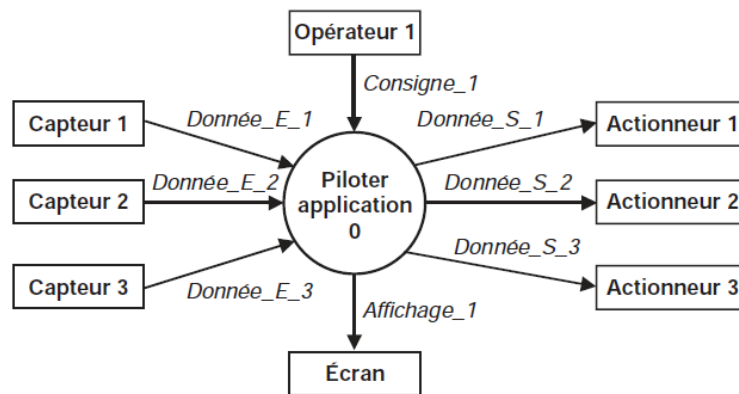


Figure 2-4 – Generic context diagram of the specification method SA-RT [6] (*piloted application: control application, opérateur: operator, capteur: sensor, actionneur: actuator, écran: screen, donnée: data, affichage: display*)

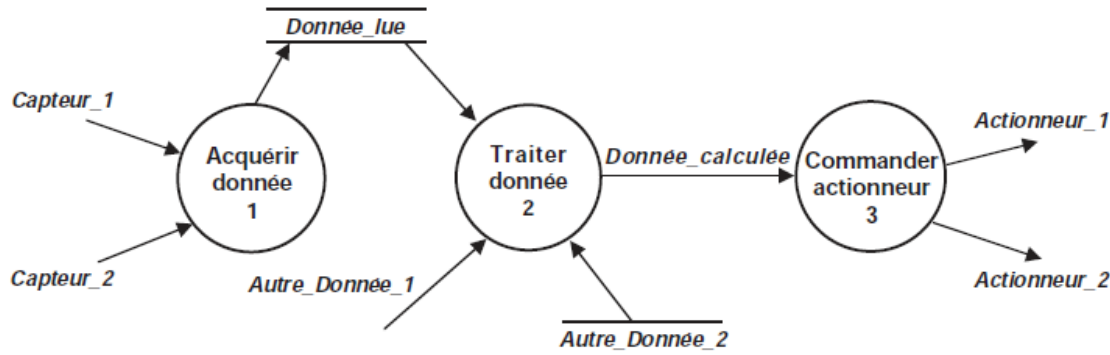


Figure 2-5 – Generic functional decomposition [6]

(Process 1, 2, 3 respectively: acquire data, treat data, command actuator; *donnée\_lue*: read\_data, *autre\_donnée*: other\_data, *donnée\_data*: calculated\_data, *capteur*: sensor, *actionneur*: actuator)

The first level of analysis is represented by the **preliminary diagram**. At this level, the diagram represents a “graphic” list of the necessary functional processes, without caring about the execution sequence. In this diagram, we can find a generic pattern to describe a function, simple or complex, by dividing it into the three basic elements:

- A data acquiring process,

- A data treating process,
- A control process.

Depending on the application, all or a part of this generic dataflow diagram may be presented for each part of the control system involved in the analysis level that is being developed. Figure 2-5 shows a generic decomposition diagram. This type of diagram can be used as both the preliminary diagram and the decomposition diagram.

## 2.3 DART Method Introduction

DARTS (Design Approach for Real-Time Systems) method is the link between the analysis method SA- RT and implementation of the application. To harmonize completely with the method of SA-RT analysis, the design method DARTS is dataflow type. Thus, the dataflow diagram of the method SA-RT (preliminary or decomposition diagrams) are translated into dataflow diagrams DARTS that represent the multitask architecture of the application. [6]

### 2.3.1 Syntax Graph of the DARTS Method

#### Modeling tasks

Task represents the basic entity of multitasking architecture. A task may have one or more inputs and one or more output dataflows (Figure 2-6). Tasks are modeled by a parallelogram that has a label. This label may correspond to that given in the SA-RT dataflow diagrams. We also have an activation signal which has a provenance and different types depending on whether we model a physical task (external activation event type) or a software task (activation internal event type: synchronization or communication). This activation signal must exist and must be unique.

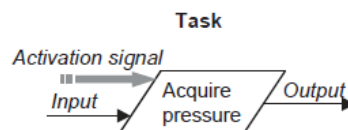


Figure 2-6 – Representation of tasks [6]

#### Modeling synchronization and communication

There are two models of synchronization: asynchronous synchronization and synchronous synchronization (rendez-vous). These models represent the flow of dependencies between tasks in terms of execution. Synchronizations are represented by an oriented symbol (Figure 2-7).



For communications, we have several models corresponding to the size of the communication and also to the management of the data storage. In general, these models “mailboxes” (*boîte aux lettres* – BAL) represent the data flow between tasks with a precedence relation. Communications are represented by an oriented symbol (Figure 2-8).

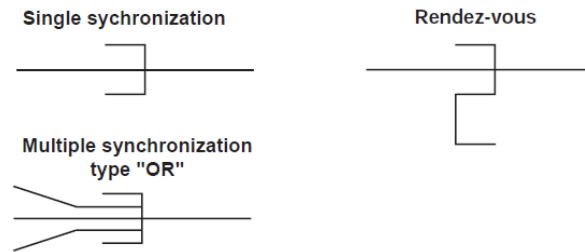


Figure 2-7 – Representations of synchronizations between tasks [6]

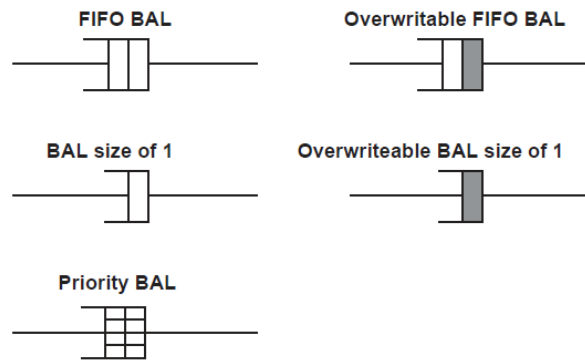


Figure 2-8 – Representations of communication between tasks [6]

We can distinguish two types of mailboxes in point of view of dependency (synchronous relationship or not):

- Blocking mailboxes that available in three models:
  - Blocking mailboxes, which can contain  $n$  messages managed as a FIFO queue,
  - Blocking mailboxes, which can contain only one message,
  - Blocking mailboxes, which can contain  $n$  messages managed as a FIFO, ordered by corresponding priority of the emitting task;
- Nonblocking mailboxes or overwriteable mailboxes that available in two models:
  - Nonblocking mailboxes, which can contain  $n$  messages managed as a FIFO queue,
  - Nonblocking mailboxes, which can contain only one message.

## Modeling task activations

As mentioned previously, there are two types of tasks: physical tasks and software tasks. In both cases, the activation is very different.

For the physical tasks that are triggered by events or external signals, we distinguish three types of signals:

- Signal “Real Time Clock – RTC” (*Horloge Temps Réel – HTR*). This signal, which comes from an internal hardware clock in the computer, corresponds to a strictly periodic signal.
- Signal “Interruption – IT.” This signal from the external process must always be considered as aperiodic due to the asynchrony of the outside world from the timing of the computer.
- Signal “Watch Dog” (*Chien de Garde – CG*). This signal is from an internal clock used as an alarm. In terms of signal, it is identical to the real time clock (internal signal), but it occurs aperiodically.

Task activations are represented by an oriented symbol (broken line) (Figure 2-9).

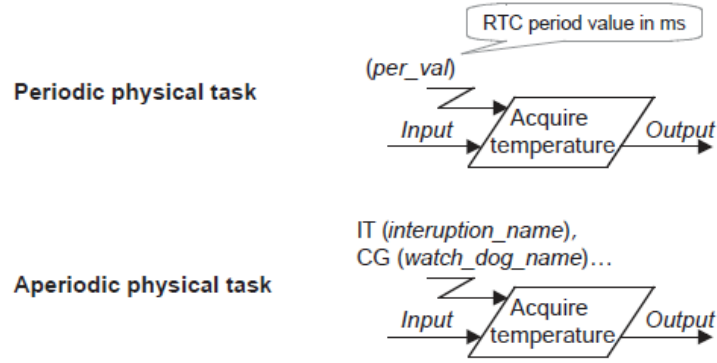


Figure 2-9 – Representations of physical task activations [6]

The software tasks are triggered by other tasks (hardware or software) with synchronization or communication mechanisms. Thus, the activation signal of these tasks can be the action of the following elements (Figure 2-10):

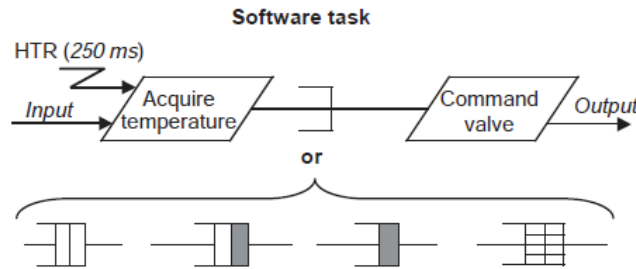


Figure 2-10 – Representations of software task activations [6]

## Modeling memory module

The last element to be modeled is the memory module (*module de donnée – MDD*) that provides a data management unit to allow two or more tasks access

to the data in *mutual exclusion* (mutex). Memory modules are represented by a rectangle associated with entries that perform an action on data.

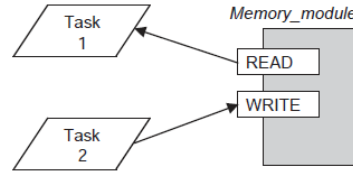


Figure 2-11 – Representation of memory modules [6]

### 2.3.2 SA-RT to DARTS Translation Simple Examples

Figure 2-12 to Figure 2-15 show some simple examples of translation from SA-RT into DARTS diagrams.

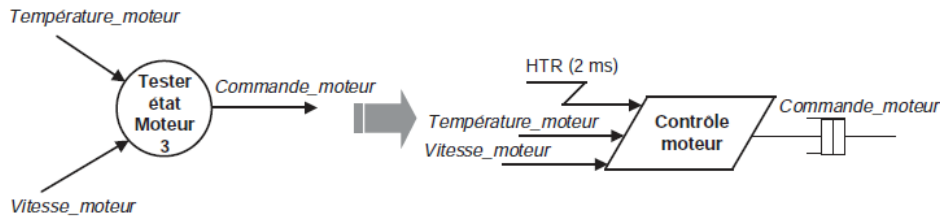


Figure 2-12 – Translation of functional data acquisition process into an periodic task [6]

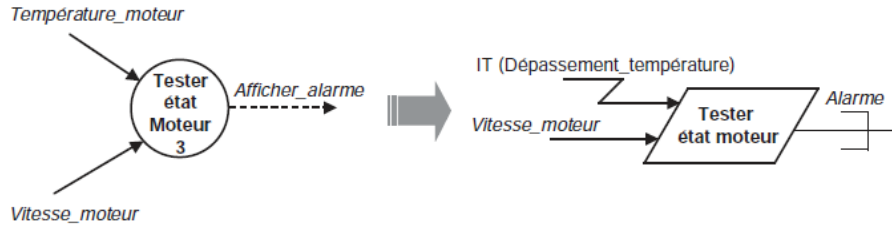


Figure 2-13 – Translation of functional data acquisition process into an aperiodic task [6]

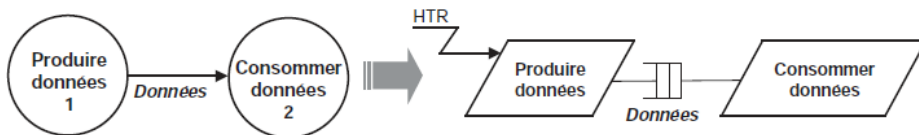


Figure 2-14 – Translation of a direct dataflow between two functional processes using mailbox [6]

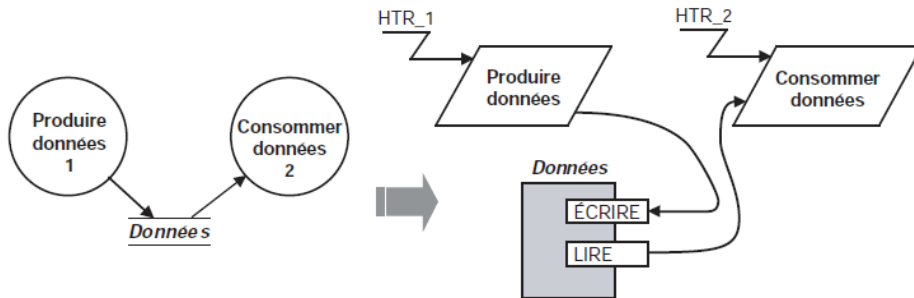


Figure 2-15 – Translation of a direct dataflow between two functional processes using memory module [6]

## Part 2

# CONTRIBUTIONS

---

This part consists of these main chapters:

- *Chapter 3* – Development Environment and Real Time Operating System 19
- *Chapter 4* – Device Preparation and Drivers ..... 25
- *Chapter 5* – Multitasking System Design..... 38
- *Chapter 6* – Communication Protocol ..... 51
- *Chapter 7* – Ground Station Design ..... 59
- *Chapter 8* – Simulink Code Generation ..... 66



# Development Environment and Real Time Operating System


*An integrated development environment (IDE) is a software that provides comprehensive facilities to programmers/developers for code development. An IDE normally consists of a code editor, a code compiler, and a debugger. Most modern IDEs also offer code completion features. Simply say, IDE will be the tool for us to program the FlyMaple controller board.*

*In this environment, the tasks for the FlyMaple will be developed. A real time operating system (RTOS) will run in the FlyMaple to schedule these tasks. This operating system must compatible with both the architecture of microprocessor on FlyMaple board, and with the development environment.*

*In order to select a suitable IDE as well as RTOS, the specification of our controller board should be considered.*

## 3.1 Integrated Development Environment

FlyMaple is a quadricopter controller board. It embeds a STM32F103RET6 (ARM Cortex-M3) as its main MCU. It integrates a 3-Axis accelerometer, a 3-Axis gyroscope, a 3-Axis compass, and a barometric pressure sensor.

FlyMaple is a product of . This manufacturer does not develop any IDE for his controller boards. The design of this FlyMaple is based on the Maple of LeafLabs, said the manufacturer. Since having the same architecture, the FlyMaple (DFRobot) can be treated as the Maple (LeafLabs) with the integrated I2C devices (the gyroscope, the accelerometer, the magnetometer and the barometer). They (the DFRobot) *recommend the **MapleIDE of LeafLabs** as the official toolchain for FlyMaple controller board.*

Our FlyMaple V1.1 uses STM32F103**RET6** ARM chip, so all the configurations on the MapleIDE toolchain, as well as all the references on LeafLabs' documentations, should be targeted to the *Maple RET6 Edition*.

Hereafter, unless there is a further complementation, all these phrases are used to mention the FlyMaple controller board: FlyMaple (board), Maple ((controller) board) and Maple RET6 ((controller) board).

### 3.1.1 MapleIDE Installation

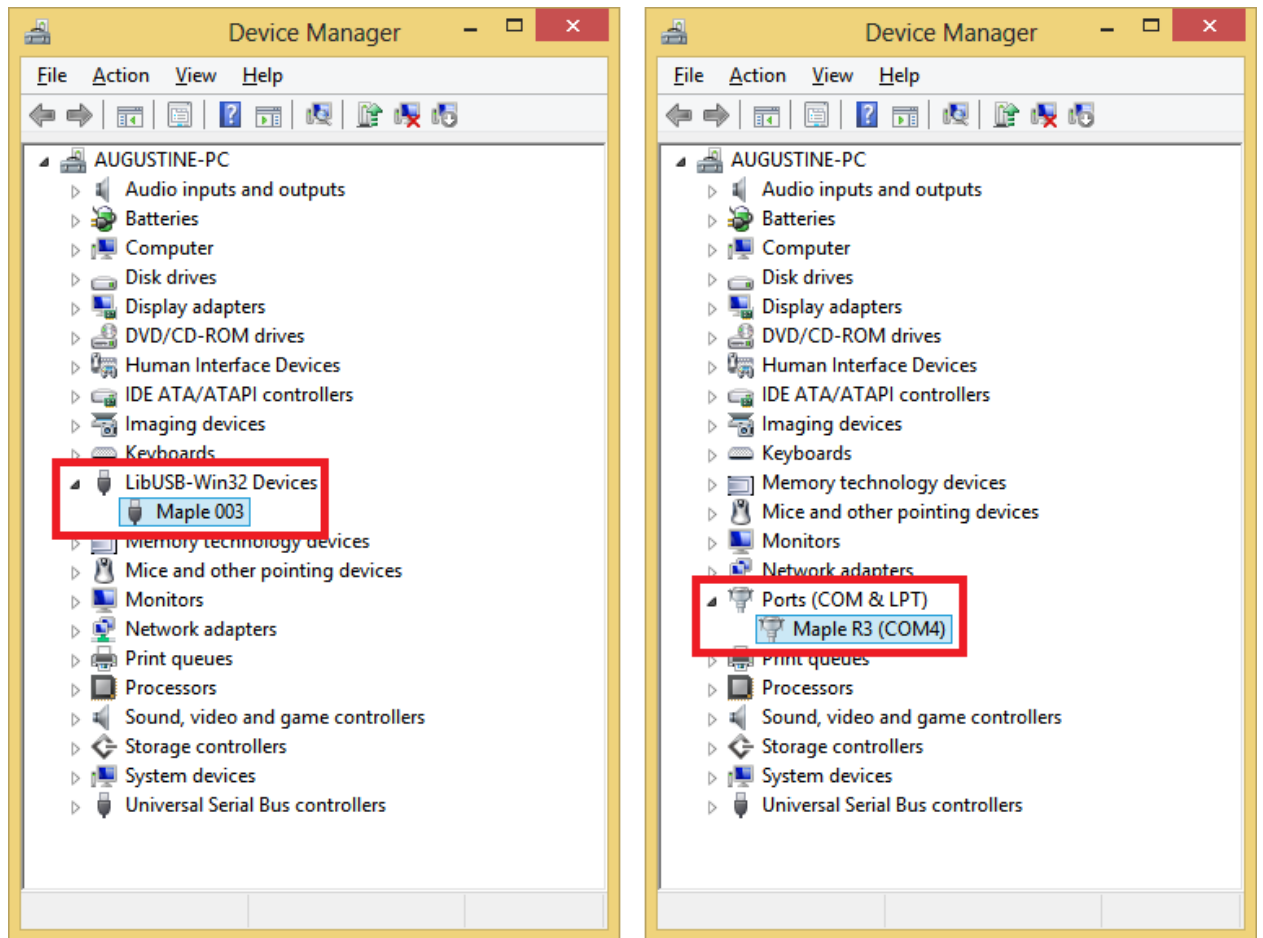
The Maple IDE toolchain is a package that bundles together a compiler, an upload utility for MCU, a software library, and a simple GUI code editor. All this software is free and open.

The Maple IDE can be freely downloaded from the LeafLabs' website. It has been officially tested for compatibility with most operating systems (Windows<sup>®</sup>, Mac<sup>®</sup> OS, Linux).

For Windows<sup>®</sup> OS, the IDE is distributed as a standalone program that can be run without any installation. However, drivers are required to use the FlyMaple:

- The DFU driver: for uploading code to the Maple,
- The serial driver: for communicating with the Maple using serial over USB.

*In Windows<sup>®</sup> 7 and later, we will not be able to install these drivers without disabling driver signing on your computer (please refer to the Technical Note #1 for more details). Currently, LeafLabs only officially supports the IDE on 32-bit Windows<sup>®</sup> XP<sup>™</sup>.*



**Figure 3-1 – Working modes of FlyMaple.**

Left: perpetual boot loader mode – DFU driver, right: normal mode – serial driver

The Maple has two operating mode: normal mode and perpetual boot loading mode. In normal mode, after being powered up, Maple boots and jumps to user code. In the other mode, Maple stays in DFU (*Device Firmware Upgrade*) state and does not jump to user code until the next reset. This is useful for guaranteeing that Maple will be always available for reprogramming, avoiding problems that can be caused by the user code (such as prevention of accessing the SerialUSB port). After successfully installing the drivers, both these two modes can be recognized, as shown in the above figure.

To most of the other controller boards that often require a “hardware programmer” for uploading code, this perpetual boot-loading mode is a good point of FlyMaple.

### 3.1.2 MapleIDE Testing

The MapleIDE will be tested to make sure that the code can be compiled and uploaded into the FlyMaple, as well as the communication with the FlyMaple via SerialUSB port can be established. First, following the MapleIDE documentation,



a simple example program that blinks the status LED has been loaded to the board.

Generally, for almost every controller board, there are two options for uploading code into the board: RAM and flash. Flash saves the program into permanent memory so the program will be run every time the board is powered up, while RAM simply injects the compiled program into the processor's memory.

Programming to RAM is faster to upload and a buggy program can be wiped away with a simple reset, while flash memory is larger and is the only option for permanently uploading a program.

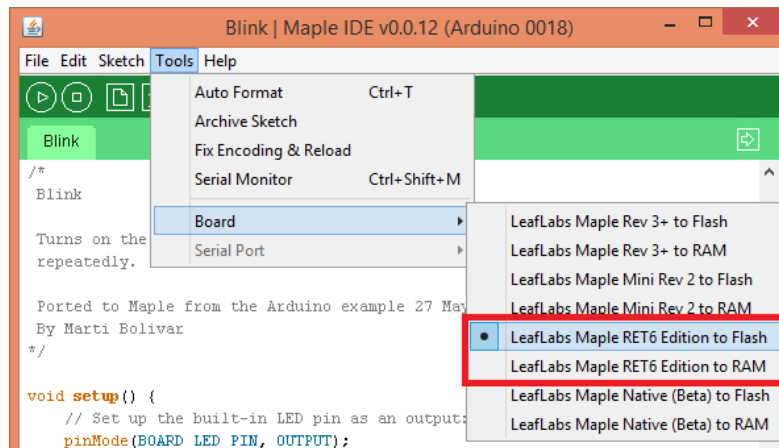


Figure 3-2 – Code uploading modes

MapleIDE is not an exception. We can easily choose which memory to upload the program via the “Tools” menu (Figure 3-2).

However, unfortunately, after many tests has been carried out, it can be said that the *RAM mode is not compatible with the FlyMaple*. Whenever trying to upload a program into RAM, the board becomes “invisible”: Windows<sup>®</sup> cannot detect the board, even after a hard reset (of the board). The only way to make the board “visible” again is putting the board in perpetual boot loader mode and uploading whatever program *into flash*, then reboot the board.

In short, a program can only be uploaded into FlyMaple board directly into flash memory, not RAM.

Then, other programs have also been uploaded to the FlyMaple board to test the communication via the SerialUSB port (using `SerialUSB.read()` and `SerialUSB.write()` methods).

## 3.2 Real Time Operating System

There are several real time operating systems that support the STM32 Cortex M3 architecture: FreeRTOS, Coocox CoOS, and ChibiOS. They are free and open RTOSes, support multitasking with many features such as semaphore, binary semaphore or mutex, mailbox or queue...

ChibiOS had been tested for LeafLabs Maple r5 and LeafLabs Maple Mini (both have *almost* the same architecture with DFRobot FlyMaple but use a MCU with smaller flash memory and smaller RAM). The instructions are officially provided on LeafLabs Wiki page, but still as a “quick and first draft” (see [wiki.leaf-labs.com/index.php?title=Running\\_ChibiOS\\_on\\_the\\_Maple\\_r5](http://wiki.leaf-labs.com/index.php?title=Running_ChibiOS_on_the_Maple_r5))

Since ChibiOS is not compatible with Maple boot loader, we must use Arduino IDE to program the controller board via an additional hardware programmer “Arduino Duemilanove.” The Maple board will be connected to this hardware programmer via the serial pins. Modifications on the Arduino IDE toolchain configuration files are required.

Coocox CoOS had been tested on MapleIDE by individual. The modified trimmed-down version for MapleIDE is shared on LeafLabs forum ([forums.leaflabs.com/topic.php?id=221](http://forums.leaflabs.com/topic.php?id=221)), but has never been “officially” supported by LeafLabs. This version is buggy and will never be finished since the author has dropped his project since 2010.

FreeRTOS is provided in the MapleIDE package as a built-in library, with the name “Maple FreeRTOS.” It becomes the official RTOS for MapleIDE. This is a lite version of FreeRTOS, with the omission of some features (for example, the queues are supported, but rewritable queues are not). Of course, main features of a RTOS (mutex, semaphore, queue...) remain.

Maple FreeRTOS is ready-to-use in MapleIDE: not any additional step is required.

**Table 3-1 – RTOSes that is compatible with FlyMaple**

	ChibiOS	Coocox CoOS	FreeRTOS
Compatible with MapleIDE	No	Yes*	<b>Yes</b>
Additional hardware required	Yes	<b>No</b>	<b>No</b>
Additional configurations/ toolchain modifications required	Yes	Yes	<b>No</b>
Officially recommended	No	No	<b>Yes</b>

*\*Tested by individual*

The table above shows that the FreeRTOS is the safest and most reliable OS to use on the MapleIDE for the FlyMaple board.



# Device Preparation and Drivers

*The quadricopter is equipped with the FlyMaple controller board, a GPS module\*, and a Wi-Fi module. A set of sensors is integrated on the FlyMaple, forms an inertial measurement unit.*

*Each device needs a driver to operate. Drivers communicate with the devices through the bus. When calling a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program.*

*This chapter will introduce the drivers for the integrated sensors (I2C devices) and the external modules (Wi-Fi, GPS).*

---

\* The extending modules for Arduino-based controller board are usually called *shields*. As defined on the Arduino's site, "*shields* are boards that can be plugged on top of the Arduino PCB extending its capabilities." In this report, both words "shield" and "module" are used as the same meaning.

## 4.1 System Overview

The figure below is the block diagram of the embedded system. It shows the connections between each hardware.

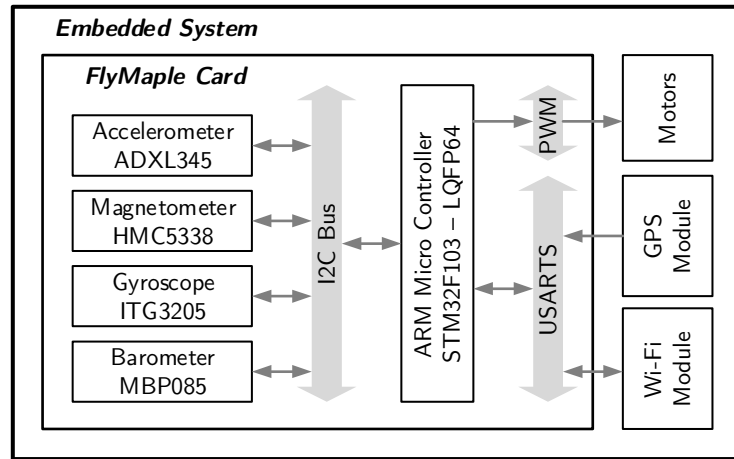


Figure 4-1 – Block diagram of embedded system

The embedded system consists of:

- A FlyMaple board with:
  - A microcontroller (ARM<sup>®</sup> Cortex<sup>™</sup>-M3 core with embedded flash and SRAM),
  - Integrated sensors built up a 10 degrees of freedom inertial measurement system:
    - a triple-axis gyroscope (ITG3200),
    - a triple-axis accelerator (ADXL345),
    - a triple-axis magnetometer (HMC5883L), and
    - a barometric pressure sensor (BMP085).
- A Wi-Fi shield (WizFi210 Wi-Fi chip) compatible with 802.11b/g/n standard, speed up to 11 Mbps (802.11b).
- A GPS shield (u-blox LEA-5H receiver) with an update rate up to 2 Hz, accuracy up to 2.5 m in position.

## 4.2 FlyMaple SDK and Available Resources

There is a development kit for the FlyMaple board called “FlyMaple SDK” that can be freely obtained via GitHub ([github.com/opendrone/flymaple-ide](https://github.com/opendrone/flymaple-ide)). This SDK is a package of drivers and several test procedures for the integrated I2C devices, but unfortunately, there is not any included documentation. In order to use this SDK, Doxygen, the de facto standard tool for generating documentation from annotated C++ sources, was used to generate the documentation. The SDK

was looked into codes, carefully inspected statement-by-statement and annotated following the syntax of Doxygen.

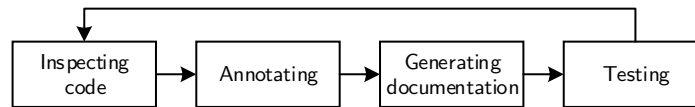


Figure 4-2 – Code annotating loop

During annotating the code, datasheet of the related peripherals are referred to check the agreement of the code and the information in the datasheets (for example, all the address of the peripherals; or sometimes, the computation from raw data...).

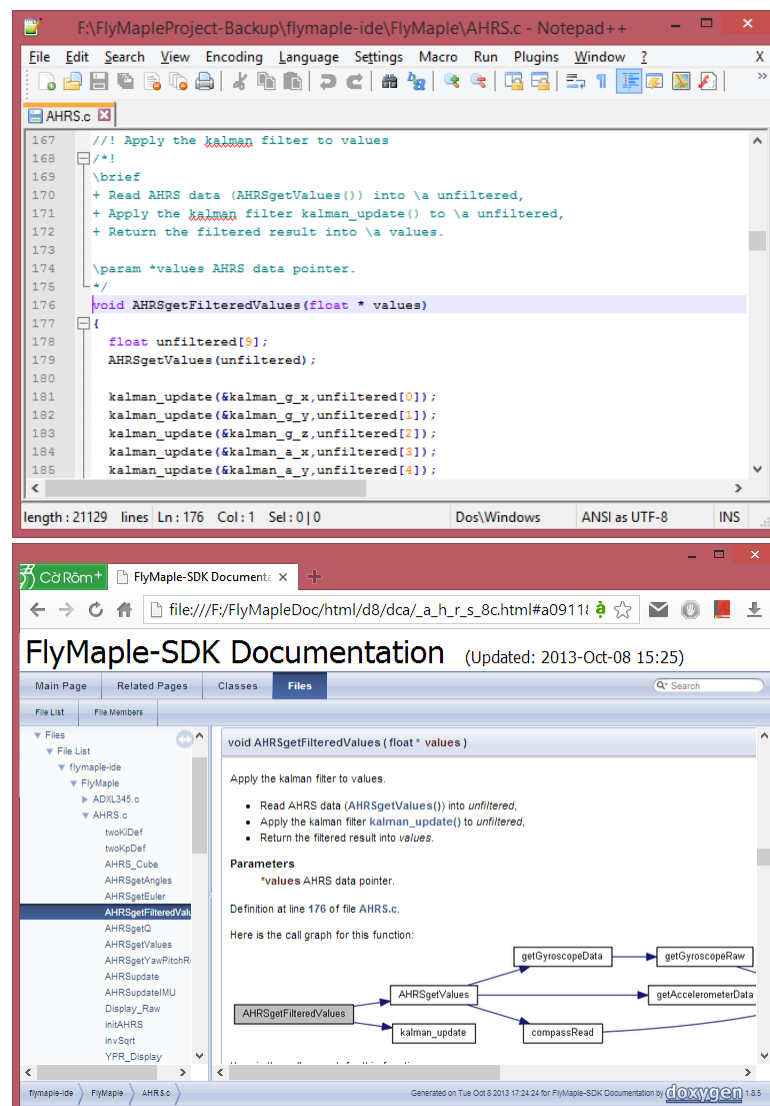


Figure 4-3 – Annotated code and corresponding generated documentation

After being generated, the correctness of the documentation will be double-checked by carrying out individual test for every method/function (or every set of methods/functions).

The drivers will be optimized later during the designing phase of the multi-tasking system.

The following table lists the resources (drivers, library) for the related peripherals that have already been developed for FlyMaple controller board.

**Table 4-1 – Available resources for FlyMaple controller board**

Library	Package	Description
i2c.h	MapleIDE built-in library	I2C peripheral support.
communication.pde	FlyMaple SDK	Communication library: <ul style="list-style-type: none"> <li>Processing I2C transactions (read, write),</li> <li>Encoding float value into a hexstring.</li> </ul>
ADXL345.pde	FlyMaple SDK	Driver for ADXL345 accelerometer.
BMP085.pde	FlyMaple SDK	Driver for BMP085 barometer.
HMC5883.pde	FlyMaple SDK	Driver for HMC5883 magnetometer.
ITG3205.pde	FlyMaple SDK	Driver for ITG3205 gyroscope.
MOTOR.pde	FlyMaple SDK	Driver for motors/PWM channels.
AHRS.pde	FlyMaple SDK	Implementation of Madgwick's data fusion algorithm Modules for converting quaternions into Euler angles

Next, each peripheral will be described in details.

## 4.3 Integrated Sensors: the I2C Devices

The FlyMaple's integrated sensors (accelerometer, magnetometer, gyroscope, and barometer) communicate with the MCU via the I2C-bus. They are connected to the I2C1 port of the MCU (pin D5 (SCL) and D9 (SDA) on the controller board).

### 4.3.1 I2C Communication

The I2C-bus was designed by Philips in the early 80s' to allow easy communication between components, which reside on the same circuit board. Philips Semiconductors migrated to NXP Semiconductors in 2006. The name I2C translates into "Inter IC." Sometimes the bus is called IIC or I<sup>2</sup>C bus.

## I2C-bus protocol

I2C-bus uses two wires, serial data (SDA) and serial clock (SCL), to carry information between the devices connected to the bus. Each device is recognized by a unique address (whether it is a microcontroller, LCD driver, memory or keyboard interface) and can operate as either a transmitter or receiver, depending on the function of the device. In addition to transmitters and receivers, devices can also be considered as masters or slaves when performing data transfers. A master is the device which initiates a data transfer on the bus and generates the clock signals to permit that transfer. At that time, any device addressed is considered a slave. [7]

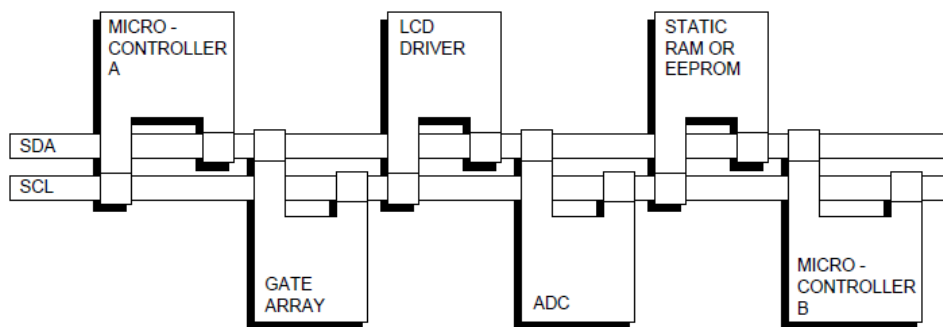


Figure 4-4 – I2C bus [7]

Let us consider the case of a data transfer between two microcontrollers connected to the same I2C-bus (Figure 4-4). This example highlights the master-slave and receiver-transmitter relationships found on the I2C-bus. Note that these relationships are not permanent, but only depend on the direction of data transfer at that time. The transfer of data would proceed as follows:

1. Suppose microcontroller A wants to send information to microcontroller B:
  - microcontroller A (master), addresses microcontroller B (slave)
  - microcontroller A (master-transmitter), sends data to microcontroller B (slave-receiver)
  - microcontroller A terminates the transfer.
2. If microcontroller A wants to receive information from microcontroller B:
  - microcontroller A (master) addresses microcontroller B (slave)
  - microcontroller A (master-receiver) receives data from microcontroller B (slave-transmitter)
  - microcontroller A terminates the transfer.

Even in this case, the master (microcontroller A) generates the timing and terminates the transfer.



### 4.3.2 The Integrated I2C Devices

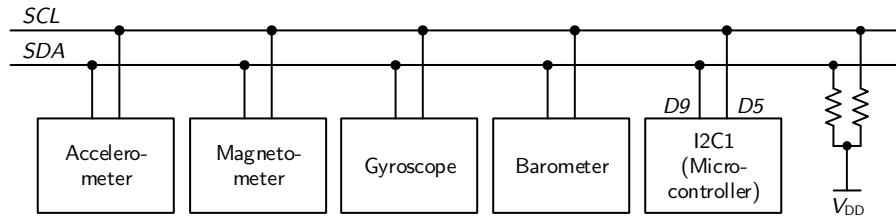


Figure 4-5 – I2C bus on the FlyMaple

The MCU (I2C1) on FlyMaple board will be initialized as bus master.

Before acquiring the measurements from the sensors, it must be initialized or calibrated. Then, the measurements then can easily be done by doing an I2C transaction between the MCU and the related sensor.

The related drivers, as listed in the Table 4-1, provide us the methods to complete these tasks. In addition, the AHRS module also provides a single method to invoke all the initializing or calibrating code for all the integrated I2C devices. With this AHRS module, we can also get the quaternions values at any time by calling only one single method. Once being invoked, this method will automatically call the related methods for each sensor to acquire all the measurements from all the I2C sensors, and then apply the data fusion algorithm (*see Chapter 2*) to return the quaternions.

### 4.3.3 Measurement Resolution

We can change the measurement resolution of the I2C sensors by changing the settings in the drivers. Notice that increasing the measurement will decrease the responding time of the sensor. *For more details, please refer to the Technical Report #4.*

## 4.4 GPS Shield

GPS shield supports both UART and I2C hardware interface. It uses NMEA data protocol. Only the official driver written for Arduino board (not FlyMaple board) for acquiring GPGGA data directly as parsed values via I2C interface is available (*for the NMEA protocol, see Part 4.4.2.1*). Note that we could not find any UART driver for this hardware.

### 4.4.1 Hardware Preparation

#### 4.4.1.1 I2C interface

The architecture of Arduino board is different from the one of FlyMaple board. On the Arduino Uno, the two data lines SDA and SCL of the I2C interface are mapped into the “*analog in*” pins A4 and A5. These positions, on the FlyMaple, are the location of the GPIO (general-purpose input/output) pins D19 and D20; the I2C pins are mapped into a separated header.

Nevertheless, why Arduino? Because this GPS shield is designed for Arduino architecture. When stacking the GPS shield into the FlyMaple, its I2C pins are connected into the GPIO pins of FlyMaple. As a result, we cannot communicate with the GPS shield using I2C interface. I2C pins cannot be remapped, so the only way to use the I2C interface is connecting the data lines of the GPS shield to the I2C header on FlyMaple (by using wires).

#### 4.4.1.2 UART interface

To acquire data from the GPS shield via UART interface, jumpers are needed on the board. The RX and TX pins are connected to the Serial2 port (pin D0 & D1) on FlyMaple board.

The Wi-Fi shield also uses this Serial2 port. Since UART interface is point-to-point, but connected to the MCU and every shields that means that only one shield can use this interface at once, and that we cannot use both Wi-Fi shield and GPS shield at the same time on the same port. So, one of these two shields must be hard-remapped (using wires) into another serial port (for example Serial1 port – pin D7 & D8).

With UART interface, only raw data can be read (as strings). We must write a specific library to acquire and parse these data into usable values.

#### 4.4.1.3 Summary

FlyMaple board uses JST connector for I2C interface. That means we need an additional JST connector to connect the I2C pins of the GPS shield to the I2C

header on the controller board (Figure 4-6). With the UART interface, we only need two wires to remap the pins as shown on the Figure 4-7 (actually, since data is only send one direction from the GPS shield to the controller board, only one wire is required).

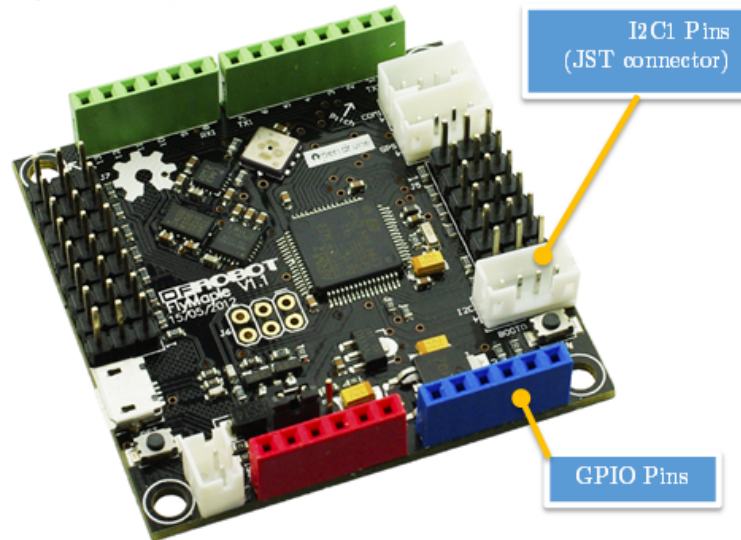


Figure 4-6 – I2C1 header on FlyMaple

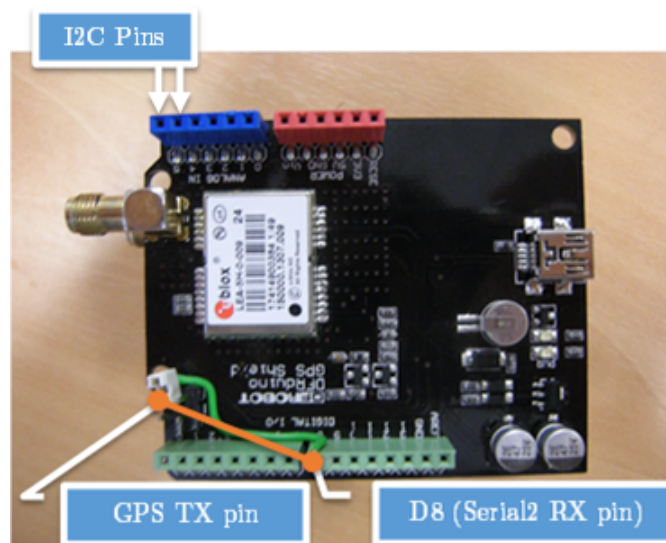


Figure 4-7 – Remapping GPS shield TX pin

The I2C interface is faster than UART. In I2C interface, data can be parsed directly from the register of GPS shield, while with the UART interface they must be transferred into serial buffer and then be read into RAM before being parsed. In the second case, data loss can occur due to buffer overflowing. In addition, there is an available driver for the I2C interface (of course, since the driver is written for Arduino board, some modifications are required for using on FlyMaple board).

However, the I2C pins of the GPS shield are connected to GPIO pins of FlyMaple controller board. On these lines, there is not any pulled-up resistor. Mistaken on configuring these pins not only causes malfunction but also can break the GPS shield and/or the controller board. While mistaken on re-mapping RX and TX pins can only cause data loss/confliction without any risk for the boards, this solution is safer than using I2C interface.

From the availabilities (available wires and jumpers) and for lowering the risk, we decided to use the UART interface. A GPS data acquiring & processing library has been developed.

## 4.4.2 UART Driver for GPS Shield

### 4.4.2.1 NMEA protocol

In normal operating mode, u-blox LEA-5H GPS shield returns data following NMEA protocol, as five types of sentence:

- \$GPRMC - Recommended minimum specific GPS/Transit data,
- \$GPVTG - Track made good and ground speed,
- \$GPGGA - Global Positioning System Fix Data,
- \$GPGSV - GPS Satellites in view,
- \$GPGLL - Geographic position, latitude / longitude.

```
$GPRMC,083723.00,A,5155.55442,N,00434.65873,E,0.211,,210810,,,A*7F<CR><LF>
$GPVTG,,T,,M,0.211,N,0.391,K,A*2A<CR><LF>
$GPGGA,083723.00,5155.55442,N,00434.65873,E,1,06,2.51,-4.0,M,46.0,M,,*70<CR><LF>
$GPGSA,A,3,27,15,09,28,18,08,,,,,,,,,3.33,2.51,2.20*04<CR><LF>
$GPGSV,3,1,12,05,04,187,,08,11,078,29,09,43,270,37,12,03,212,*75<CR><LF>
$GPGSV,3,2,12,15,78,214,34,17,25,106,18,18,32,294,39,22,13,324,20*77<CR><LF>
$GPGSV,3,3,12,24,04,342,,26,55,139,24,27,62,276,36,28,39,055,26*77<CR><LF>
$GPGLL,5155.55442,N,00434.65873,E,083723.00,A,A*6E<CR><LF>
```

Figure 4-8 – Sample data returned from LEA-5H GPS shield in normal operating mode

In order to get the longitude, latitude and attitude, only the \$GPGGA sentence is enough. The structure of \$GPGGA has 17 fields:

```
$GPGGA,hhmmss.ss,Latitude,N,Longitude,E,FS,NoSV,HDOP,msl,m,Altref,m,DiffAge,<↵>
DiffStation*cs<CR><LF>
```

Table 4-2 – Structure of GPGGA sentences

Field no.	Example	Format	Name	Unit	Description
0	\$GPGGA	string	\$GPGGA	-	Message ID, GGA protocol header
1	092725.00	hhmmss.ss	hhmmss.ss	-	UTC time at current position
2	4717.11399	ddmm.mmmm	Latitude	-	Latitude, degrees + minutes

Field no.	Example	Format	Name	Unit	Description
3	N	character	N	-	N/S indicator, N = North or S = South
4	00833.91590	ddmm.mmmm	Longitude	-	Longitude, degrees + minutes
5	E	character	E	-	E/W indicator, E = East or W = West
6	1	digit	FS	-	Position fix status indicator
7	8	numeric	NoSV	-	Satellites used, range from 0 to 12
8	1.01	numeric	HDOP	-	Horizontal dilution of precision
9	499.6	numeric	msl	m	Mean sea level altitude
10	M	character	uMsl	-	Units, Meters (fixed field)
11	48.0	numeric	Altref	m	Geoid Separation
12	M	character	uSep	-	Units, Meters (fixed field)
13		numeric	DiffAge	s	Age of differential corrections, blank (null) fields when DGPS is not used
14	0	numeric	DiffStation	-	Differential reference station ID
15	*5B	hexadecimal	cs	-	Checksum
16		characters	<CR><LF>	-	Carriage return and Line feed

#### 4.4.2.2 Code description

##### GPGGA data record

```
typedef struct {
    float time; // HHMMSS.SSS
    float lat;
    float lon;
    float alt;
    char quality;
    byte sat_num;
} GPGGA;
```

##### Method description

```
unsigned GetGPSRaw (char *rawGPS, char ID[])
```

Waiting for data stream until header matches with message ID **ID** can be found, and then acquire the complete sentence (that ended with <CR> character). The sentence will be put into the **rawGPS** string and ended with '\0' character.

**Input:** string **ID**: message ID to get, ended with a comma. For example "\$GPGGA,"

**Output:** **rawGPS**.

Return value: 1 when done.

Warning: this method can cause blocking.

##### Example:

Assume we have the data stream on Serial1 as shown in the Figure 4-8, the procedure **GetGPSRaw(buff, "\$GPGGA,")** will read and skip every coming byte

until the matching header is found (the red characters in Figure 4-8). Then, the next bytes (the blue characters in Figure 4-8) will be read into **buff** until a <CR> character is found. Now, **buff** is:

083723.00,5155.55442,N,00434.65873,E,1,06,2.51,-4.0,M,46.0,M,,\*70<\0>

The procedure exit at the <CR> character. The rest of the data stream will be left untouched.

**void indexing(const char message[], unsigned int \* index)**

**Input:** a **message** string contains words that separated by comma ‘,’.

**Output:** an **index** array that shows the position of each word in the message.

**Example:**

message = “084746.00,4639.64861,N,00021.71397,E,1,04,2.74,97.6,M,47.0,M,,\*66”

Word	Length	i	index[i]
084746.00,	9+1	0	0
4639.64861,	10+1	1	10
N,	1+1	2	21
00021.71397,	11+1	3	23
E,	1+1	4	35
1,	1+1	5	37
04,	2+1	6	39
2.74,	4+1	7	42
97.6,	4+1	8	47
M,	1+1	9	52
47.0,	4+1	10	54
M,	1+1	11	59
<null>,	0+1	12	61
*66	3	13	62

**GPGGA parse\_GPGGA(const char message[])**

**Input:** GPGGA sentence (without “\$GPGGA” header)

**Output:** parsed values (as a GPGGA structure)

**Table 4-3 – GPGGA parsing method**

Name	Raw form (strings)	Parsed value	Type	GPGGA Field
Time	“hhmmss.ss”	hhmmss.ss	Float	time
Longitude, N	“ddmm.mmmmm”, “N”	$\text{sign}(N) * (\overline{dd} + \overline{mm.mmmmm}/60)$	Float	lat
Latitude, E	“dddmm.mmmmm”, “E”	$\text{sign}(E) * (\overline{ddd} + \overline{mm.mmmmm}/60)$	Float	lon
msl, AltRef	“aa.a”, “gg.g”	$\overline{aa.a} + \overline{gg.g}$	Float	alt
Position fix status indicator	“c”	‘c’	Character	quality

Name	Raw form (strings)	Parsed value	Type	GPGBA Field
Satellites used	"nn"	nn	Byte	sat_num

N	sign(N)	E	sign(E)
'N'	+1	'E'	+1
'S'	-1	'W'	-1

### Example:

message = "084746.00,4639.64861,N,00021.71397,E,1,04,2.74,97.6,M,47.0,M,,\*66"

Raw Value (strings)	Field	Parsed Value	Type	Explanation
"084746.00"	GPGBA.time	84746.0	Float	string to float
"4639.64861", "N"	GPGBA.lat	46.66081	Float	Sign('N') $\times$ (46 + 39.64861/60)
"00021.71397", "E"	GPGBA.lon	0.01611866	Float	Sign('E') $\times$ (0 + 21.71397/60)
"97.6", "47.0"	GPGBA.alt	144.6	Float	97.6 + 47.0
"1"	GPGBA.quality	'1'	Char	
"04"	GPGBA.sat_num	4	Byte	string to byte

## 4.5 Wi-Fi Shield

The Wi-Fi shield uses UART interface for communicating with FlyMaple board. If the Wi-Fi shield is configured to connect to a Wi-Fi network, data sent from FlyMaple board to the serial port will be sent automatically to the network, and data received from the network by the Wi-Fi shield will be sent automatically to the FlyMaple board.

Sending and receiving data to/from the serial port can be done using `Serial.write()` and `Serial.read()` methods. The sender that prepares data to be sent and receiver that parses the received data will be developed together with the transfer protocol (Chapter 6). No additional driver is required.

### 4.5.1 Hardware Preparation

When being stacked into the controller board, the Wi-Fi shield is connected to the Serial2 port. No further hardware configuration is required. Be sure that the serial pins of the GPS shield are remapped as discussed in Part 4.4.1.

## 4.5.2 Network Configuration

In order to connect to a network, the Wi-Fi shield must be configured via the UART interface using AT commands. Following the documentation, this can be done by switching the Wi-Fi shield into Arduino mode. However, once again, we encountered with hardware compatibility. This Wi-Fi shield is designed for Arduino. Let see how the “USB/Arduino” actually works.

There is a “USB/Arduino” mode switch on the shield. In the Arduino mode, Wi-Fi shield and controller board communicate to each other via UART interface: TX pin of Wi-Fi shield is connected to RX pin of the controller board, and RX pin to TX pin. In the USB mode, the TX pin of Wi-Fi shield is connected to TX pin of the controller board, and RX pin to RX pin. With this connection, Arduino controller board will act like a USB/serial adapter. We can configure the Wi-Fi shield via SerialUSB port directly.

Unfortunately, this USB mode is unusable in our case. “Unlike the Arduino, none of the Maple’s serial ports is connected to the USB port on the Maple board [8].” Since then, a soft-bridge (a program) between serial port and USB port on FlyMaple board must be established, in order to forward data from SerialUSB port (which is connected with the PC) to Serial2 port (which is connected with Wi-Fi shield), and vice versa.

Whenever the bridge program runs on FlyMaple board, AT commands can then be send from PC to the Wi-Fi shield via SerialUSB port. The Wi-Fi shield must be configured according to the router information. The following table shows some sample AT commands.

**Table 4-4 – AT commands**

Command	Function
at+wd	Disassociate from previous connection
at+wwpa=<ap_password>	Set the password.
at+ndhcp=1	Enable DHCP settings.
at+ndhcp=0	Disable DHCP. IP must be set manually. Please refer to the WizFi datasheet.
at+wa=<yourssid>	Define router’s “ssid”
at+nstat=?	Show the current wireless and network config.
at+wstatus	Adapter reports the current network config to serial host
at+dnslookup=google.com	Test your connection to the internet. If successful it will return Google’s IP address
at+nstcp=4000	Set TCP server at port 4000



atc1	Set to autoconnect at restart
at&w0	Save settings to profile "0"
ata	Connect

---

\* \* \*

In short, in order to run the AT commands for Wi-Fi shield on FlyMaple board, these steps should be followed (*Once again, note that the official documentation for this Wi-Fi shield is targeted to the Arduino controller board, so the hardware settings written on that documentation are different from these below. Please do not confused!*):

1. Upload the bridge program into the FlyMaple board.
2. Stack the Wi-Fi shield to the FlyMaple board.
3. Set the "USB/Arduino" switch to "Arduino" side to enable UART communication between the FlyMaple and the Wi-Fi shield.
4. Set the "Run/Prog" to "Run" side.
5. Connect the FlyMaple board to PC.
6. Send AT commands from the PC via SerialUSB comm port.

*Please refer to the Technical Note #2 for more details about the bridge program and the AT commands.*

# Multitasking System Design

## 5.1 System Specification

The Figure 5-1 is the block diagram of the complete system. The sensors (accelerometer, magnetometer, and gyroscope, barometer, and GPS module) carry out the measurements. The micro-controller acquires these data; exchanges data with the Ground Station via Wi-Fi, and then calculate the PWM signals to control the actuators (the motors). Thus, we have the context diagram as shown in the Figure 5-2.

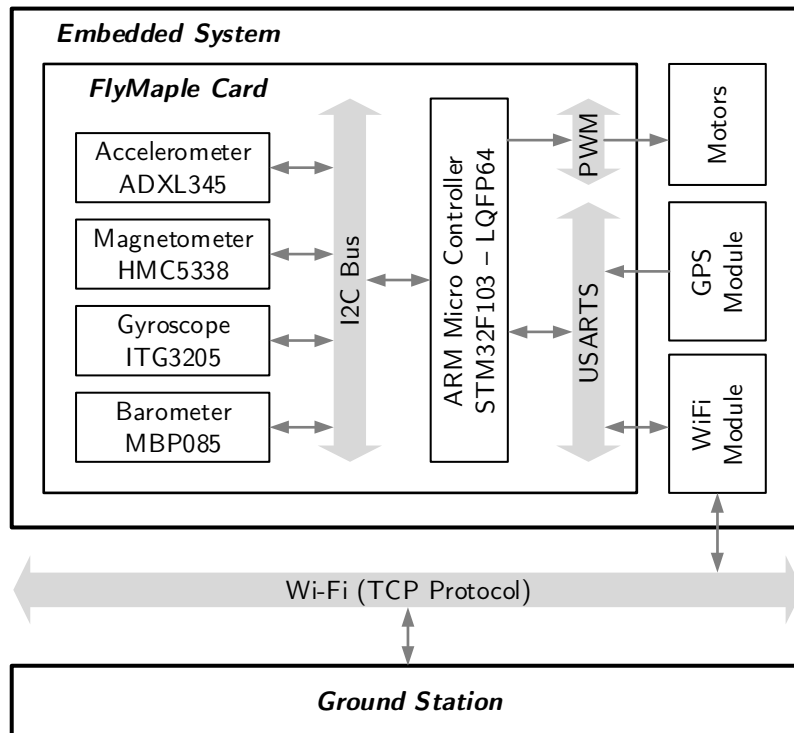


Figure 5-1 – Complete block diagram

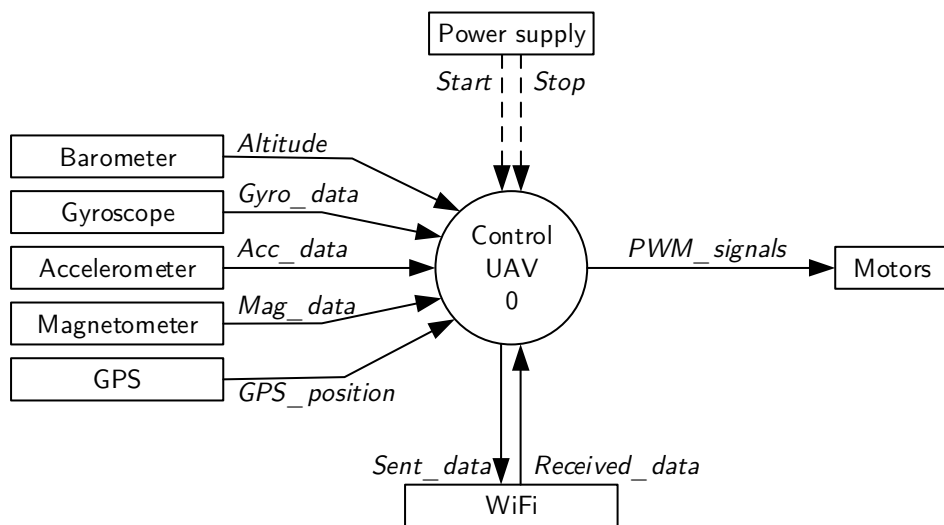


Figure 5-2 – Quadricopter context diagram

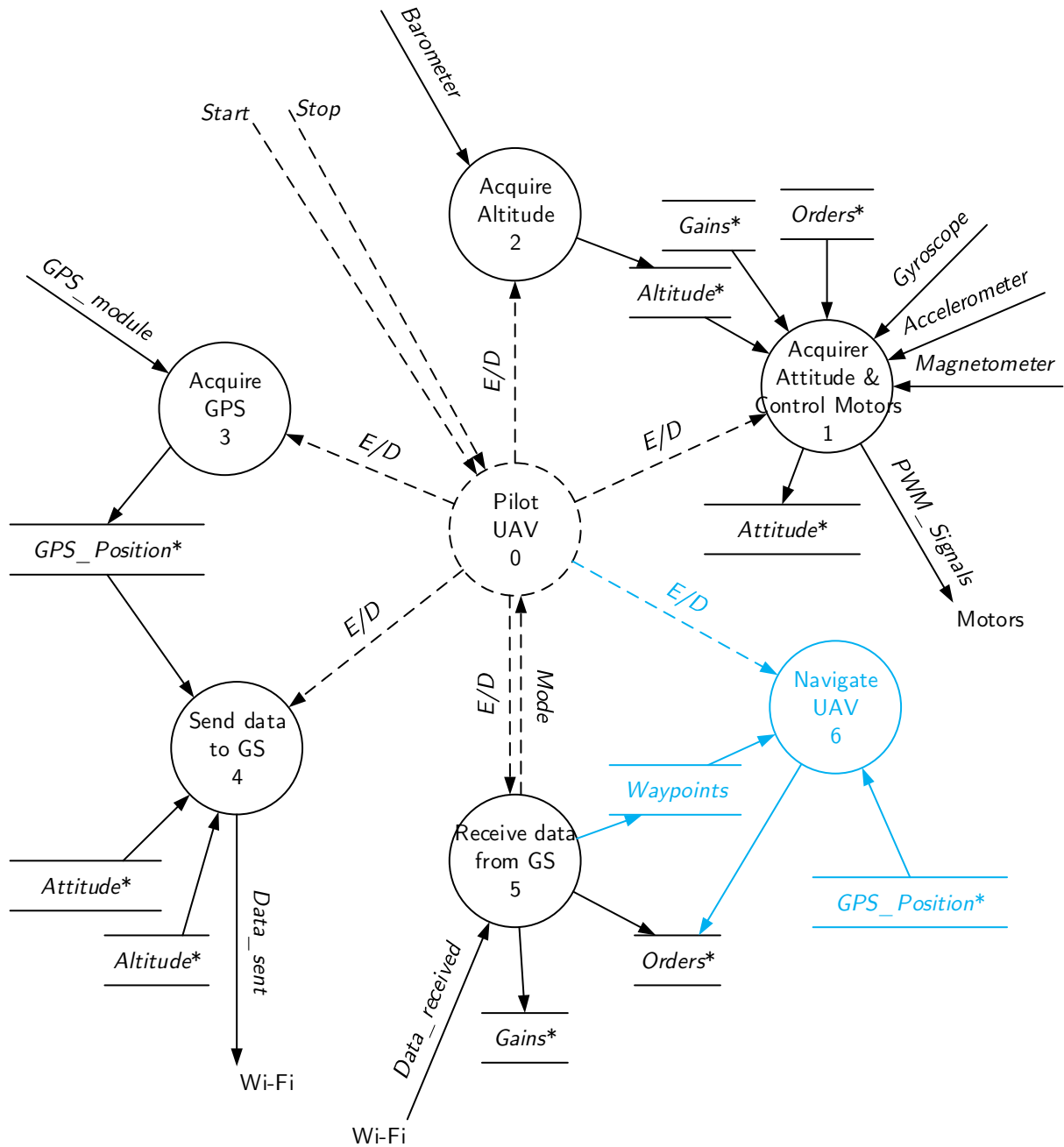


Figure 5-3 – Preliminary diagram of Quadricopter control system

The Quadricopter will be designed to operate in two mode: assisted mode and auto mode. In the auto mode, the Quadricopter generates the orders (reference roll, pitch, yaw angle and reference altitude) itself thanks to the flight path generator. In assisted mode, the orders will be sent directly from the Ground Station.

The preliminary diagram can now be derived (Figure 5-3).

In the **Process 1**, the measurements from sensors (gyroscope, accelerometer, and magnetometer) is firstly be read and fused into quaternion values using data

fusion algorithm (*See Part 2.1*). The attitude of the Quadricopter will then be acquired from quaternion values, in form of Euler's angles—roll, pitch, and yaw. The flight control system will then be applied for the current attitude, altitude, orders and gains. The output of the flight control system will then be converted into PWM values and sent to the actuators (motors).

**Process 2** acquires the measurements from barometer sensor and calculate the correspondent altitude.

**Process 3** acquires data from the GPS module and parses them into usable values (UTC time; current position—longitude, latitude, altitude; ...). These values will be put into GPS\_position record.

**Process 4** sends the current state of the Quadricopter (altitude, attitude, position) down to the Ground Station via Wi-Fi connection.

**Process 5** receives data (orders, gains, mode, and waypoints) from Ground Station.

In auto mode, **Process 6**—flight path generator—will calculate the orders from the current GPS position of the Quadricopter and the waypoints. In this mode, Ground Station does *not* send the orders. In assisted mode, this process is disabled.

In the Figure 5-4, the process *Send Data to GS* and *Receive Data from GS* are combined into one process *Exchange Data with GS*. The process *Navigate UAV* is composed as a part of the main process *Acquire Attitude & Control Motors*. The main process (Process 1) is decomposed in Figure 5-5.

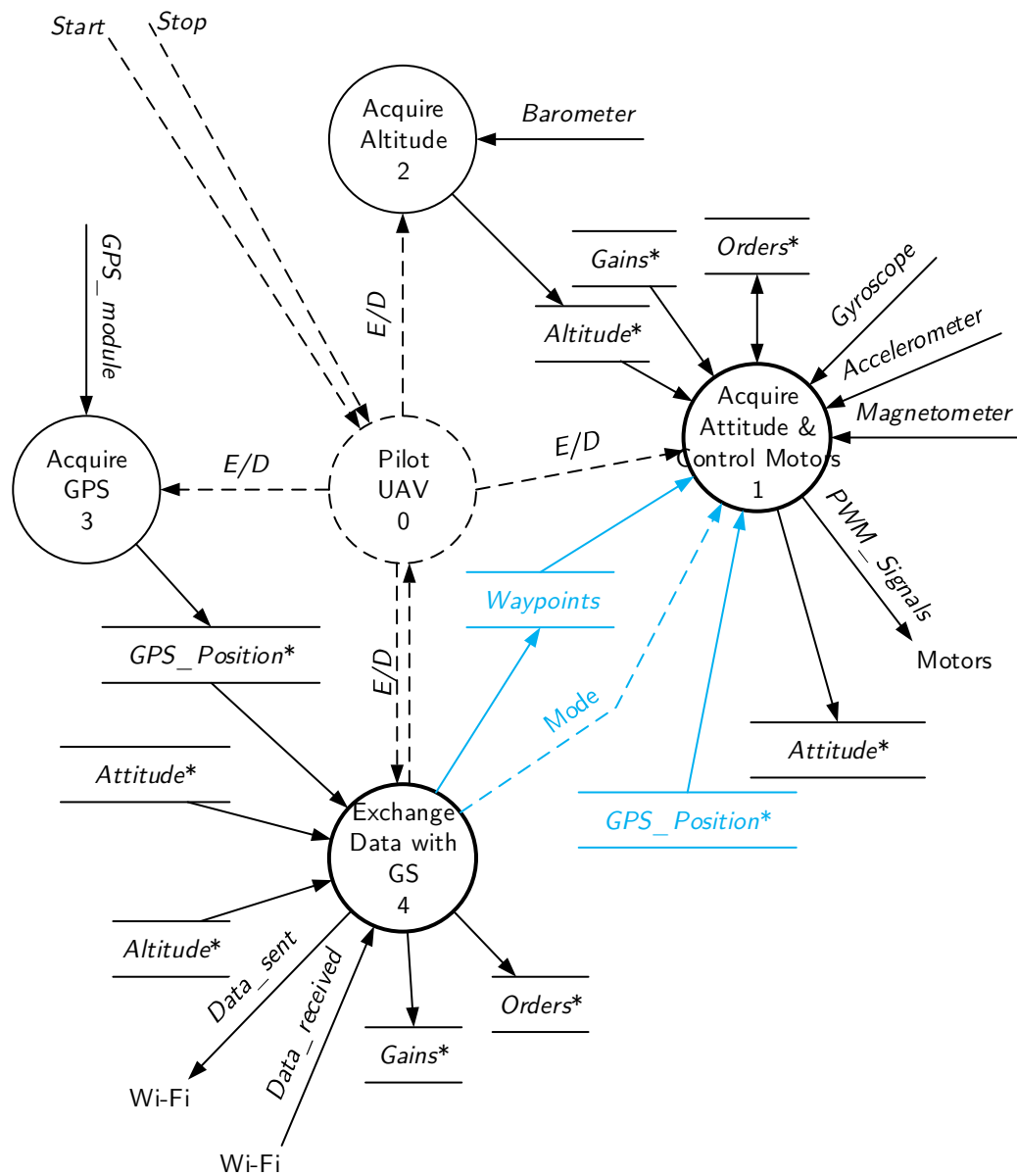


Figure 5-4 – Simplified preliminary diagram of Quadricopter control system

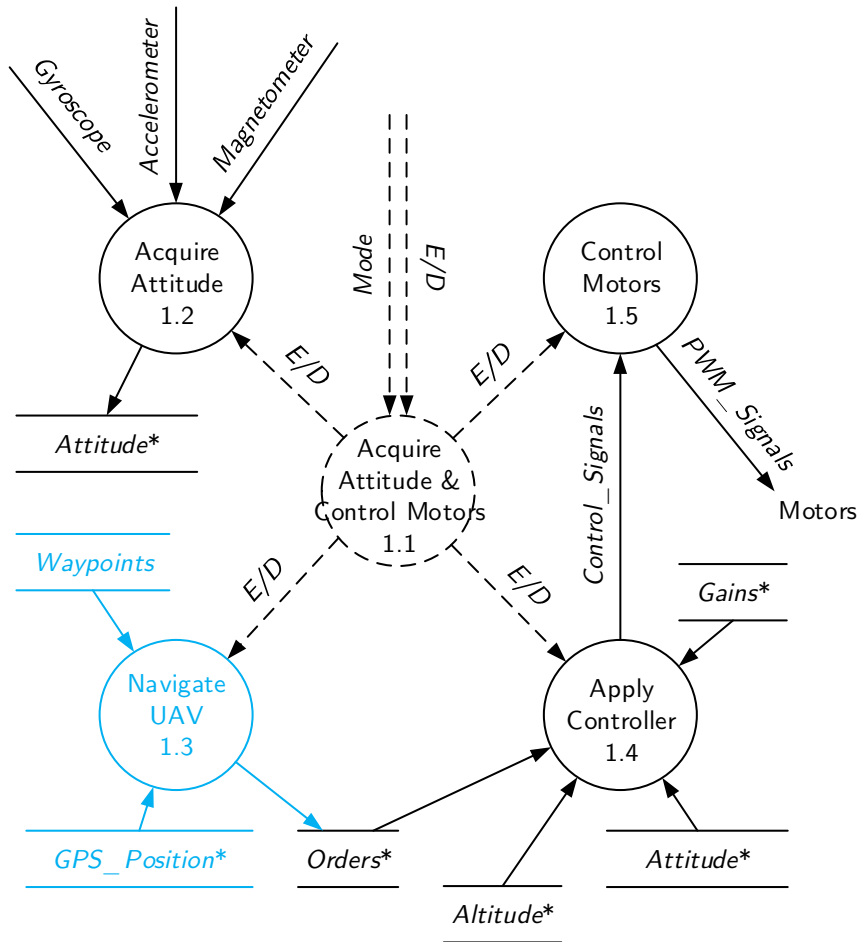


Figure 5-5 – Decomposition of Process 1

When Process 1 run, it firstly reads the sensor measurements and calculate the current attitude (primitive functional process 1.2). Next, it checks if the mode is auto, the navigator (primitive functional process 1.3) will be enabled to calculate the orders. If not, the navigator will be disabled. Then the controller will be applied (primitive functional process 1.4) and PWM signals will be calculated and sent to the motors (primitive functional process 1.5).

Now, a multitask design of the system can be carried out. It is presented using a DARTS (Design Approach for Real-Time Systems) diagram.

## 5.2 System Design

Each process in the SART diagram will be passed into a task, the data flow and event (mode) flow will be implemented as the memory modules.

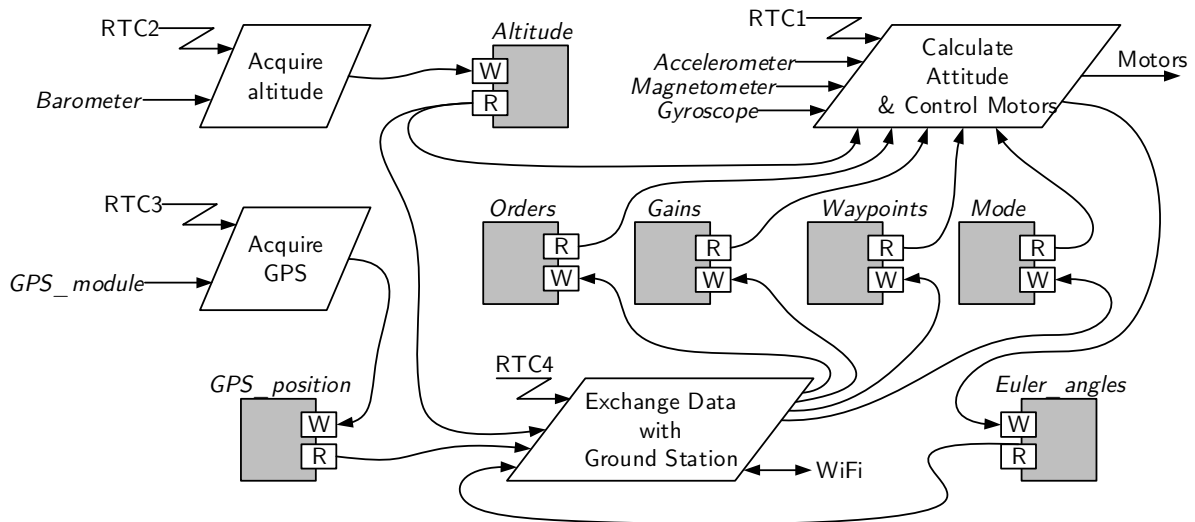


Figure 5-6 – Quadricopter DARTS diagram

**Task 1** calculates the Euler's angles, generates the PWM signals, and send PWM signals to the motors:

- **Task 1.2** (Attitude Acquirer) acquires data from gyroscope, accelerometer and magnetometer (read data, apply Kalman filter...) and fuses them into form of Euler's Angles (attitude).
- If mode is "auto," **Task 1.3** (Navigator) calculates the orders from waypoints and GPS position.
- **Task 1.3** (Controller) is the implementation of the flight control system, designed in MATLAB/Simulink.
- **Task 1.4** converts the control signals (output of 1.3) into PWM signals and send to the motors.

**Task 2** acquires the altitude from the barometer.

**Task 3** acquires the GPS information from the GPS module.

**Task 4** communicates with the ground station: sending the current attitude, altitude, GPS information, MCU time...; acquiring the orders, waypoints, gains (for the PID controller)... to and from the ground station.

\* \* \*

All the tasks above are scheduled as periodic tasks.

The memory modules and (critical) resources are listed in the following table.



Table 5-1 – List of resources

Resource	Data Type	Sharing Task(s)	Access Mode
Serial/Serial1	ASCII characters	3	Read/Write
Serial/Serial2	ASCII characters	4	Read/Write
<b>SerialUSB</b>	ASCII characters	1, 2, 3, 4	Write ( <i>Debug mode only</i> )
I2C/accelerometer	Raw bytes	1	Read/Write
I2C/barometer	Raw bytes	2	Read/Write
I2C/gyroscope	Raw bytes	1	Read/Write
I2C/magnetometer	Raw bytes	1	Read/Write
PWM channels	PWM signals	1	Write
<b>MDD_altitude</b>	Float	2 1, 4	Write Read
<b>MDD_eulerAngles</b>	Float×3	1 4	Write Read
<b>MDD_gains</b>	Float×9	1 4	Read Write
<b>MDD_GPSposition</b>	( <i>GPGL Record*</i> )	(1) 3 4	Read ( <i>auto mode only</i> ) Write Read
<b>MDD_orders</b>	Float×4	(1) 4	Read ( <i>assisted mode</i> ) Write ( <i>auto mode</i> ) Write ( <i>assisted mode</i> )
<b>MDD_mode</b>	Byte (0: assisted, 1: auto)	1 4	Read Write
<b>MDD_waypoints</b>	( <i>Record</i> )	(1) 4	Read ( <i>auto mode only</i> ) Write

\* see Part 4.4.2.2

## 5.3 System Implementation

### Critical resources

Critical resources (or shared resources) are the resources that are shared between two or more tasks. In FreeRTOS for Maple IDE, we can use mutexes to guard access to the critical resources. When a task wishes to access the resource it must first obtain (*take*) the token. When it has finished with the resource, it must *give* the token back – allowing other tasks the opportunity to access the same resource.

A memory module can be defined by a mutex and a global variable (Figure 5-7). Figure 5-8 shows the steps to update the value of a memory module.

```

/*
 * Memory modules (MDDs)
 */
float MDD_altitude = 0.0;
xSemaphoreHandle xSem_altitude = NULL;
// ...

void setup ()
{
    // Peripheral initialization
    // ...
    // End of initialization code
    xSem_altitude = xSemaphoreCreateMutex();
    // ...
    // Tasks scheduling
    // ...
}

```

Figure 5-7 – Memory module definition and initialization

```

void vTaskAcquireAltitude(void *pvParameters)
{
    // ...
    altitude = acquireAltitude(); // acquire current altitude from barometer
    if ( xSemaphoreTake( xSem_altitude, portMAX_DELAY) == pdTRUE )
        // request access to the memory module
    {
        MDD_altitude = altitude; // update memory module
        xSemaphoreGive(xSem_altitude); // release memory module
    }
    // ...
}

```

Figure 5-8 – Update a memory module

## The complete implementation

Following the design in the previous parts, a complete implementation of the multitasking system is carried out as following.

```

/*****
 * Quadricopter
 * MAIN PROGRAM : Task scheduler
 * =====
 * BUI Nha-Dat @ HCMUT/ENSMA
 * Quadricopter project
 * Last modified : 2014-Jan-15
 * 12:00 AM
 *****/

#include <MapleFreeRTOS.h>
// define other includes, global variables, definitions...

/*
 * Real time clocks
 */
const portTickType RTC1 = 100/portTICK_RATE_MS; // main_task
const portTickType RTC2 = 100/portTICK_RATE_MS; // altitude
const portTickType RTC3_short = 50/portTICK_RATE_MS; // GPS
const portTickType RTC3_long = 1000/portTICK_RATE_MS; // GPS
const portTickType RTC4 = 100/portTICK_RATE_MS; // communicator

```

```

portTickType xLastWakeTime_Task1; // main task
portTickType xLastWakeTime_Task2; // acquiring altitude task
portTickType xLastWakeTime_Task3; // acquiring GPS task
portTickType xLastWakeTime_Task4; // communicator

/*
 * Memory modules (MDDs)
 */
float MDD_altitude = 0.0;
GPGBGA MDD_GPSposition = initGPGBGA();
float MDD_orders[4] = {0.0, 0.0, 0.0, 2.0};
float MDD_gains[9] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
float MDD_eulerAngles[3] = {0.0, 0.0, 0.0};
byte MDD_mode = ASSISTED;
xSemaphoreHandle xSem_altitude = NULL;
xSemaphoreHandle xSem_GPSposition = NULL;
xSemaphoreHandle xSem_orders = NULL;
xSemaphoreHandle xSem_gains = NULL;
xSemaphoreHandle xSem_eulerAngles = NULL;
xSemaphoreHandle xSem_mode = NULL;
xSemaphoreHandle xSem_USB = NULL;

/*
 * Task definitions
 */
// TASK 1 //////////////////////////////////////
void vTaskCalculateQuaternionsAndControlMotors(void *pvParameters)
{
    // Define & init local variables: altitude, gains, orders, eulerAngles,
    //                               controlSignal, pwmSignal
    // ...
    xLastWakeTime_Task1 = xTaskGetTickCount();
    while (1)
    {
        AHRSgetQ(q); // Reading sensors and calculate quaternions
        Q2E(q, eulerAngles); // Convert quaternions into Euler angles

        // Reading memory module: MDD_altitude -> altitude
        // Reading memory module: MDD_gains -> gains
        // Reading memory module: MDD_orders -> orders

        /* PID Flight Controller Integration
         * -----
         * Input : eulerAngles, altitude, orders, gains
         * Output : controlSignal
         * ...
         * End of integration
         */

        calculateMotorData(controlSignal, pwmSignal); // Calculating PWM signals
        controlMotors(motorData); // Send PWM signals to motors

        // Writing memory module: MDD_eulerAngles <- eulerAngles

        vTaskDelayUntil( &xLastWakeTime_Task1, RTC1 ); // Wait for the next period
    }
}

// TASK 2 //////////////////////////////////////

```

```

void vTaskAcquireAltitude(void *pvParameters)
{
    // Define & init local variable(s): altitude
    // ...
    xLastWakeTime_Task2 = xTaskGetTickCount();
    while (1)
    {
        altitude = ...; // Acquire altitude from barometer
        // Writing memory module: MDD_altitude <- altitude
        vTaskDelayUntil( &xLastWakeTime_Task2, RTC2 ); // Wait for the next period
    }
}

// TASK 3 ////////////////////////////////////////
void vTaskAcquireGPS(void *pvParameters)
{
    // Define & init local variables: GPSposition
    // ...

    /* Synchronizing with the period of GPS signal */
    // Skip the current period
    Serial1.flush();
    // Wait for the first byte of the next period come
    while (!Serial1.available()) {};
    /* Period started --- begin schedulation */

    xLastWakeTime_Task3 = xTaskGetTickCount();
    while (1)
    {
        if (Serial1.available())
        {
            // Acquire and process GPS data -> GPSposition
            // Writing memory module: MDD_GPSposition <- GPSposition

            // Wait for the next period.
            vTaskDelayUntil( &xLastWakeTime_Task3, RTC3_long );

        }
        else
        {
            /* Do nothing, wait for the next period. */
            vTaskDelayUntil( &xLastWakeTime_Task3, RTC3_short );
        }
    }
}

void vTaskCommunicator(void *pvParameters)
{
    // Define & init local variables: MCUTime,
    // ...
    xLastWakeTime_Task4 = xTaskGetTickCount();
    while (1)
    {
        /* Sender do work:
        * Sending: MCUTime
        * Reading & sending memory modules:
        *   MDD_eulerAngles, MDD_altitude, MDD_GPSposition
        */
    }
}

```

```
/* Receiver do work:
 * Receiving and processing data
 * Updating related memory modules:
 *   MDD_order, MDD_gains, MDD_mode
 */

// Wait for the next period.
vTaskDelayUntil( &xLastWakeTime_Task4, RTC4 );
}
}

void setup ()
{
    // Init code for peripherals (I2C devices, Serial interfaces...)
    // ...

    // Creating mutexes
    xSem_altitude   = xSemaphoreCreateMutex();
    xSem_GPSposition = xSemaphoreCreateMutex();
    xSem_orders      = xSemaphoreCreateMutex();
    xSem_gains       = xSemaphoreCreateMutex();
    xSem_eulerAngles = xSemaphoreCreateMutex();
    xSem_mode        = xSemaphoreCreateMutex();
    xSem_USB         = xSemaphoreCreateMutex();

    // Creating tasks
    xTaskCreate(vTaskCalculateQuaternionsAndControlMotors,
        (signed char *) "main_task",
        configMINIMAL_STACK_SIZE+256, NULL, tskIDLE_PRIORITY, NULL);
    xTaskCreate(vTaskAcquireAltitude, (signed char *) "acquireAltitude",
        configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY+1, NULL);
    xTaskCreate(vTaskAcquireGPS, (signed char *) "acquireGPS",
        configMINIMAL_STACK_SIZE+128, NULL, tskIDLE_PRIORITY, NULL);
    xTaskCreate(vTaskCommunicator, (signed char *) "communicator",
        configMINIMAL_STACK_SIZE+256, NULL, tskIDLE_PRIORITY+2, NULL);
    // Start scheduling
    vTaskStartScheduler();
    // End of code.
}
void loop() {} // This loop will never be reached!
```

# Communication Protocol

*C*ommunication protocol helps the Quadricopter and the Ground Station communicate with each other via Wi-Fi network. It defines the syntax, semantics, and synchronization of the communication.

## 6.1 Float Value Transmission

Most of data exchanged between the Quadricopter and the Ground Station are float numbers. They are stored in memory in IEEE 754 floating-point format, which has about seven significant numbers.

Each IEEE 754 floating-point number is stored in the memory as 4 bytes. There are several ways to transfer this number:

- Sending a human readable string that represents its value,
- Sending a hex string that represents the bytes stored in the memory,
- Sending directly the raw bytes in the memory.

Let see through each method by the following examples to see the pros and cons of them.

### Method 1

The number will be sent as a string that represents for its value.

Value	Bytes stored in memory	Bytes to be sent	Total bytes to be sent
3.1	0x66-66-46-40	0x33-2E-31"3.1"	3
3.141593	0xDC-0F-49-40	0x33-2E-31-34-31-35-39-33 "3.1415926"	8
3.141593E-6	0x29-D4-52-36	0x30-2e-30-30-30-30-33-31-34-31-35-39-33 "0.00003141593"	13
3.141593E12	0x62-DD-36-54	0x33-31-34-31-35-39-33-30-30-30-30-30-30 "3141593000000"	13

### Method 2

The number will be sent as a hex string that represents the bytes stored in the memory.

Value	Bytes stored in memory	Bytes to be sent	Total bytes to be sent
3.1	0x66-66-46-40	0x36-36-36-36-34-36-34-30 "66664640"	8
3.141593	0xDC-0F-49-40	0x44-43-30-46-34-39-34-30 "DC0F4940"	8
3.141593E-6	0x29-D4-52-36	0x32-39-44-34-35-32-33-36 "29D45236"	8
3.141593E12	0x62-DD-36-54	0x36-32-44-44-33-36-35-34 "62DD3654"	8

### Method 3

The raw bytes of the number in the memory will be sent directly.

Value	Bytes stored in memory	Bytes to be sent ( <i>non-readable characters</i> )	Total bytes to be sent
3.1	0x66-66-46-40	0x66-66-46-40	4
3.141593	0xDC-0F-49-40	0xDC-0F-49-40	4
3.141593E-6	0x29-D4-52-36	0x29-D4-52-36	4
3.141593E12	0x62-DD-36-54	0x62-DD-36-54	4

It is not easy to say what method is better than one.

- Method 1 is easy to implement for both sending side—just “print” out the variable, and receiving side—“read” the string and parse it into float value. Since then, the transmitted data are readable at data level, also easy to log and to debug. The size of transmitted data completely depends on the value of the variable.
- Method 3 is more complex than Method 1 in implementation: the variable must be accessed at byte-level. The transmitted data are not readable, thus difficult to log or to debug. The size of transmitted data is fixed at 4 bytes for each float value.

- Method 2 is a hybrid method. We still have to access to each byte of the float variable. However, since this method send readable characters, the transmitted data can easily be read and logged for debugging. Compare to Method 1, it is better in the point of view of designing a transmission protocol: the size of each information to be sent is fixed.

The following tables summarizes the cons and pros of each method.

**Table 6-1 – Methods for transmitting a float value**

Method	Complexity of implementation	Bandwidth usage	Ease of logging and debugging
<b>Readable string (1)</b>	Simple	High	Easy to log Easy to read
<b>Hex string (2)</b>	Normal	Average	Easy to log Readable
<b>Byte-level (3)</b>	Normal	Optimized	Unreadable

Our transmitting device have a wide throughput (11 Mbps Wi-Fi module operating at 115 200 baud). That means, within 10 ms, about 14 float values can be transmitted if the Method 2 is being used. It is abundant since we have no more than about 20 values to be transmitted (both send and receive) at a time, and the period of the transmitting task is 100 ms. Thus, the Method 2 has been chosen for developing our communication protocol.

## 6.2 Ground-to-Air Protocol

The Quadricopter will be controlled from the Ground Station. Ground Station sends orders or reference altitude & attitude to the Quadricopter. For the PID controller, the gains sometimes need to be tuned. These values are also updated via the Ground Station.

Currently, the Quadricopter only operates in the assisted mode. Thus, the protocol has only been designed for transferring these data type:

- The orders: the reference roll, pitch, yaw and altitude,
- The gains for the PID controller: Kp, Ki, Kd for roll, pitch, yaw channel,
- The mode: assisted mode or automatically mode.

There is an ID assigned for each data type. Since then, the protocol can easily be extended to send other data type such as waypoints.



### 6.2.1 General Syntax

Data sent to the Quadricopter in form of a sentence. Each sentence begins with a begin-transmission <STX> character for synchronizing, ends with an end-of-transmission <EOT> character for closing the transmission. The sentence may consist one or more blocks of data. Every block also has a begin character '\$', then the data type ID, following with the data to be transferred, and ends with an end-of-block-transmission <EBT> character.

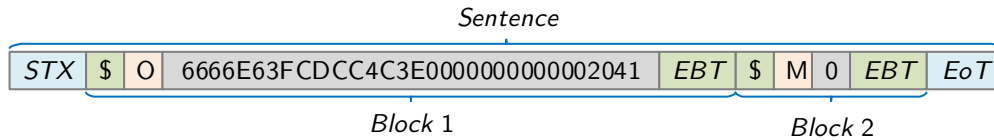


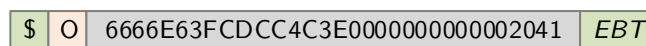
Figure 6-1 – Ground data structure

The control characters are non-readable. For the ASCII code of them, please refer to the Table 6-2.

### 6.2.2 Block Semantic

#### “Orders” block

“Orders” block consists of four floating-point values that are the desired roll, pitch, yaw, and altitude. These values are encoded into hex codes (*little* endian) and sent sequentially. The total size of the block is  $4 \times 8 + 2 + 1 = 35$  bytes. The block ID is ‘O’.



Roll:	1.8 degrees	– '6666E63F
Pitch:	0.2 degrees	– 'CDCC4C3E
Yaw:	0	– '00000000
Altitude:	10 meters	– '00002041

Figure 6-2 – “Orders” block

#### “Gains” block

In the PID controller, for each channel (roll, pitch, yaw, and altitude), there is a set of proportional gain, integral gain, and derivative gain ( $K_P, K_I, K_D$ ).

One “Gains” block consists one set of gains for one channel. The block ID is ‘G’, following by a character indicates the related channel (‘0’ for roll, ‘1’ for pitch, and ‘2’ for yaw). The values are encoded into hex codes (*little* endian) and sent sequentially. The total size of each blocks is  $3 \times 8 + 2 + 2 = 28$  bytes.

\$	G	0	77BE7F4017B7D1387593AE40	EBT
\$	G	1	0000004017B7D138C7BA6840	EBT
\$	G	2	3333333F6F12833A4E621840	EBT

Roll: 3.9960, 0.0001, 5.4555  
 — '77BE7F4017B7D1387593AE40  
 Pitch: 2.0000, 0.0001, 3.6364  
 — '0000004017B7D138C7BA6840  
 Yaw: 0.7000, 0.0010, 2.3810  
 — '3333333F6F12833A4E621840

Figure 6-3 – “Gains” block

### “Mode” block

Currently, the Quadricopter only operates in assisted mode. This block is define for later use. The block ID is ‘M’ and following by a character represents for the mode: ‘0’ for assisted mode and ‘1’ for automatically mode. The total size of each block is 2+2=4 bytes.

\$	M	0	EBT
----	---	---	-----

Figure 6-4 – “Mode” block

## 6.2.3 Summary

Table 6-2 – Control characters used in communication protocol

	Synch Byte	End Byte
<b>Sentence</b>	STX (0x02)	EOT (0x04)
<b>Block</b>	‘\$’ (0x24)	EBT (0x17)

Table 6-3 – Three block types of ground data

Block	ID	Data	Size
<b>Orders</b>	O	4×float	35 bytes
<b>Gains</b>	G0, G1, G2	3×float	28 bytes
<b>Mode</b>	M	1×char	4 bytes

One sentence can consist of more than one block. However, in action, when the Wi-Fi shield receives data from the network, it will put these data into the buffer of FlyMaple serial port. When the buffer is full, if another new byte comes, it will overwrite the current data in the buffer, data loss occurs. Since then, the size of a sentence should not exceed the buffer size – 63 bytes – to avoid buffer overflow.

Thus, we can only combine one “Mode” block with another block, or two “Gains” block in a sentence. However, the mode is rarely changed in operation.

The gains are usually tuned for only one channel at a time. Therefore, this combination is not necessary. Since then, this protocol can be optimized by completely remove the <STX> and <EOT> bytes. Every block will be considered as a complete message and will be sent at every time. This optimization only gives an insignificant reduction on bandwidth usage, but simplifies the receiver code on the Quadricopter.

## 6.3 Air-to-Ground Protocol

The Ground Station sends many types of data (orders, gains, mode...). Difference from it, data form sent from Quadricopter is unchanged. One complete Quadricopter message consists of these informations: the current MCU time stamp, attitude (roll, pitch, and yaw), altitude, and GPS informations (GPS time, longitude, latitude, altitude, satellite number, quality index). We call each information as a field. Two successive fields will be separated from each other by a space character. Each message starts with synch character '\$' and ends with a carriage return immediately followed by a line feed ("r\n"). The total size of a message is 87 bytes max. The protocol semantic is shown in the following table.

Table 6-4 – Air-to-Ground protocol semantic

No.	Field	Type	Transfer form	Field length*	Unit
0	Synch byte	Char	'\$' (0x24)	1 char	-
1	Time stamp	UInt	Hex code	8+1 chars max	Micro seconds
2	Roll	Float	Hex code	8+1 chars	Degrees
3	Pitch	Float	Hex code	8+1 chars	Degrees
4	Yaw	Float	Hex code	8+1 chars	Degrees
5	Altitude	Float	Hex code	8+1 chars	Meters
6	GPS time	Float	Hex code	8+1 chars	Degrees
7	Longitude	Float	Hex code	8+1 chars	Degrees
8	Latitude	Float	Hex code	8+1 chars	Degrees
9	Altitude	Float	Hex code	8+1 chars	Meters
10	Sat. number	Byte	Hex code	1+1 chars	-
11	Quality index	Char	Char	1 char	-
12	End bytes	Chars	"r\n" (0x0D 0x0A)	2 chars	-

\* "+1" is the size of space character (0x20) used as the separator

The hex codes in fields 1 and 10 are encoded by the Maple IDE built-in method, `HardwareSerial::print(variable, HEX)`. The length of the strings returned from this method depends on the range of the numbers (for example, 15 is encoded as "E" which has the length of 1, but 16 is encoded as "10" which has the length of 2). Thus, field 1 has an undefined length. For the field 10 (satellite

number), its value is in the range of 0 to 12, which has the related hex code length of only one character. Fields 2 to 9 are float numbers, of which the hex codes have the fixed length of 8 characters. Since then, only one separator is needed after field 1. For the fields 2 to 11, they can be spitted from the received message by their length; the delimiters can be completely removed.

### Pros and cons of delimiters

Removing delimiters means reducing the message size, thus reducing the bandwidth usage and the transmitting time, lowering the potential buffer overflow on the receiver.

Let see how a message is sent. Firstly, data will be sent from FlyMaple to the Wi-Fi shield via RS-232 interface (serial port). The Wi-Fi shield will then forwarding the data into the network. On the Ground Station, these data will be acquired and put into the buffer by the network stream before being processed the other task.

Since the buffer size of the network stream on the Ground Station receiver is very large (8 KB), buffer overflow cannot occur. The only thing that limits the message size is the bandwidth. The baud rate of the RS-232 transmission between the FlyMaple and the Wi-Fi shield is 115 200 baud (very slow compared to the Wi-Fi connection speed, 11 Mbps).

In RS-232 transmission, one byte take 10 bits of transmission (1 start bit + 8 bits of data + 1 stop bit). The size of one message is 87 bytes, since then the *maximum* time of transmission for one message is:

$$\frac{87 \times 10}{115\,200} \approx 0.0076 \text{ s} = 7.6 \text{ ms}$$

(Note that the baud rate is not equal to the bit rate. In real case, the bit rate may exceed the baud rate.)

By removing these 9 delimiters, a reduction of  $9/87=10.3\%$  of message size is achieved. In this case, the *maximum* transmitting time of each message is:

$$\frac{(87 - 9) \times 10}{115\,200} \approx 0.0068 \text{ s} = 6.8 \text{ ms}$$

A reduction of 10.5% (0.8 ms) transmitting time is relatively noticeable. However, let put this into the context. The period of the transmitting task is 100 ms, so this reduction will lower an amount of  $0.8/100=0.8\%$  processor utilization, which is negligible.

On the contrary, when the fields are split by delimiters, if data loss occurs in a field, it does not effect to the other following fields. In additional, delimiters

make the implementation of the receiver becomes easier and simpler. With delimiters, the received message can easily be split into fields as a string array by using the method `string.Split(' ')`. Without delimiters, fields in received message can be “extracted” using `string.Substring(Int32 index1, index2)` method.

From these cons and pros, the protocol is implemented with the delimiters.

## Chapter 7

# Ground Station Design

*Ground Station plays an important role in the system. It is the only mean to control and monitor the Quadricopter.*

*Ground Station receives data from the Quadricopter: altitude, attitude, GPS position..., and displays them on screen via the user interface. Orders, gains... are entered/changed via the Ground Station; they will be sent to the Quadricopter...*

## 7.1 Overview

User interacts with Ground Station via user interface (UI). The UI displays the information of the Quadricopter (current altitude, attitude, GPS...), connection status... on the screen. Through the UI, user can monitor and control the Quadricopter by changing the orders, tuning the gains, changing the operating mode...

This Ground Station is a Win Form Application that developed in Microsoft Visual Studio 2013 (free student license) using C#.NET. It can be separated into two main “modules,” (or actually, the two groups of functional methods):

- The user interface (UI): updates informations to screen, handles user events (button click, text box leaved, scroll bar scrolled...); updates the connection status; logs events, messages.
- The Networking module: establishes and monitors connection to the Quadricopter, acquires and send data from/to the Quadricopter; sends data to the external module “Artificial Horizon.”

The external module Artificial Horizon indicates the attitude of the Quadricopter as a visual graphic.

In this chapter, only general specifications will be discussed. The Ground Station and its external modules (Artificial Horizon, AHcommLib, and HexCodec) are described in details in the *Technical Note #3*.

## 7.2 User Interface

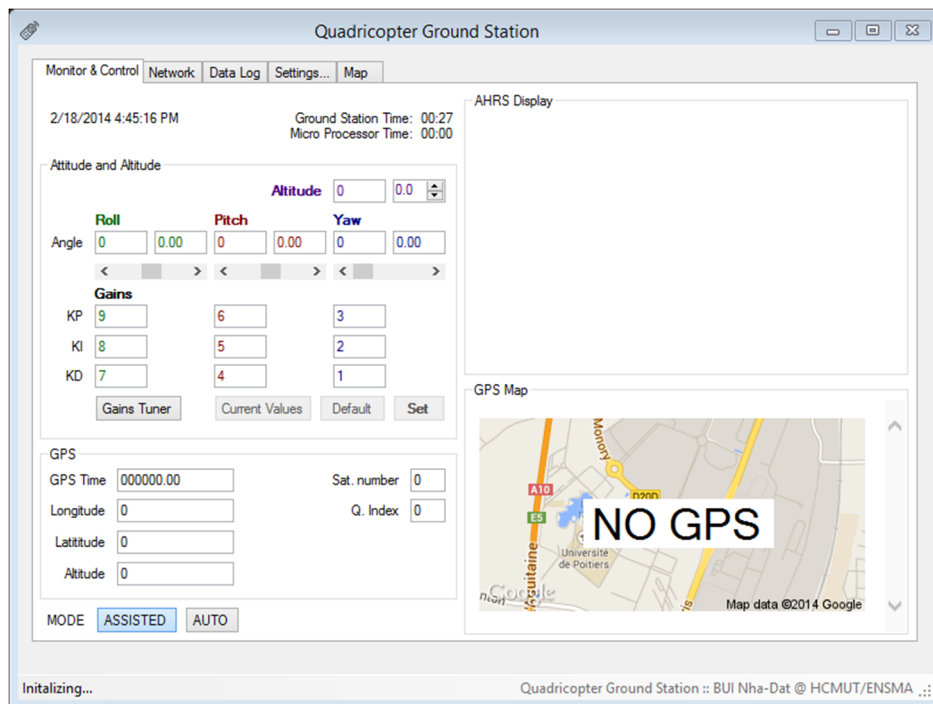


Figure 7-1 – Ground Station user interface – Monitor & Control tab

The user interface consists of these tabs:

- **Monitor & Control:** helps user to monitor and control the Quadricopter.
  - The current date time and MCU time are display on the top.
  - The panel *Altitude and Attitude* displays the current state of the Quadricopter, gathers the desired values via both text boxes and scroll bars; there is the *gains tuner* for tuning the gains of the Quadricopter PID controllers.
  - The panel *GPS* displays the current GPS information.

- On the bottom, there are two toggle buttons for displaying and toggling the current operating mode.
- On the right hand side, the *AHRS Display* is left blank for later use if necessary. Currently, the free module Artificial Horizon is used instead.
- The *GPS Map* display the current position of the Quadricopter from its longitude and latitude. This map uses Google Maps API. With Google Maps API, we can do path tracking with intractable map. However, due to the limitation of time, this GPS is no more than a static map centered at the current position of the Quadricopter.

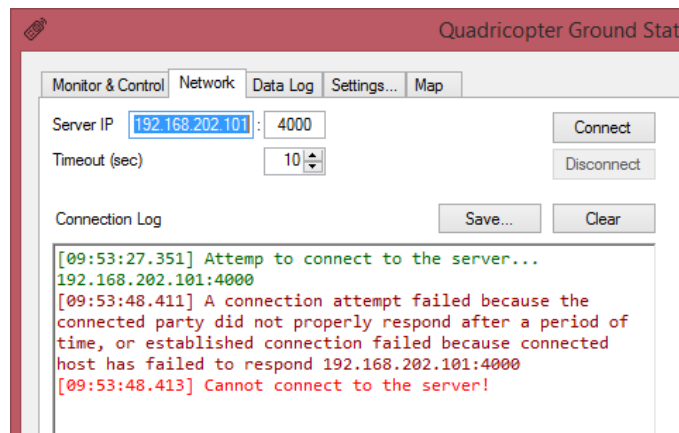


Figure 7-2 – Ground Station user interface – Network tab

- **Network:** IP address of the Quadricopter is defined in this tap. Two buttons *Connect* and *Disconnect* help to establish or close the connection to the defined IP. The timeout interval can be changed. The *Connection Log* logs all the events.
- **Data log:** log the sent and received messages.
- **Settings... & Map** are currently not implemented yet. Conceptually, the settings tab helps to save such data that the Quadricopter IP, the timeout interval, the default gains; the Map will implement an interactive map that can zoom in/out, track the Quadricopter, display the waypoints (in auto mode) , etc...

The interface will be updated by the *UI Updaters* – the methods: `UpdateUIClocks`, `UpdateUIAltitude`, `UpdateUIAttitude`, and `UpdateUIGPS`. These UI Updaters are triggered periodically every 100 ms by the timer `timer-Interface` (Figure 7-3).



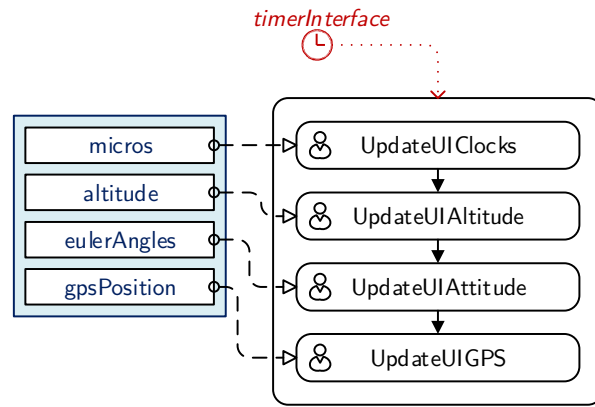


Figure 7-3 – UI Updaters

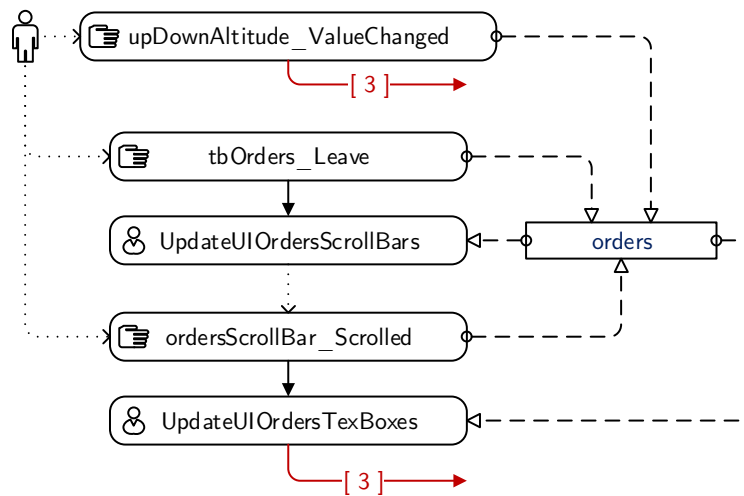


Figure 7-4 – Event handlers – orders

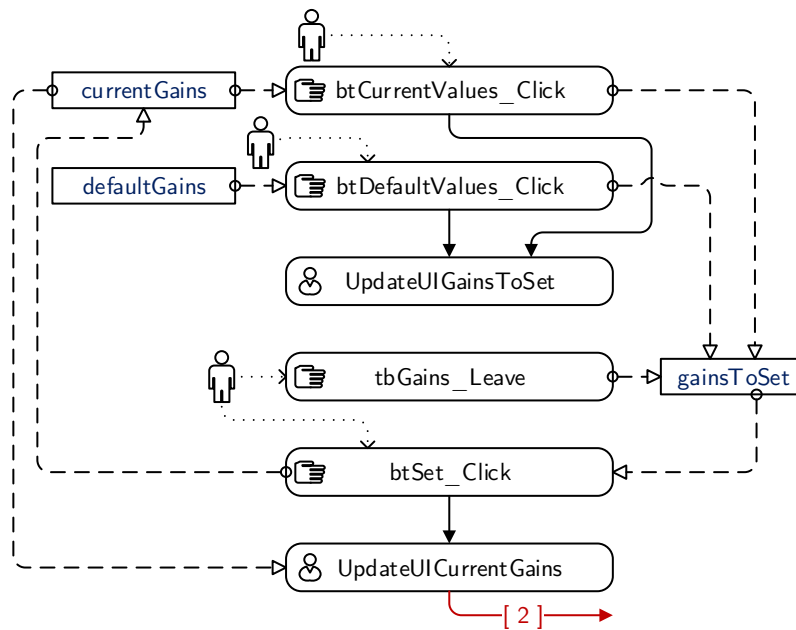


Figure 7-5 – Event handlers – gains tuner

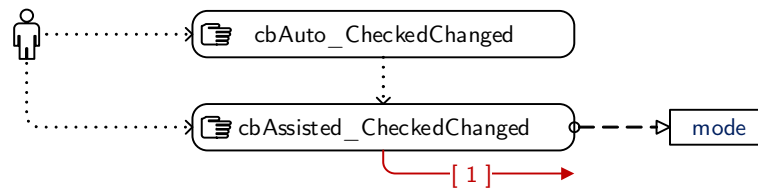


Figure 7-6 – Event handlers – mode

When user interacts with the UI (change text in the text boxes, scroll the scroll bars, and toggle the toggle buttons...), the related *event handler* will be invoked. Figure 7-4, Figure 7-5 and Figure 7-6 show the sequence/data flows when user triggers an event on the UI. The orders, gains, or mode will be parsed from UI components and put into related memory module (the global variable **orders**, **gains**, or **mode**), related *message updater* (see Part 0) will then be invoked (the sequence flows [1], [2] and [3]).

### Artificial Horizon

The free a340gc (Airbus A340 Glass Cockpit) project is used as a part of the Quadricopter Ground Station: the artificial horizon.

*“The Glass Cockpit Library (libGC) is created for the Airbus A340 Glass Cockpit (a340gc) which is an Open Source project. The a340gc project is part of the Airbus A340 simulator project of the IRADIS Foundation. The goal of libGC and a340gc is to create a free framework that can be used to build a glass cockpit upon.”*

—a340gc.iradis.org

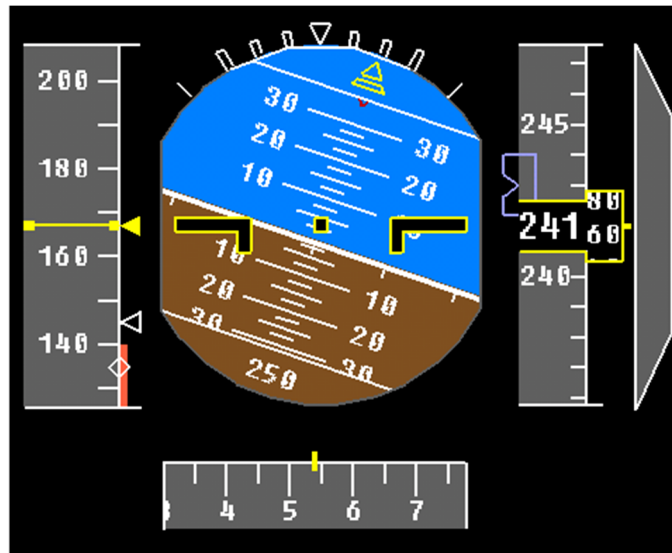


Figure 7-7 – Graphic rendered by the Artificial Horizon module

The Ground Station communicates with a340gc by sending RDDP packet via UDP. AHcommLib.dll, a library for preparing data stream in form of RDDP packet, has been built. *For further information, please refer to the Technical Note #3.*

## 7.3 Networking

Networking module consists of the **TransceiverDoWork** method (hereafter *transceiver*), which is invoked periodically by the timer **timerInterface**, a **SenderDoWork** method (hereafter *sender*), a **ReceiverDoWork** method (hereafter *receiver*), three functional methods (hereafter *message updaters*) **UpdateModeToSend**, **UpdateGainsToSend**, and **UpdateOrdersToSend**, and two control methods **Connect** and **Disconnect**.

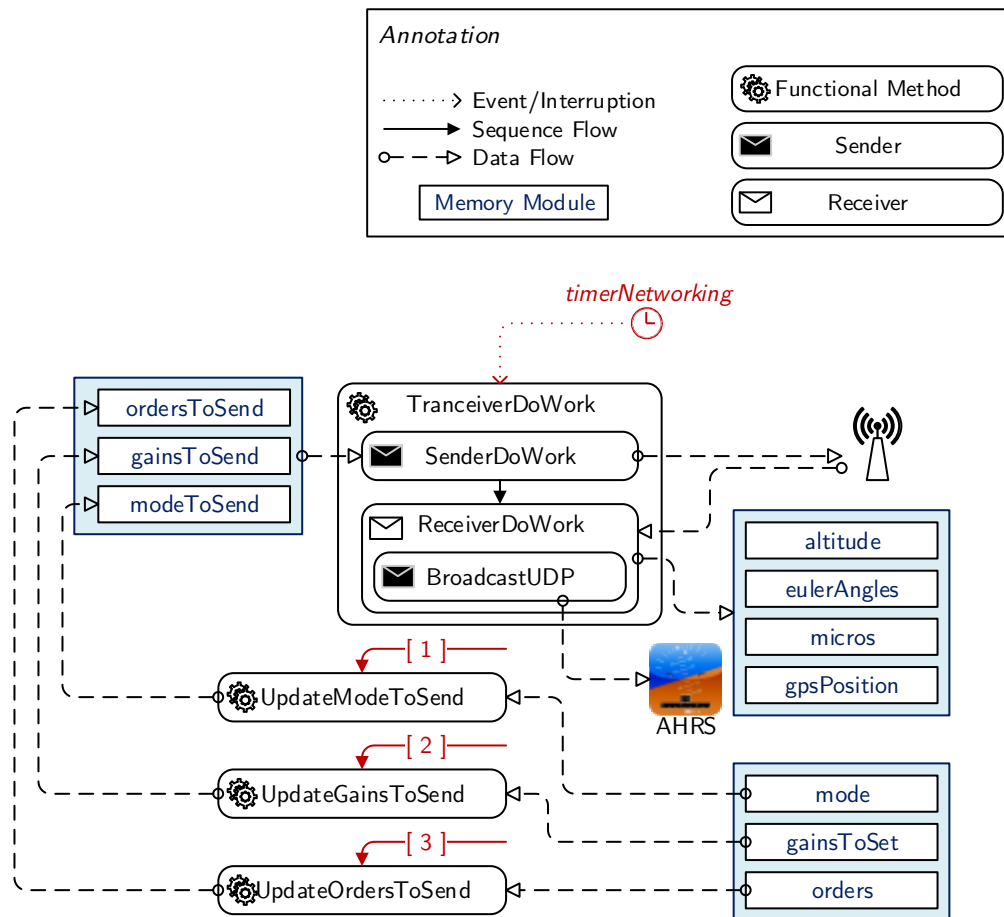


Figure 7-8 – Ground Station Networking Module

These **message updaters** are invoked by UI event handlers, whenever orders or mode is changed, or the new gains are set by user via UI. The invoked message updater prepare the message, following the transmission protocol, and put the message(s) in the related *message holder* (the strings **orderToSend**, **gain-sToSend**, **modeToSend**).

Once invoked by the timer, the **transceiver** checks the connection status (connected/disconnected, timeout). If the connection is OK, it will invoke the **SenderDoWork** method and then the **ReceiverDoWork** method.

- **ReceiverDoWork** will check the network stream. If the stream is available, the receiver waits for dollar sign '\$' on the data stream for synchronizing. Once synchronized, the complete message will be acquired and parsed; the global variables (**micros**, **altitude**, **eulerAngles**, and **gpsPosition**) will then be updated. If synchronization cannot be done for an interval of time, the timeout flag **isTimeout** will be turned on.
- **SenderDoWork** will check the length of the message holders. If one of these strings is not empty, it will be sent to the Quadricopter via network stream. After being sent, the related message holder will be cleared.

If timeout occurs, the flag **disconnectNow** will be turned on. **Disconnect** method will then be invoked.

The **Connect** method will try to establish a TCP connection to the Quadricopter via the IP and port number entered on the UI. If this TCP connection can be established, the timer **timerNetworking** will then be started, and an UDP socket (for sending data to Artificial Horizon module) will be opened.

The **Disconnect** method closes the connection and stops the timer.

## Chapter 8

# Simulink Code Generation

*Gene-Auto is an open source toolset for real-time embedded systems. The toolset takes as input a functional description of an application specified in a high level modelling language (Simulink/Stateflow/Scicos) and produces C (in close future also Ada) code as output.*

*The Gene-Auto toolset was developed during an ITEA project (project no 05018) by an European consortium with partners from France, Estonia, Belgium and Israel, as an open source alternative for industrial code generators that are currently in use for converting control system models to executable code in aerospace and automotive domains. The main goals of the project were to develop a code generator that is qualifiable according to the norms of DO-178B(C)/ED-12B and, at the same time, to guarantee long-term maintainability of this toolset through the open-source development.*

—geneauto.krates.ee

Nevertheless, why Gene-Auto? MATLAB already have the Real Time Workshop (RTW) and RTW Embedded Coder as the code generator for Simulink. However, since they are targeted to industrial coding purpose, generated code from this built in generator is very complex and difficult to understand, modify, and even difficult to integrate. For non-industrial purposes, Gene-Auto seems to be more “user-friendly” and easier to startup with.

This chapter will describe the steps to use Gene-Auto for our Simulink model—the PID flight control system.

## 8.1 Preparation: Java Runtime Environment

Java runtime environment (hereafter JRE) is required to run Gene-Auto. For Windows® OS, after installing JRE, environment variables must be added to run JRE in command line mode:

Table 8-1 – JRE environment variables in Windows® OS

Variable	Value
JAVA_HOME	[JRE Installation Path]
Path	Add “%JAVA_HOME%\bin”

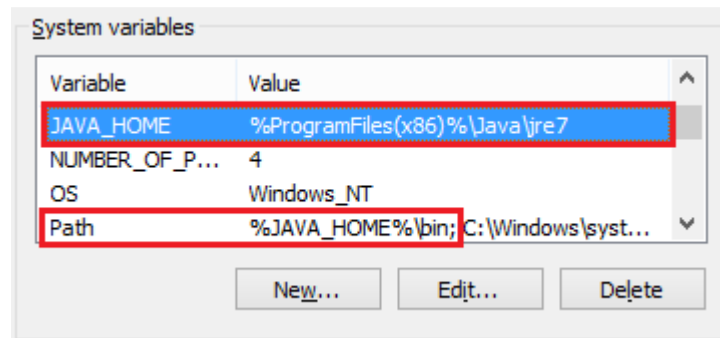


Figure 8-1 – Setting environment variables in Windows® OS

To check if the JRE is installed and configured correctly, we can run the “java – version” command:

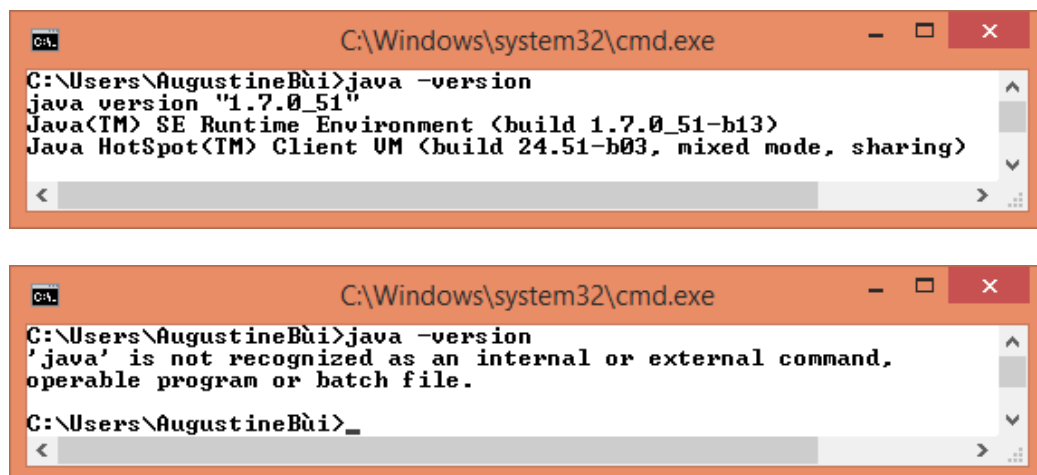


Figure 8-2 – JRE installed correctly (upper) and not correctly (lower)

## 8.2 Code Generating Procedure

The code generating procedure has three main phases:

1. Prepare the model in MATLAB/Simulink,

2. Generate code using Gene-Auto,
3. Integrate the generated code into the larger system.

### 8.2.1 Simulink Model Preparation

#### *Simulink model*

Currently, Gene-Auto only supports the Simulink/Stateflow model file in format version of *Simulink R14 SP3* or *r2006b*. Models in other Simulink/Stateflow file formats may pass, but the toolset is validated only for those two versions.

In action, several simple Simulink models were created in MATLAB/Simulink r2010b. Gene-Auto has been tested to generate code from these unsupported-format models. All of them passed the parsing phase in Gene-Auto, but caused the *unknown* critical error in the following phase. Not any code could be generated. These models were then converted into r2006b format (simply using the “*save as...*” function in Simulink). After all, the Gene-Auto can successfully generate the C codes.

One more thing should be pay attention to is the blocks that are supported by Gene-Auto are listed on Gene-Auto website (Gene-Auto.krates.ee/node/22).

Finally, the dimensions (size) and type of every in-port and out-port in the model must be defined.

#### *m-files*

Parameters that used in the Simulink model can be defined by *explicit* values via m-file(s); formulae are not supported. For example, assume we have three parameters **a1**, **a2**, and **a3** as: **a1** = 10, **a2** = 5 and **a3** = **a1** + **a2**. Then, in m-file, **a3** must be defined explicitly as **a1** = 15.

Parameters defined in m-files will become into constants in the generated C code.

### 8.2.2 Code Generation

Gene-Auto runs through command lines, as shown below. These command lines must be called from Gene-Auto home directory (directory that contains the genauto.\*.jar files).


```
SET GENE-AUTO_HOME=%~dp0/%  
java -Xmx1024m -cp "%~dp0Gene-Auto.GALauncher-2.4.10.jar"   
Gene-Auto.launcher.GALauncher "<path>\<modelName>.mdl" -m "<path>\<param>.m"
```

Figure 8-3 – Gen-Auto command lines

In these command lines, `<path>` is the path to the model file `<modelName>.mdl` and m-file `<param>.m`.

When entering the generating process, Gene-Auto creates a folder named `<modelName>_ga` in the `<path>`. The temporary files (xml, log, and metrics) generated during the generation will be stored in the sub folder named `tmp`. The log file saves all the on-screen information during the generating process. It will be sometimes useful for troubleshooting when the generating process is failed. The `code_metrics` file provides a better insight into the generated code: developer can get such informations as the number of variables, statements, comment lines, code lines, code size, and the dependency of each generated module.

Once the process is completed successfully, all the generated code will be written in the `<path>\<modelName>_ga` folder.

### 8.2.3 Code Integration

Gene-Auto is primarily used for *module (component) based* code generation. Currently, Gene-Auto does not produce any wrapper code to integrate the generated modules into a larger system. Such integration depends largely on the target platform and the integration is usually carried out by the end-users.

The pattern for integrating the code produced by Gene-Auto is following:

- **Import the root module generated by Gene-Auto**

Depending on the model, Gene-Auto generates one or more source code modules. One of them is the root module. By default, this has the name of the model. Import header of the root model, for example:

```
#include "../<modelName>_ga/<modelName>.h"
```

- **Create variable instances for the IO and State variables**

For example:

```
t_<modelName>_io io;
t_<modelName>_state state;
```

- **Call the `init(..)` function to initialize the (sub)system that was generated with Gene-Auto**

For example:

```
<modelName>_init(&state);
```

- **Implement the read-compute-write cycle**

Embed the *read-compute-write* chain in a task called by the operating system.

For example:

```
task_doTask {
    // ...
    /* read inputs */
    do_read(&(io.x), &(io.b));
}
```



```

    /* perform a computation step */
    <modelName>_compute(&io, &state);
    /* write outputs */
    do_write(io.y);
    // ...
}

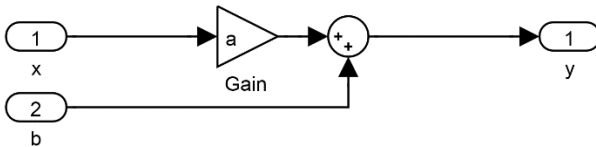
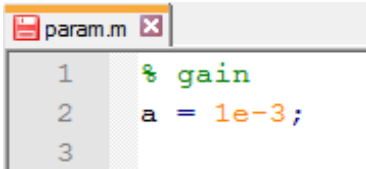
```

**Note:** Do not create any custom files in the folder “<modelName>\_ga” generated by Gene-Auto. This folder is always emptied and recreated, when you rerun Gene-Auto. However, you can refer to the source code in this folder from external code, since the file and identifier naming follows fixed rules.

## 8.2.4 Summary

The three above steps are summarized in the following table:

**Table 8-2 – Three main steps of code generating procedure**

1. Prepare the Simulink model in MATLAB/Simulink	
	
<ul style="list-style-type: none"> <li>• Simulink model &lt;modelName&gt;.mdl must be saved in <i>r2006b</i> format.</li> <li>• Blocks used in model must be supported by Gene-Auto.</li> <li>• Parameters defined in m-file must be in <i>explicit</i> values.</li> <li>• The dimensions (size) and type of every in-port and out-port in the model must be defined.</li> </ul>	
2. Run Gene-Auto – generate code	
<pre> SET GENE-AUTO_HOME=%~dp0/% java -Xmx1024m -cp "%~dp0Gene-Auto.GALauncher-2.4.10.jar" Gene-Auto.launcher.GALauncher "&lt;path&gt;\&lt;modelName&gt;.mdl" -m "&lt;path&gt;\&lt;param&gt;.m" </pre>	
<ul style="list-style-type: none"> <li>• Common generated files:           <ul style="list-style-type: none"> <li>&lt;modelName&gt;.c</li> <li>&lt;modelName&gt;.h</li> <li>&lt;modelName&gt;_&lt;param&gt;.c</li> <li>&lt;modelName&gt;_&lt;param&gt;.h</li> <li>&lt;modelName&gt;_types.h</li> <li>...</li> <li>GACCommon.h</li> <li>GATypes.h</li> </ul> </li> </ul>	
3. Integrate generated C code in embedded system's task	
<ul style="list-style-type: none"> <li>• Import the root module generated by Gene-Auto:           <pre>#include "../&lt;modelName&gt;_ga/&lt;modelName&gt;.h"</pre> </li> <li>• Create variable instances for the IO and State variables:           <pre>t_&lt;modelName&gt;_io io; t_&lt;modelName&gt;_state state;</pre> </li> </ul>	

- Call the `init(..)` function to initialize the (sub)system that was generated with Gene-Auto:  
`<modelName>_init(&state);`

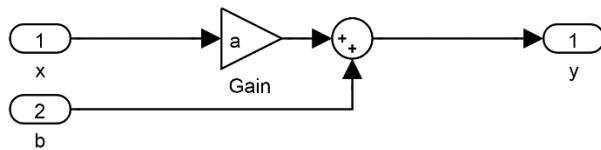
- Implement the read-compute-write chain in a task:

```
task_doTask {  
    // ...  
    /* read inputs */  
    do_read(&(io.x), &(io.b));  
    /* perform a computation step */  
    <modelName>_compute(&io, &state);  
    /* write outputs */  
    do_write(io.y);  
    // ...  
}
```

## 8.3 Test Case: the Very Simple Model

Simulink model file and m-file:

VerySimpleModel.mdl



param.m

```
% gain
a = 1e-3;
```

Generated modules:

- Module **VerySimpleModel**  
Code file: VerySimpleModel.c  
Header file: VerySimpleModel.h  
Dependencies: GATypes.h, VerySimpleModel\_types.h, VerySimpleModel\_param.h,
- Module **VerySimpleModel\_param**  
Code file: VerySimpleModel\_param.c  
Header file: VerySimpleModel\_param.h  
Dependencies: GATypes.h,
- Module **VerySimpleModel\_types**  
Header file: VerySimpleModel\_types.h  
Dependencies: GATypes.h,

The module `<modelName>_types` defines the I/O structure for the model:

```
typedef struct {
    GAREAL x;
    GAREAL b;
    GAREAL y;
} t_VerySimpleModel_io;
```

The module `<modelName>_<param>` contains the parameter (as constant) that is defined in the input m-file param.m:

```
const volatile GAREAL a = 1.0E-3;
```

The main module `<modelName>` contains the initializing routine and the main calculation routine of the model. In this model, there is not any block that contains the initial condition, thus the initializing code is empty.

```
void VerySimpleModel_init() {
}

void VerySimpleModel_compute(t_VerySimpleModel_io *_io_) {
```

```

/* Output from <SystemBlock: name=VerySimpleModel>
    /<SourceBlock: name=x>
    /<OutDataPort: name=> */
GAREAL x;
/* Output from <SystemBlock: name=VerySimpleModel>
    /<SourceBlock: name=b>
    /<OutDataPort: name=> */
GAREAL b;
/* Output from <SystemBlock: name=VerySimpleModel>
    /<CombinatorialBlock: name=Gain>
    /<OutDataPort: name=> */
GAREAL Gain;
/* Output from <SystemBlock: name=VerySimpleModel>
    /<CombinatorialBlock: name=Sum>
    /<OutDataPort: name=> */
GAREAL Sum;
/* START Block: <SystemBlock: name=VerySimpleModel>/<SourceBlock: name=x> */
x = _io_->x;
/* END Block: <SystemBlock: name=VerySimpleModel>/<SourceBlock: name=x> */
/* START Block: <SystemBlock: name=VerySimpleModel>
    /<CombinatorialBlock: name=Gain> */
Gain = a * x;
/* END Block: <SystemBlock: name=VerySimpleModel>
    /<CombinatorialBlock: name=Gain> */
/* START Block: <SystemBlock: name=VerySimpleModel>/<SourceBlock: name=b> */
b = _io_->b;
/* END Block: <SystemBlock: name=VerySimpleModel>/<SourceBlock: name=b> */
/* START Block: <SystemBlock: name=VerySimpleModel>
    /<CombinatorialBlock: name=Sum> */
Sum = Gain + b;
/* END Block: <SystemBlock: name=VerySimpleModel>
    /<CombinatorialBlock: name=Sum> */
/* START Block: <SystemBlock: name=VerySimpleModel>/<SinkBlock: name=y> */
_io_->y = Sum;
/* END Block: <SystemBlock: name=VerySimpleModel>/<SinkBlock: name=y> */
}

```

## 8.4 Quadricopter Flight Control System Implementation

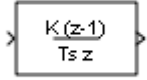
As mention previously, Gene-Auto does not produce any wrapper code to integrate the generated modules into a larger system. Such integration depends largely on the target platform and the integration is usually carried out by the end-users. Since then, the sample time is determined by the larger system. Any block concerns to the time domain (for example, the integrators and the derivative blocks) are not supported.

One of the controllers for our Quadricopter uses the PID controller, which always requires the integral and derivative calculation. Since the built-in integrator and derivative blocks are not supported by Gene-Auto, equivalent models must be carried out in order to implement this type of controller.

Thus, in this part, we will build the integrator and derivative model by using the blocks that supported by Gene-Auto. These models will then be used as the replacement of the built in integrator and derivative block.

#### 8.4.1 Equivalent Models of Integrator and Derivative blocks

##### Discreet Derivative Block



The built-in Discrete Derivative block computes an optionally scaled discrete time derivative as follows:

$$y(t_n) = \frac{u(t_n) - u(t_{n-1})}{T_s}$$

where  $u(t_n)$  and  $y(t_n)$  are the block's input and output at the current time step, respectively,  $u(t_{n-1})$  is the block's input at the previous time step, and  $T_s$  is the simulation's discrete step size, which must be fixed.

Follows these above calculation steps, an equivalent model has been built:

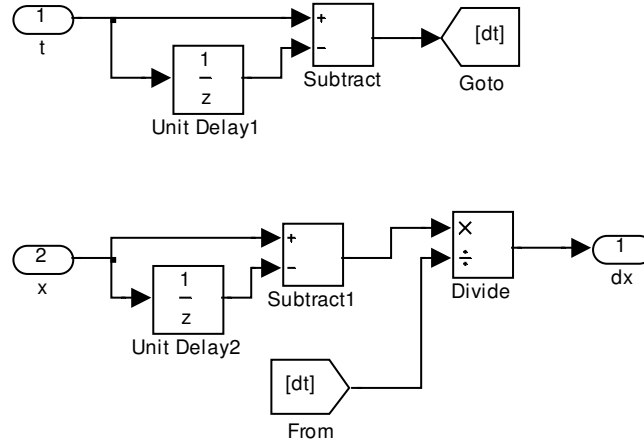


Figure 8-4 – Equivalent discreet derivative model

In this model,  $x$  is the input signal,  $dx$  is the output, and  $dt$  is the step size (sample time) will be calculated from the triggering time  $t$ .

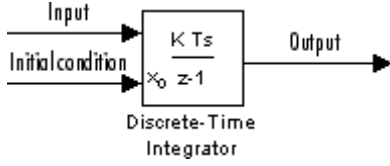
##### Discreet Integrator Block

The built-in integrator block can integrate a signal using one of the three methods:

- Forward Euler method,
- Backward Euler method, and
- Trapezoidal method.

Assume that  $u$  is the input,  $y$  is the output, and  $x$  is the state. For a given step  $n$ , Simulink updates  $y(n)$  and  $x(n+1)$ . In integration mode,  $T$  is the block sample time (delta  $T$  in the case of triggered sample time).

**Forward Euler method** (the default), also known as Forward Rectangular, or left-hand approximation.



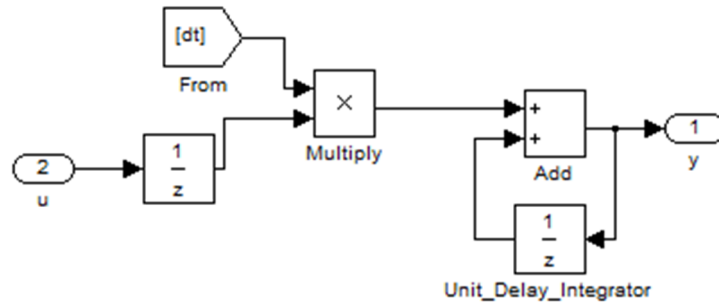
For this method, the expression for the output of the block at step  $n$  is:

$$y(n) = y(n-1) + Tu(n-1)$$

Let  $x(n+1) = x(n) + Tu(n)$ . The block uses the following steps to compute its output:

Step 0:	$y(0) = x(0) = IC$
	$x(1) = y(0) + Tu(0)$
Step 1:	$y(1) = x(1)$
	$x(2) = x(1) + Tu(1)$
Step $n$ :	$y(n) = x(n)$
	$\quad = y(n-1) + Tu(n-1)$
	$x(n+1) = x(n) + Tu(n)$

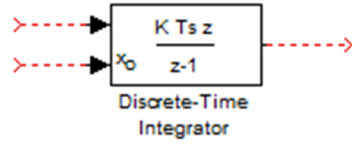
Follows these above calculation steps, an equivalent model has been built (in this model,  $dt$  is the sample time  $T$ ):



**Figure 8-5 – Equivalent integrator model: Forward Euler method**

The initial condition  $x(0) = IC$  of the integral is the initial condition of the *Unit\_Delay\_Integrator* block.

**Backward Euler method**, also known as Backward Rectangular or right-hand approximation.



For this method, the expression for the output of the block at step  $n$  is:

$$y(n) = y(n-1) + Tu(n)$$

Let  $x(n) = y(n-1)$ . The block uses the following steps to compute its output:

$$\begin{aligned} \text{Step 0:} \quad & y(0) = x(0) = IC \\ & x(1) = y(0) \\ \text{Step 1:} \quad & y(1) = x(1) + Tu(1) \\ & x(2) = y(1) \\ \text{Step } n: \quad & y(n) = x(n) + Tu(n) \\ & = y(n-1) + Tu(n) \\ & x(n+1) = y(n) \end{aligned}$$

Follows these above calculation steps, an equivalent model has been built (in this model,  $dt$  is the sample time  $T$ ):

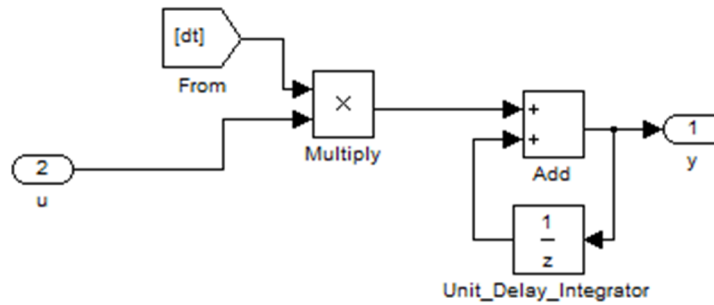
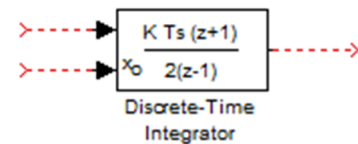


Figure 8-6 – Equivalent integrator model: Backward Euler method

The initial condition  $x(0) = IC$  of the integral is the initial condition of the *Unit\_Delay\_Integrator* block.

### Trapezoidal method



For this method, if  $T$  is variable (for example, obtained from the triggering times), the block uses the following algorithm to compute the outputs:

$$\begin{aligned} \text{Step 0:} \quad & y(0) = x(0) = IC \\ & x(1) = y(0) \end{aligned}$$

Step 1:

$$y(1) = x(1) + \frac{T}{2}(u(1) + u(0))$$

$$x(2) = y(1)$$

Step  $n$ :

$$y(n) = x(n) + \frac{T}{2}(u(n) + u(n-1))$$

$$= y(n-1) + \frac{T}{2}(u(n) + u(n-1))$$

$$x(n+1) = y(n)$$

Follows these above calculation steps, an equivalent model has been built (in this model,  $dt$  is the sample time  $T$ ):

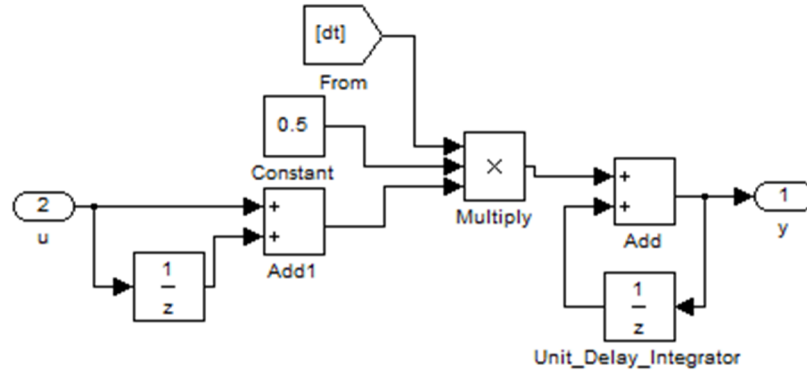


Figure 8-7 – Equivalent integrator model: Trapezoidal method

The initial condition  $x(0) = IC$  of the integral is the initial condition of the *Unit\_Delay\_Integrator* block.

## Test and comparisons

Test benches have been constructed for comparing the results from the built-in blocks and the above equivalent models.

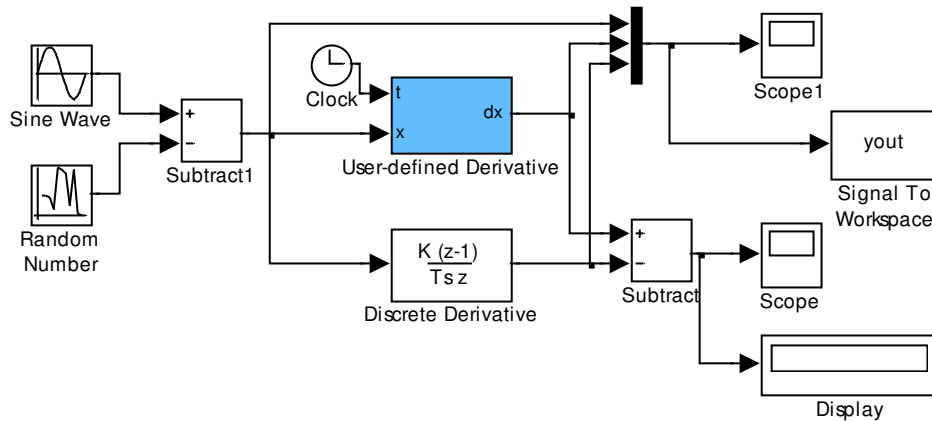
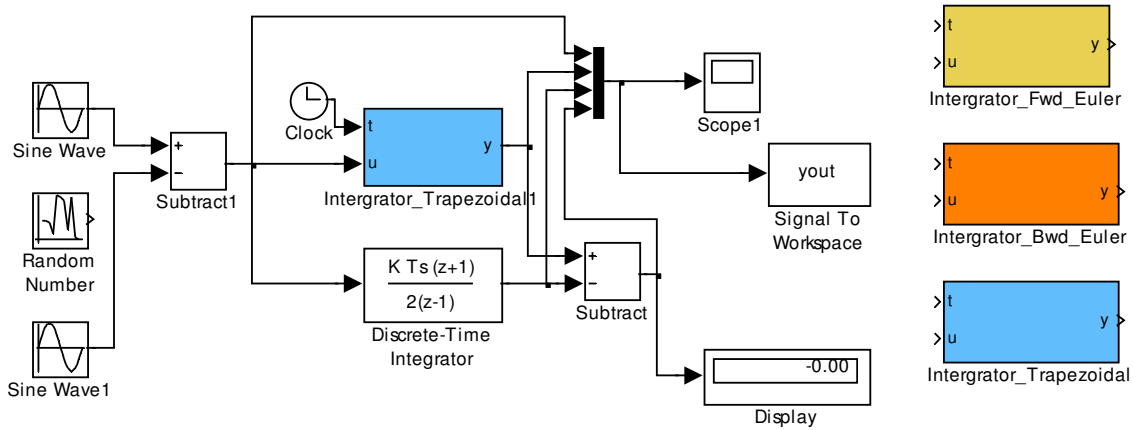


Figure 8-8 – Test bench for the equivalent derivative model





**Figure 8-9 – Test bench for the equivalent integrator model**

By observing the test bench runs a simulation of 500 seconds with the sampling time of 0.5 seconds (so, we have 1000 samples) and comparing the results of the built-in blocks with the equivalent models, the maximum relative errors were recorded in the following table:

**Table 8-3 – Comparison of equivalent models to the built-in blocks**

Calculation	Maximum relative error
	$\frac{ equivalent\ model - built-in\ block }{ built-in\ block }$
Derivative	0
Integral, Forward Euler	0
Integral, Backward Euler	0
Integral, Trapezoidal	Less than $8 \times 10^{-13}$

With this error level, these equivalent models can absolutely be used as the replacements of the unsupported built-in blocks.

## 8.4.2 Quadricopter PID Flight Control System Implementation

### 8.4.2.1 Simulink model preparation

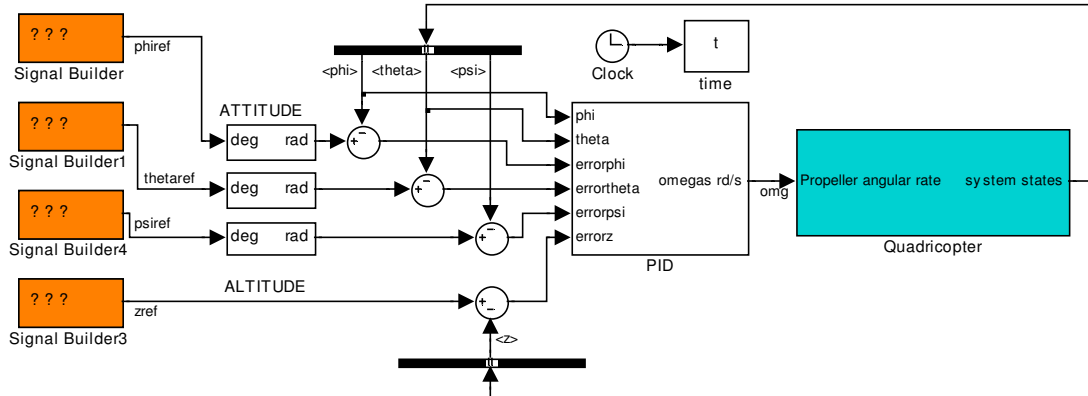


Figure 8-10 – The complete Quadricopter simulating model with PID controller (some components removed)

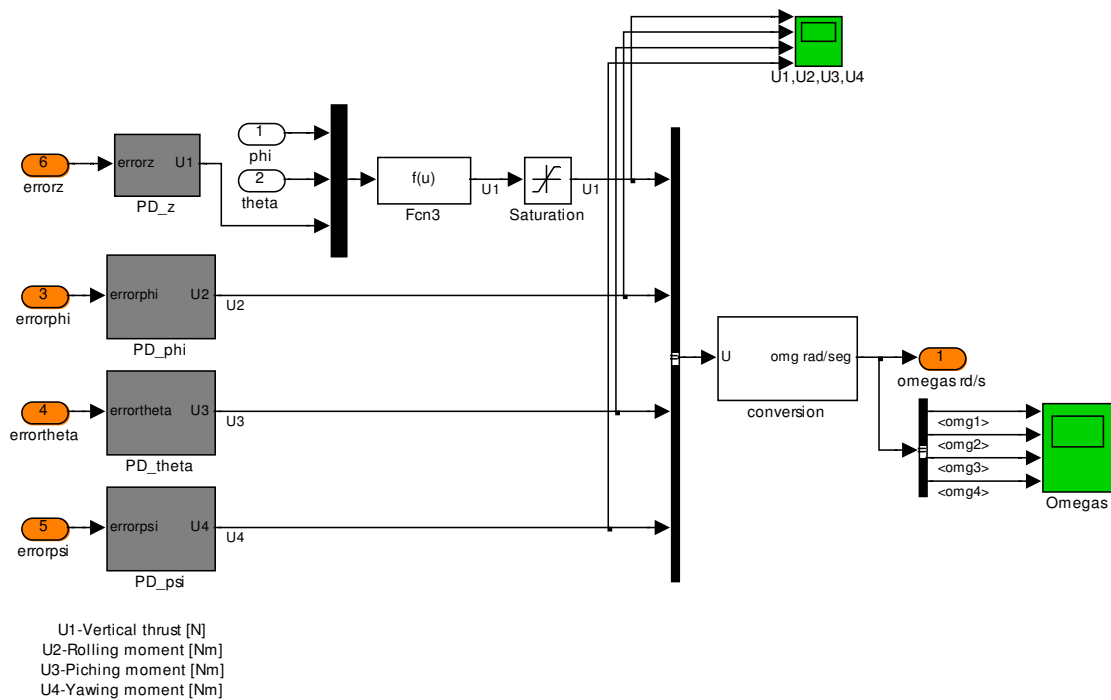


Figure 8-11 – The PID block, original design

Figure 8-10 shows the complete Quadricopter simulating model using the PID flight control system. The signal builders generate the reference values of the attitude and altitude for simulating. The “Quadricopter” block is the dynamic model of the real Quadricopter. The  $\langle \phi \rangle$ ,  $\langle \theta \rangle$ ,  $\langle \psi \rangle$  and  $\langle z \rangle$  is the feedback values of the attitude and altitude. The clock is the real time clock of

the system. The “PID” block is the flight control system. In integration, the feedback values are the measurements (Euler angles and altitude) acquired from the sensors, the reference values are the orders, and the clock is the MCU time.

Figure 8-11 shows more details of the PID block. This block consist of the single PID controllers for every channel (roll, pitch, yaw, and altitude), the limiters (saturation), the converter to convert the thrust/moments to the related rotating velocities—the omegas. **Currently, we will only generate code for the PID controllers for the four channels** (the PID\_phi, PID\_theta, PID\_psi and PID\_z blocks). Since the converter (“fcn3” block) and the limiter (“saturation” block) are depend on many of empirical parameters that have not been tested yet, they will be converted manually while doing the code integration, after measuring the characteristics of the motors and propellers. Each PID controller for a channel consists of the components as shown on the Figure 8-12.

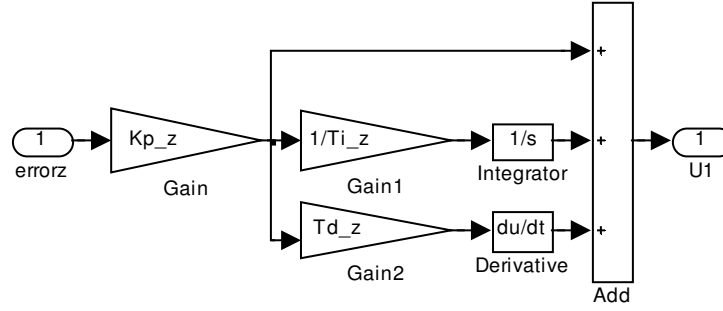


Figure 8-12 – PID controller for z-channel (“PID\_z” sub-block), original design

This original design need to be modified in order to pass into Gene-Auto. Figure 8-13, Figure 8-14 and Figure 8-15 show the modified model that compatible with Gene-Auto.

The modified model combined four controllers into one block “Controller.” In addition to the errors, the real time clock is added as another input variable (Figure 8-13).

In the original model, the gains’ values are defined by the m-file’s parameters (for example, Kp\_z, Ti\_z, and Td\_z in the Figure 8-12). As described previously, these m-file’s parameters will be converted into constants during the code generation. Since then, in the modified model, these blocks are replaced by the multipliers blocks, and the gains’ values are defined as input variables (in-port) (Figure 8-15).

The modified model also replace the integrator and derivative blocks with the equivalent models (Figure 8-15).

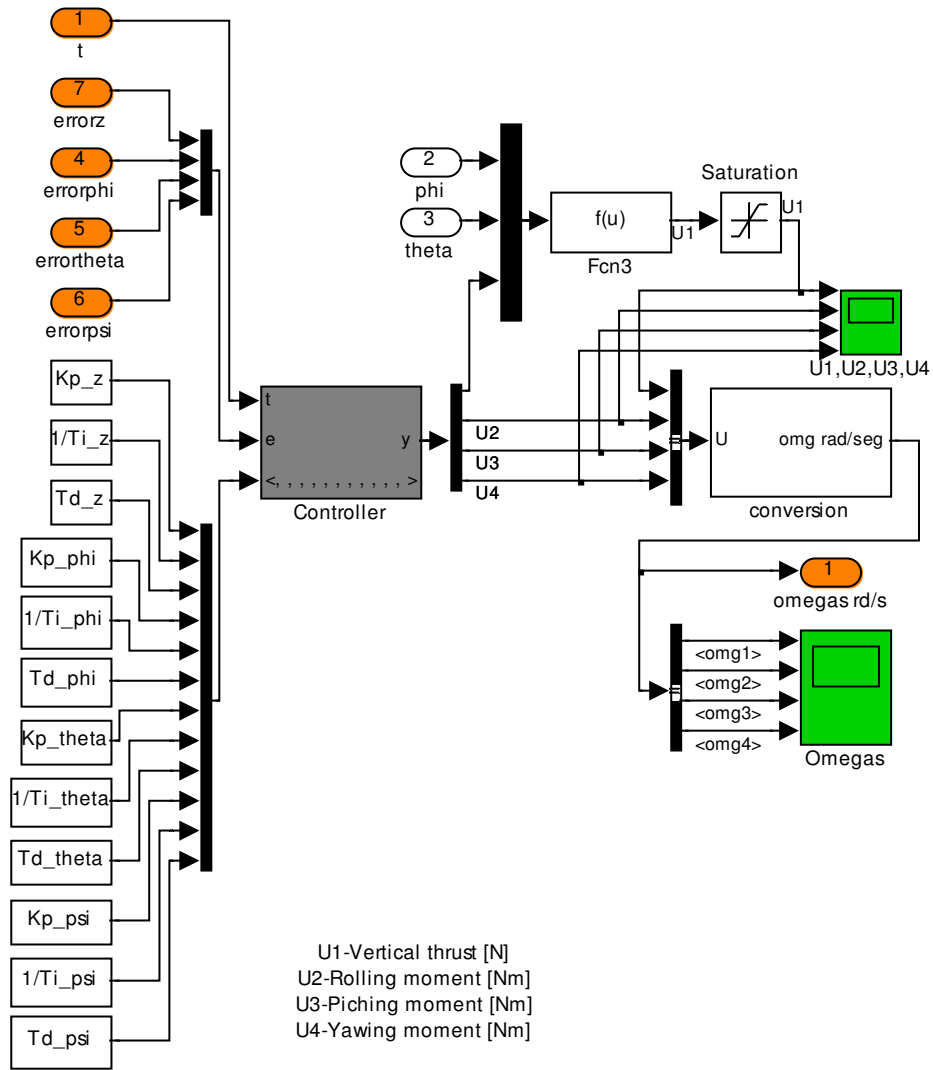


Figure 8-13 – The PID block, modified version

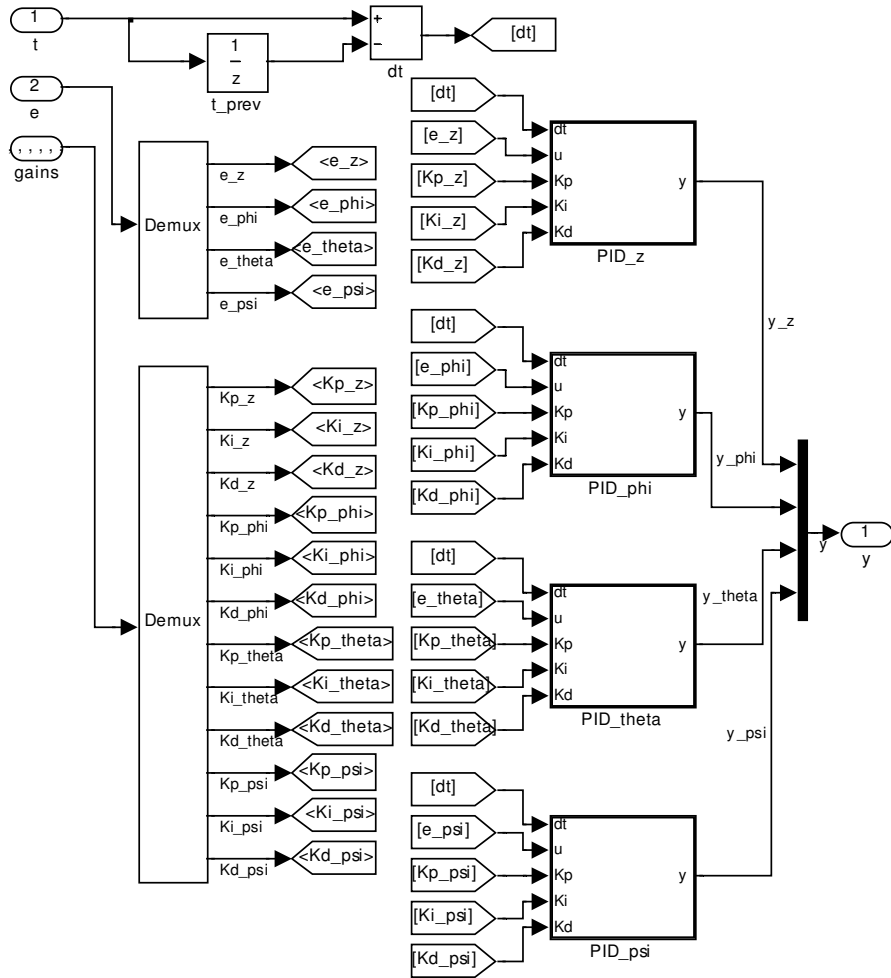


Figure 8-14 – The “Controller” sub-block of the modified “PID” block

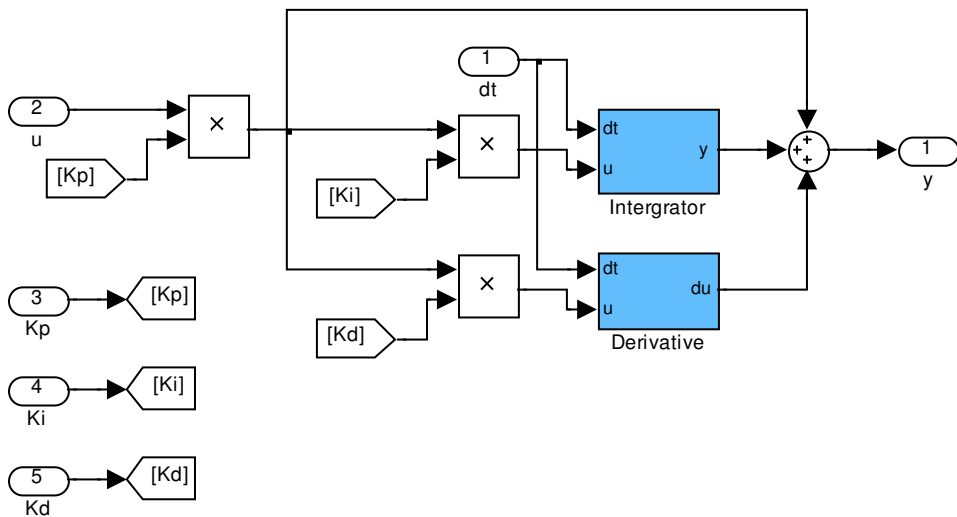


Figure 8-15 – PID controller for z-channel (“PID\_z” sub-block), modified version using alternative derivative and integrator blocks

After modifying the model to make it compatible with Gene-Auto, we move on the next phase: code generation. In the next part, the “Controller” block in the modified model will be converted in to code.

### 8.4.2.2 Code generation

Before moving on the code generation phase, the “Controller” block (Figure 8-14) must be saved as a separated model file, R2006B format. Let us name it “PID-controller.mdl.” In this model, all the parameters will be defined as input variables, there is not any m-file is required.

The data type and port dimensions of each Input and Output block is defined as in the following table.

**Table 8-4 – Controller I/O ports**

Block	Name	Data Type	Dimensions	Description
Inport 1	t	Double	1	MCU Time (s)
Inport 2	e	Double	4	Feedback error [0] e_z (m) [1] e_phi (rad) [2] e_theta (rad) [3] e_psi (rad)
Inport 2	gains	Double	12	Gains [0, 1, 2] Kp,i,d_z [3, 4, 5] Kp,i,d_phi [6, 7, 8] Kp,i,d_theta [9, 10,11] Kp,i,d_psi
Output 1	y	Double	4	Controller outputs [0] Vertical Thrust (N) [1] Rolling moment (Nm) [2] Pitching moment (Nm) [3] Yawing moment (Nm)

Now, the model is ready for passing into Gene-Auto.

### Generated code

The generated code consists of two modules\* as shown in the following table.

**Table 8-5 – Generated modules**

Module	Code File	Header File	Description
PIDcontroller	PIDcontroller.c	PIDcontroller.h	Main module, do the calculation
PIDcontroller_types	-	PIDcontroller_types.h	Define the structure of I/O variable and state variable for PIDcontroller module

\* The two Gene-Auto common modules GACCommon and GATypes are not considered here.

In Gene-Auto, we can generate a block as a separated module by setting on the parameter “Treat as atomic unit” of that block (Figure 8-16). For the complex models, this will help keeping the code more clear.

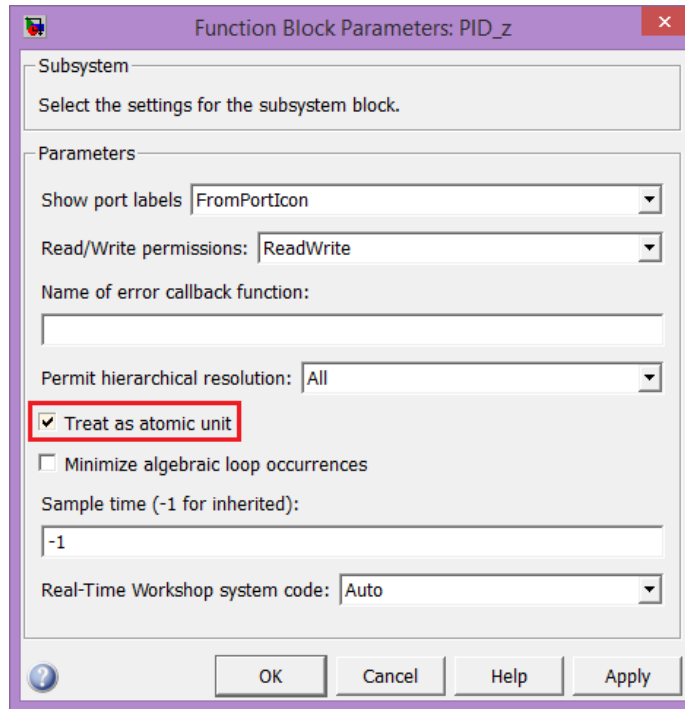


Figure 8-16 – Setting a block to be generated as separated module

Let make the PID\_z, PID\_roll, PID\_pitch and PID\_yaw blocks (in short: PID\_<channel> blocks) as “atomic units” and see the code change.

In addition to the modules listed on Table 8-5, each “atomic unit” has a related module, named as the name of the related block (Table 8-6).

Table 8-6 – Additional modules

Module	Code File	Header File
PID_z	PID_z.c	PID_z.h
PID_roll	PID_roll.c	PID_roll.h
PID_pitch	PID_pitch.c	PID_pitch.h
PID_yaw	PID_yaw.c	PID_yaw.h

In the main module, four additional I/O variables are defined for these four additional modules (lines 18 to 21, modified version, Table 8-7). They are local variables.

Table 8-7 – Additional I/O variables

Original	Modified
16 /* Variable definitions */	16 /* Variable definitions */
17	17
18 GAUINT8 i1;	18 t_PID_z_io_PID_z_io;
	19 t_PID_phi_io_PID_phi_io;
	20 t_PID_theta_io_PID_theta_io;
	21 t_PID_psi_io_PID_psi_io;
	22 GAUINT8 i1;

The next two code segments in Table 8-8 and Table 8-9 show the part of code that generated for four PID\_<channel> blocks before and after setting them as “atomic units.”

By using the atomic units, code generated is simpler and clearer; however, the code size, local variable number... increase (Table 8-10). When the resources (MCU speed, flash memory size, RAM size) are limited, it should be considered carefully between two models—using or not using atomic units. In our case, the resources are plentiful<sup>†</sup>; both models can be used.

Table 8-8 – Code segment of the PID\_&lt;channel&gt; blocks (code file: PIDcontroller.c)

```

300 /* END Block: <SystemBlock: name=PIDcontroller>/<CombinatorialBlock: name=Demux1>
    */
301 /* START Block: <SystemBlock: name=PIDcontroller>/<SystemBlock:
    name=PID_phi>/<CombinatorialBlock: name=Divide> */
302 Divide_2 = Demux_1 * Demux1_4;
<...SOME CODE and COMMENTS ARE CLIPPED...>
326 add_u_u_prev_2 = Divide1_2 + u_prev_1;
329 Multiply_2 = dt * half_2 * add_u_u_prev_2;
332 add_u_u_prev_1 = Divide1_1 + u_prev_4;
335 Multiply_1 = dt * half_1 * add_u_u_prev_1;
338 add_u_u_prev_3 = Divide1_3 + u_prev_6;
341 Multiply_3 = dt * half_3 * add_u_u_prev_3;
344 add_u_u_prev_4 = Divide1_4 + u_prev_8;
347 Multiply_4 = dt * half_4 * add_u_u_prev_4;
350 Add_2 = Multiply_2 + y_prev_2;
353 Add_1 = Multiply_1 + y_prev_1;
356 Add_3 = Multiply_3 + y_prev_3;
359 Add_4 = Multiply_4 + y_prev_4;
362 Divide2_2 = Divide_2 * Demux1_6;
365 Divide2_1 = Divide_3 * Demux1_12;
368 Divide2_3 = Divide_5 * Demux1_9;
371 Divide2_4 = Divide_7 * Demux1_3;
374 Subtract1_2 = Divide2_2 - u_prev_2;
377 Divide_1 = Subtract1_2 / dt;
380 y_phi = Divide_2 + Add_2 + Divide_1;
383 Subtract1_1 = Divide2_1 - u_prev_3;
386 Divide_4 = Subtract1_1 / dt;

```

<sup>†</sup> Code size of the complete multitasking system (without flight control system) is about 46.5 KB, 9% of total flash size (512 KB); the worse observable CPU utilization (WCET) of the main task without the flight control system is 3% (from observation of 4000 periods).



```

389 y_psi = Divide_3 + Add_1 + Divide_4;
392 Subtract1_3 = Divide2_3 - u_prev_5;
395 Divide_6 = Subtract1_3 / dt;
398 y_theta = Divide_5 + Add_3 + Divide_6;
401 Subtract1_4 = Divide2_4 - u_prev_7;
404 Divide_8 = Subtract1_4 / dt;
407 y_z = Divide_7 + Add_4 + Divide_8;
409 /* START Block: <SystemBlock: name=PIDcontroller>/<CombinatorialBlock: name=Mux> */

```

Table 8-9 – Code segment of the PID\_&lt;channel&gt; blocks (treated as atomic units)

Code file: PIDController.c
<pre> 136 /* END Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;CombinatorialBlock: name=Demux1&gt;     */ 137 /* START Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_z&gt; */ 138 _PID_z_io.dt = dt; 139 _PID_z_io.u = Demux_2; 140 _PID_z_io.Kp = Demux1_2; 141 _PID_z_io.Ki = Demux1_1; 142 _PID_z_io.Kd = Demux1_3; 143 PID_z_compute(&amp;_PID_z_io, _state_); 144 y_z = _PID_z_io.y; 145 /* END Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_z&gt; */ 146 /* START Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_phi&gt; */ 147 _PID_phi_io.dt = dt; 148 _PID_phi_io.u = Demux_1; 149 _PID_phi_io.Kp = Demux1_4; 150 _PID_phi_io.Ki = Demux1_5; 151 _PID_phi_io.Kd = Demux1_6; 152 PID_phi_compute(&amp;_PID_phi_io, _state_); 153 y_phi = _PID_phi_io.y; 154 /* END Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_phi&gt; */ 155 /* START Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_theta&gt; */ 156 _PID_theta_io.dt = dt; 157 _PID_theta_io.u = Demux_3; 158 _PID_theta_io.Kp = Demux1_7; 159 _PID_theta_io.Ki = Demux1_8; 160 _PID_theta_io.Kd = Demux1_9; 161 PID_theta_compute(&amp;_PID_theta_io, _state_); 162 y_theta = _PID_theta_io.y; 163 /* END Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_theta&gt; */ 164 /* START Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_psi&gt; */ 165 _PID_psi_io.dt = dt; 166 _PID_psi_io.u = Demux_4; 167 _PID_psi_io.Kp = Demux1_10; 168 _PID_psi_io.Ki = Demux1_11; 169 _PID_psi_io.Kd = Demux1_12; 170 PID_psi_compute(&amp;_PID_psi_io, _state_); 171 y_psi = _PID_psi_io.y; 172 /* END Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;SystemBlock: name=PID_psi&gt; */ 173 /* START Block: &lt;SystemBlock: name=PIDcontroller&gt;/&lt;CombinatorialBlock: name=Mux&gt; */ </pre>
Code file: PID_phi.c
<pre> 030 void PID_phi_compute(t_PID_phi_io *_io, t_PIDcontroller_state *_state) { &lt;...SOME CODE and COMMENTS ARE CLIPPED...&gt; 086 Divide_4 = u * Kp; 095 Divide1 = Divide_4 * Ki; 098 add_u_u_prev = Divide1 + u_prev_3; 101 Multiply = dt * half * add_u_u_prev; 104 Add = Multiply + y_prev; 110 Divide2 = Divide_4 * Kd; 113 Subtract1 = Divide2 - u_prev_4; </pre>

```

116 Divide_3 = Subtract1 / dt;
119 Sum = Divide_4 + Add + Divide_3;
122 _io->y = Sum;
133 }

```

Table 8-10 – Gene-Auto generated code metrics

	Using Atomic Units	Not Using Atomic Units
Generated functions	11	3
Local variables	103	75
Statements	163	107
Comment lines	541	305
Code lines	809	525
Code size (KB)	57.37	42.97

### 8.4.2.3 Code integration

Following the steps as in Part 8.2.3, the code integration is done on the main file (**Error! Reference source not found.**):

- **Import the root module generated by Gene-Auto**

Copy the generated folder <PIDcotroller\_ga> into the root folder of the project's code. Import the root module:

```
#include "PIDcontroller_ga/PIDcontroller.h"
```

- **Create variable instances for the IO and State variables**

```

/*
 * Variables for Gene-Auto
 */
// IO structure for the PIDcontroller_mod2 block
t_PIDcontroller_io io;
// State structure for the PIDcontroller_mod2 block
t_PIDcontroller_state state;

```

- **Call the `init(..)` function to initialize the (sub)system that was generated with Gene-Auto**

```

void setup ()
{
    // Init code for peripherals (I2C devices, Serial interfaces...)
    // ...
    /* Initialise the system */
    PIDcontroller_init(&state);
    // Creating mutexes
    // ...
    // End of code.
}

```

- **Implement the read-compute-write cycle**

Firstly, we need to create the read/write methods to load/get data into/from the computation step.

```
/*
 * IO methods for Gene-Auto code
 */
/* Reads the input data */
void do_read(const float time, const float *eulerAngles,
             const float altitude, const float *orders,
             const float *inGains, GAREAL *t, GAREAL *e, GAREAL *gains)
{
    *t = time;

    e[0] = (orders[0] - altitude)*M_PI/90;
    e[1] = (orders[1] - eulerAngles[0])*M_PI/90;
    e[2] = (orders[2] - eulerAngles[1])*M_PI/90;
    e[2] = (orders[3] - eulerAngles[2])*M_PI/90;

    for (i = 0, i<12, i++)
        gains[i] = inGains[i];
}

/* Write output data */
void do_write(GAREAL y, float *controlSignals) {
    for (i=0,i<4,i++)
        controlSignals[i] = y[i];
}
```

Finally, implement the read-compute-write cycle into the related task:

```
/*
 * Task definitions
 */
// TASK 1 ////////////////////////////////////////
void vTaskCalculateQuaternionsAndControlMotors(void *pvParameters)
{
    // Define & init local variables: altitude, gains, orders, eulerAn-
    // gles,
    //                               controlSignal, pwmSignal
    // ...
    xLastWakeTime_Task1 = xTaskGetTickCount();
    while (1)
    {
        AHRSgetQ(q);           // Reading sensors and calculate quaternions
        Q2E(q, eulerAngles); // Convert quaternions into Euler angles

        // Reading memory module: MDD_altitude -> altitude
        // Reading memory module: MDD_gains    -> gains
        // Reading memory module: MDD_orders   -> orders

        /* PID Flight Controller Integration
         * -----
         * Input  : eulerAngles, altitude, orders, gains
         * Output : controlSignal
         */
    }
}
```

```
* read-compute-write cycle
*/
time = (real) millis()*1000; // get the current time in seconds
do_read(time, eulerAngles, altitude, orders, gains, &(io.t),
        &(io.e), &(io.gains));
PIDcontroller_compute(&io, &state);
do_write(&(io.y), controlSignals);
/*
* End of integration
*/
calculateMotorData(controlSignal, pwmSignal); // Calculating PWM
signals
controlMotors(motorData); // Send PWM singals to motors

// Writing memory module: MDD_eulerAngles <- eulerAngles

vTaskDelayUntil( &xLastWakeTime_Task1, RTC1 ); // Wait for the next
period
}
}
```

Now, the implementation is complete.



# Conclusion

## 8.5 Achievements

There are some achievements in this study:

- Many technical issues have been solved: the hardware setup has been carried out, drivers for all peripherals has been developed.
- The gains in the Madgwick data fusion algorithm (the IMU/MARG filter) has been tuned to get a better altitude measurement.
- A complete multitasking system has been developed.
- A Ground Station has been developed.
- A communication protocol for communicating between the quadricopter and the ground station has been proposed and implemented.
- The PID flight control system designed in MATLAB/Simulink can be translated into C code, ready for integrating into the multitasking system.
- Documentation and technical reports are committed with the works, ensuring that the works can be reused as well as updated easily.

However, there are some later works must be completed.

## 8.6 Later Works

- Optimized gains for the Madgwick data fusion algorithm should be estimated. Other data fusion algorithms should be considered.
- The noise level of the barometer is very high. Thus, a filter should be considered to apply for this sensor, to smooth out the measurement.
- Tests should be carried out to validate the multitasking system.
- Currently, the drivers for the I2C devices and the multitasking system were developed separately. Thus, code optimization can be done to reduce the code size (for examples, there are some variables that can be merged together).
- Time-out check has only been implemented in the Ground Station's receiver. In the GPS acquiring task and the receiver of the communicating

task, time-out check should also be implemented to avoid task blocking due to lack of data.

- The missing part of the flight control system (the limiter, the signal converter that convert the controller signal to relative PWM signals...) should be implemented and integrated into the system.
- Finally, the path generator—a complex Simulink model—will be a big challenge to pass into C code and to integrate into the system.

— ★ —

# Bibliography

- [1] Lauszus, "A practical approach to Kalman filter and how to implement it," TKJ Electronics, September 2012. [Online]. Available: <http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/>.
- [2] S. O. Madgwick, "An efficient orientation filter for inertial and inertial/magnetic sensor arrays," 2010.
- [3] R. Mahony, T. Hamel and J.-M. Pfimlin, "Complementary filter design on the special orthogonal group  $SO(3)$ ," in *44th IEEE Conference on Decision and Control and the European Control Conference*, Seville, 2005.
- [4] G. Baldwin, R. Mahony, J. Trumpf, T. Hamel and T. Chevion, "Complementary filter design on the Special Euclidean group  $SE(3)$ ," 2007.
- [5] W. Premerlani and P. Bizard, "Direction Cosine Matrix IMU: Theory," 2009.
- [6] F. Cottet and E. Grolleau, *Système Temps Réel de Contrôle-Commande, Conception et Implémentation*, Paris: Dunod, 2005.
- [7] NXP Semiconductors, *I2C-bus specification and user manual*, 2012.
- [8] LeafLabs, "LeafLabs Maple Documentation," 15 January 2014. [Online]. Available: <http://leafflabs.com/docs/>.