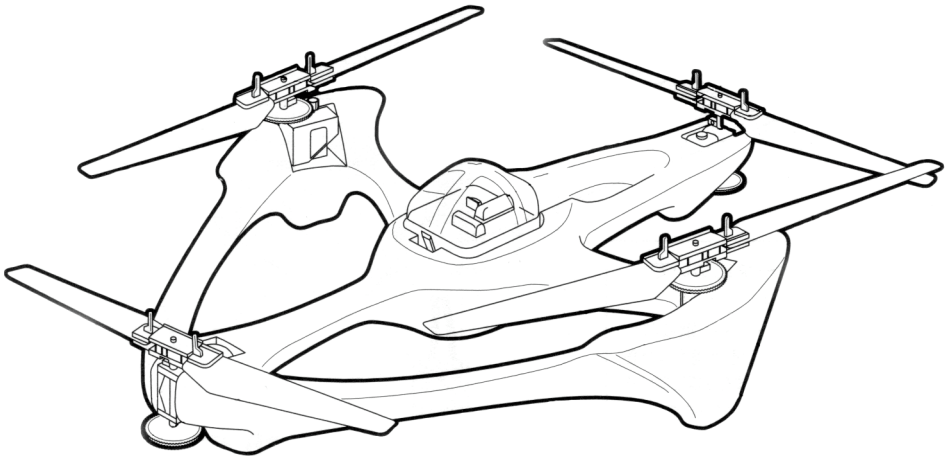
AUTONOMOUS SYSTEMS LAB

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE



COMPUTER-BASED CONTROL SYSTEM FOR A MODEL HELICOPTER

Microengineering Department
Semester Project 2002 Report

Professor : Roland Siegwart
Assistants : Daniel Burnier and Jean-Christophe Zufferey

TABLE OF CONTENTS

PROJECT'S GOAL

Introduction

Compared to land vehicles, helicopters are very interesting because they can move freely in the 3 dimensions, and with fewer constraints than airplanes: for example, a helicopter can hover stationary.
Moving in 3 dimensions greatly helps autonomous robots i.e. they are less affected by the relief of the ground, they are able to avoid obstacles more easily, they have a better perception of their environment and so on.

Mobile helicopter robots are a very challenging concept, but unfortunately, unlike airplanes, helicopters are naturally unstable in the air. This well-known quote from Harry Reasoner tells everything:

"*The thing is, helicopters are different from planes. An airplane by its very nature wants to fly and, if not interfered with too strongly by unusual events or by a deliberately incompetent pilot, it will fly. A helicopter does not want to fly. It is maintained in the air by a variety of forces and controls working in opposition to each other and, if there is any disturbance in this delicate balance, the helicopter stops flying; immediately and disastrously. There is no such thing as a gliding helicopter.*"

The first big step in building an autonomous helicopter is obviously to develop a flying machine that is able to hover without human intervention.
Hence, the goal of this project is to build a computer-based control system around a commercially available indoor / outdoor model helicopter: the Keyence Engager GSIII. The resulting system will be used as a starting point for future autonomous helicopter projects at the ASL.

Objectives for this project

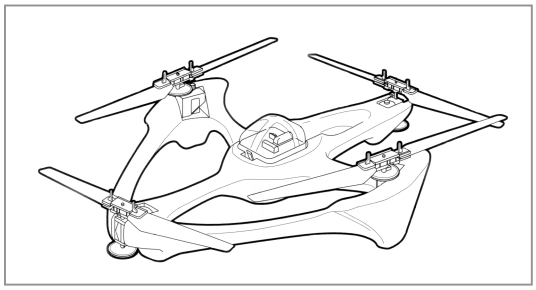
- Evaluate the Engager GSIII (behavior when flying, maximal payload, etc…),
- Find which sensors are required for autonomous hovering,
- Choose the sensors and microcontroller to use,
- Replace the radio link with a RS232 wired link,
- Design and build a PCB to host the sensors, the microcontrollers and the motor amplifiers,
- Develop PC software to read the sensor values from the PCB and set the motor speeds.

KEYENCE ENGAGER GSIII MODEL HELICOPTER

The Engager GSIII is a commercially available small radio-controlled helicopter produced by Keyence[1]. Since it is electrically powered and can be used indoor and outdoor, it is a well-suited model helicopter for this project.

Overview

The Engager GSIII is a four-rotor heads model helicopter that uses electric motors. It is small, light, and relatively quiet – well, compared to model helicopters with fuel engines. The rotor heads do not have any pitch or collective control. With standard Ni-Cd batteries, the Engager GSIII is able to fly up to a few minutes.
An on-board PCB with angular velocity sensors and a microcontroller drives the motors and helps the pilot controlling the machine.



Note: refer to annex E for more information regarding the PCB itself.

First impressions

The Engager GSIII definitely looks like a cheap toy: fragile polystyrene fuselage and blades, simple motor gears, bad motor fixations… Fortunately, once you know it better, it appears to have a very smart design: it's quite light, built with very few pieces, and replacing blades is easy.

Although I do have some experience in piloting model helicopters, controlling the Engager GSIII was far from easy. Since it is very light, this helicopter has very little inertia, and is therefore quite unstable and sensitive to turbulences.
After a few flights, I was however able to practice hovering, translation and rotation. Intense concentration was still required: since the blades are made of polystyrene, they bent a lot when producing the thrust necessary to compensate for the helicopter's weight. When bent, their lift force is not very good, and if the helicopter tilts too much, it will immediately stall.

Measurements have shown that the Engager GSIII may sink up to 19A at 8V DC when the four motors are actively running.
The helicopter weights approximately 350g and its maximal payload is about 90g (including the weight of the power cables and of the protection cross[2]).

[1] Web page: http://www.keyence.co.jp/hobby/english/saucer.html
[2] We fixed a carbon-built cross under the fuselage to protect the blades.

Dynamic behavior

A four-rotors structure presents several advantages:

- increased payload: since the thrust of a rotary wing is proportional to the square root of its area, several rotor disks mean more thrust, and consequently, more payload. However, it's true that multiple rotors also mean losses due to interaction of the air underneath.
- no need for adjustable pitch / collective rotor heads to control movement: this simplifies a lot the helicopter's mechanic and reduces its weight.
- a larger time-constant: according to a draft whitepaper of the HoverBot project [1], "*the distributed weight of the 4 rotor heads increases the moment of inertial and thereby the time-constant.*" Increasing the time-constant is a very important factor for model helicopters, because it's their small size that makes them so difficult to stabilize.

We can also point out two disadvantages:

- four rotors generate intense turbulences, especially when the helicopter is close to the ground – consequently, hovering at low altitude is very difficult. The only advantage of flying low is less power required to hover because of the "in ground effect"[1].
- controlling the rotor thrust by adjusting the motor power is not an efficient means (motors do not respond quickly), while adjusting the rotor blades pitch has an almost-immediate effect.

Because of the induced moments[2] generated by the rotors, it is necessary that 2 opposite rotors turn clockwise while the 2 others turn counter-clockwise. If all rotors generate the same amount of induce moment, the sum of all induced moments is null, which prevents any horizontal rotation.

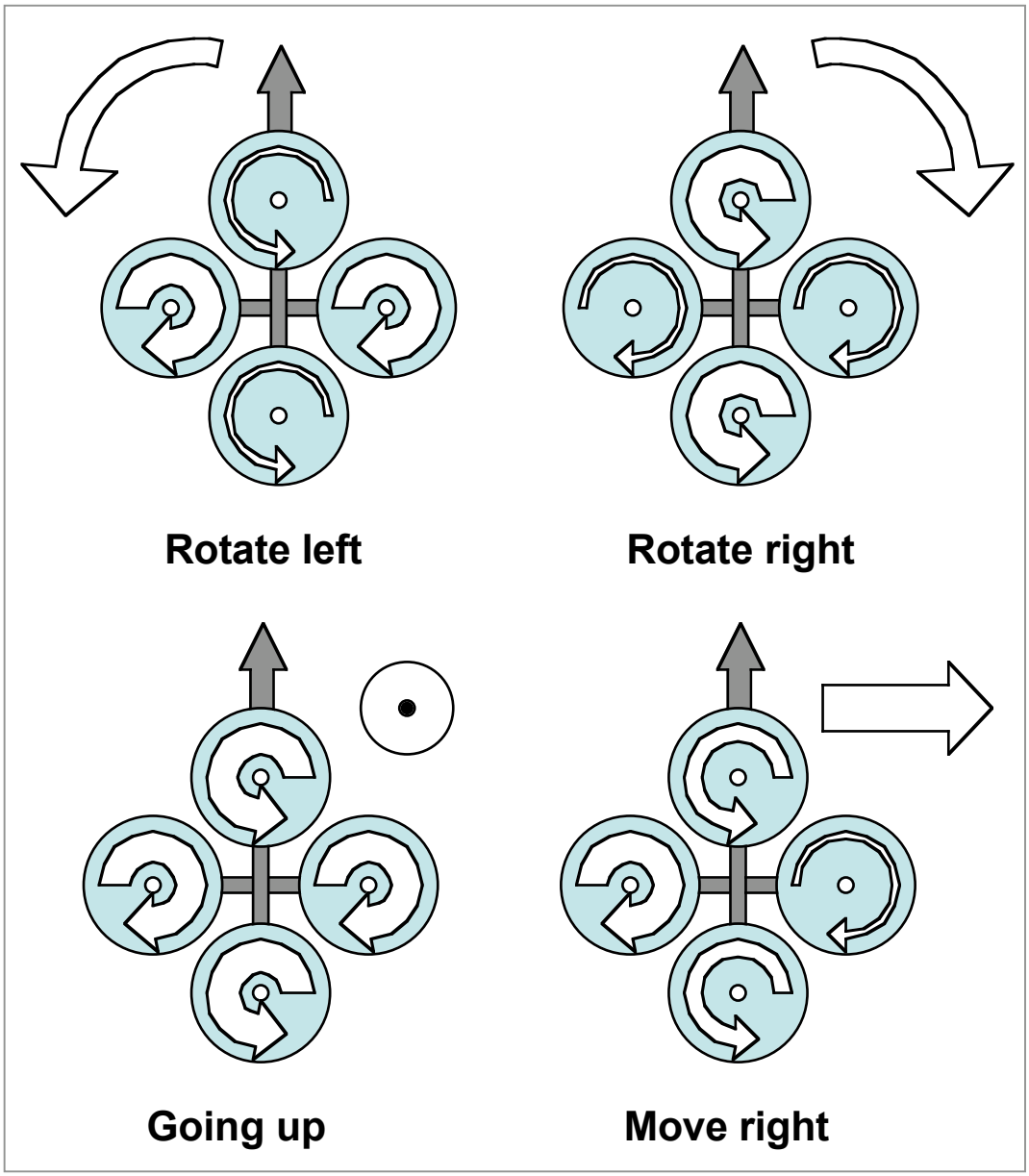
Since the four electric motors are connected to fixed-pitch rotor heads, helicopter movement is achieved by adjusting motor speeds:

- Collectively increasing or decreasing the power to all 4 motors easily controls up / down motion. Since all rotors turn at the same speed, there is no horizontal rotation.
- Horizontal rotation is less intuitive and is created by making the total induced moment non-null: if the counter-clockwise rotors increase their rotational speed, the resultant induced moment will cause the Engager GSIII to turn counter-clockwise. At the same time, clockwise rotors should decrease rotational speed so that the global lift force remains the same and altitude is maintained. In this case, all rotor thrusts are strictly vertical, therefore the helicopter remains horizontal and no translation occurs.
- To translate laterally or longitudinally, the rotational speed of the rotor in the wanted direction must be decreased. This will roll the helicopter and it will start gliding in the desired direction. Simultaneously, the opposite rotor should turn faster so that the total induced moment of these 2 rotors does not change, and the helicopter does not rotate. At this time, the helicopter being no more horizontal, drag forces appear, lift forces are reduced and the helicopter starts losing altitude. To compensate, it's necessary to collectively increase the power of all 4 motors.

[1] In ground effect occurs when hovering less than one rotor diameter above the ground: the airflow interferes with the ground and the lift force increases.

[2] The reactive force of the air against the rotor causes a reactive moment, in the direction opposite to the rotation of the rotor. Its amount essentially depends on the rotor rotational speed.

These graphics show the rotation speed needed on each motor to achieve some typical movements (a bigger arrow means a faster rotation speed):



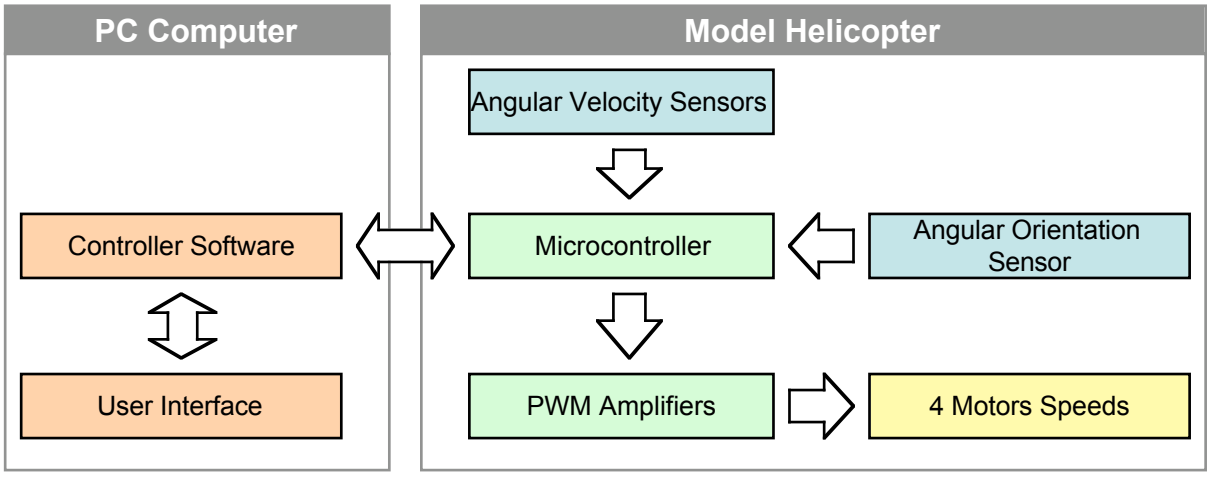
CONTROL SYSTEM DESIGN

This section briefly resumes the steps we accomplished during the system's design.

Control system overview

First, we needed to choose between an on-board controller and a remote controller on a PC computer. Working on a remote PC controller presents several advantages, e.g. reduced development time, easier debugging, the possibility to draw graphics with the sensors values, etc. On the other hand, it increases latency and may create timing inaccuracies.
We eventually decided to build a PC-based controller system to ease development of future controllers – when the controller code is done, it may be rewritten for an on-board microcontroller anyway.

Then, we defined the requested information to stabilize the helicopter, whereas we did not care about translation at this time: the goal is to maintain the engine horizontal. Position control (and true hovering) is left to a future improved version of the control system.
Stabilizing the helicopter requires information about its angular orientation and angular velocity along the 3 axes.



As said earlier, model helicopters are especially unstable because of their small time-constant, as they are small, light, and they lack damping when flying. Consequently, our controller needs a high bandwidth: we defined a targeted frequency of 50Hz[1].

To minimize the development time, we decided to re-use partially the original PCB of the Engager GSIII: we would keep the power unit (PWM motor drivers, 5V and 10V DC power sources), while replacing the control unit with a new one (refer to annex E for more information regarding the original PCB). This approach is better than trying to reverse-engineering the control unit and "hack" it: eventually, we have a PCB that we perfectly know and that can be easily upgraded, or even repaired if anything goes wrong.

[1] This value is a correlation between PC timing precision, sensor bandwidths… and ASL people experience!

Microcontroller and sensors choice

Re-designing the control unit is essentially about choosing the right set of sensors and the best microcontroller:

To measure angular velocity, we chose the Murata Gyrostar ENC-05 sensors (also called "gyroscopes"), which output an analog signal proportional to the rotational speed on its main axis. Therefore, one sensor is needed per axis.
To measure the angular orientation (or "tilting"), we selected the TCM2-50 made by Precision Navigation, Inc. It outputs the compass, roll and pitch values through a standard RS232 connection.

Note: refer to the further sections "Angular orientation measurement" and "Angular velocity measurement" for detailed information regarding these sensors.

The choice of the microcontroller was more difficult since it has to meet several minimal requirements:

- 4 A/D channels to connect the 3 gyroscopes and possibly an altitude sensor,
- 4 PWM outputs to drive the PWM amplifiers that power the motors,
- 2 USARTs for TCM2 and PC connections,
- 1 SPI to connect I^2C extension modules.

We were able to find only two microcontrollers that fit our special needs, but unfortunately none of them was available for sale at that time:

Microcontroller name	A/D channels	PWM outputs	USART units	SPI units	Clock (MHz)	Availability
Atmel ATmega64[1]	8	6+2	2	1	0-16	Q3 2002
Microchip PIC18F6620[2]	12	5	2	1	40	Q4-2002

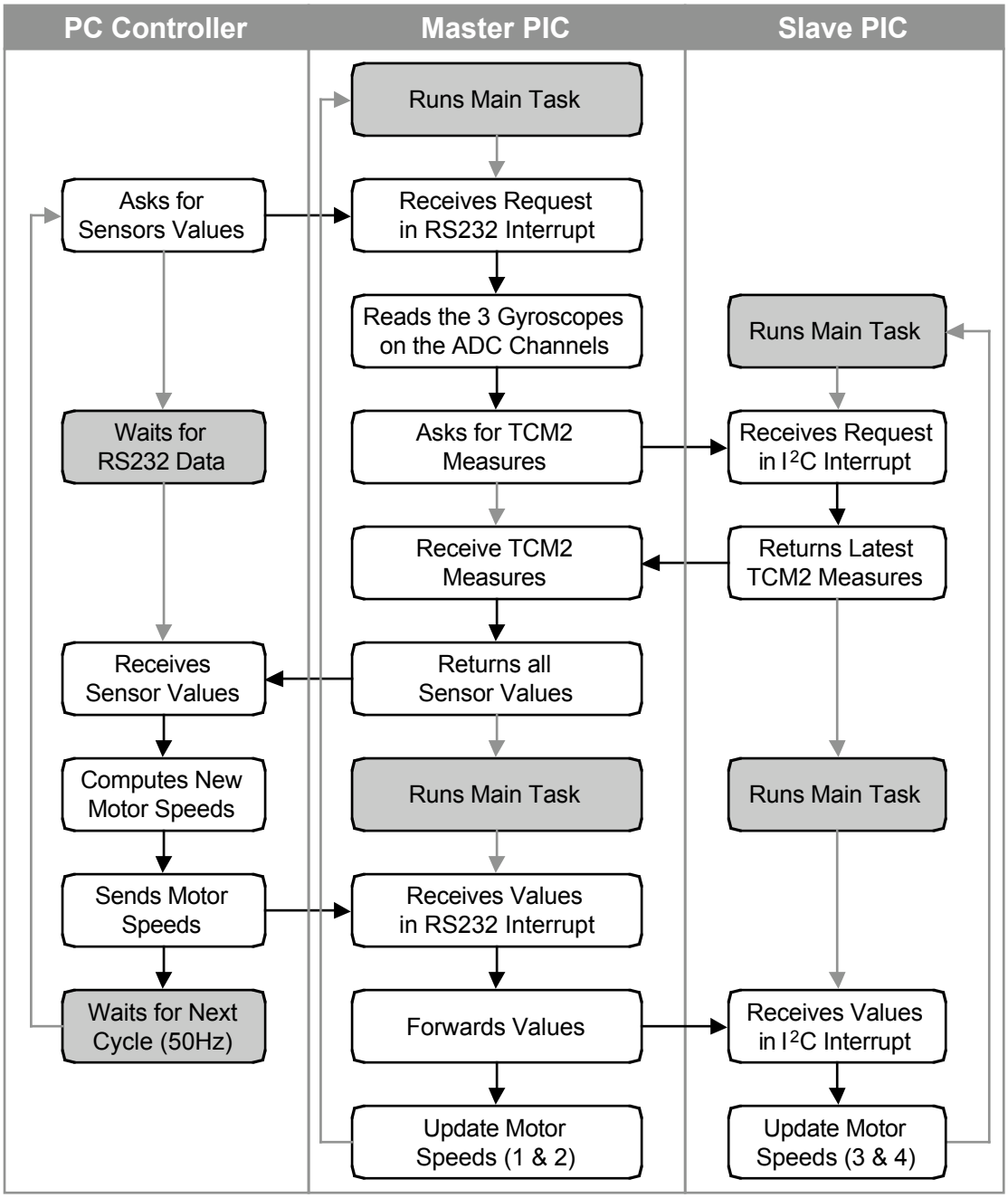
Other possibilities have been evaluated like using a K-Team Kameleon board[3] or an Ubicom SX microcontroller[4]. The Kameleon allowed only minimal control over the generated PWM signals while the SX microcontroller, which implements PWMs, USARTs and SPIs as software modules (Virtual Peripherals), did not have enough processing power to handle all the requested tasks.

We eventually decided to use Microchip PICs 16F876: they run up to 20MHz and have 2 10bits PWM outputs, 5 10bits A/D channels, 1 USART and 1 SPI. Furthermore, they are extensively used at the ASL and the development kits are already available.
By using two PIC 16F876 that communicate on the I^2C bus, it is possible to meet the initial requirements.

[1] Web page: http://www.atmel.com/atmel/products/prod199.htm
[2] Web: http://www.microchip.com/1010/pline/picmicro/category/embctrl/32kbytes/devices/18f6620/index.htm
[3] Web page: http://www.k-team.com/boards/kameleon/index.html
[4] Web page: http://www.ubicom.com/products/sx/sx.html

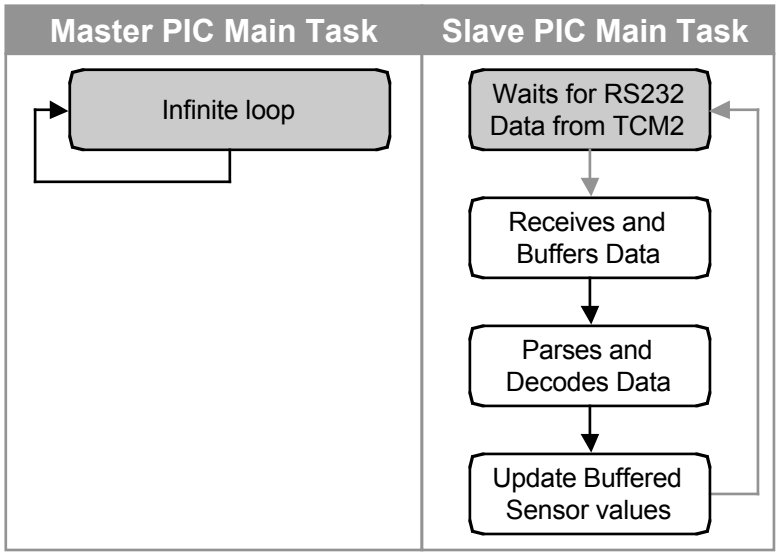
System operation

One PIC, defined as the "Master PIC", handles communication with the PC, control 2 motors and reads the gyroscopes values through its A/D converters. It acts as a master on the I^2C bus to which the second PIC is connected. This "Slave PIC" receives the measures from the TCM2, forwards them to the Master PIC, and controls the 2 other motors.



Note: annex B outlines the PIC's communication protocol, while the PIC's source code is available as an annex to this report.

Because the TCM2 has a latency way bigger than the controller's cycle time (see section "Angular position measurements"), it has to run asynchronously in "continuous sampling" mode. The TCM2 actually sends its measures at 40Hz to the Slave PIC that buffers the latest values. These are sent to the Master PIC when then PC controller requests the sensor values.



PCB design notes

- Each PIC's hardware interrupt pin is connected to an IO pin of the other PIC to allow cross-interrupts generations for synchronization purposes if required.
- Each PIC has its own programming connector and reset button, along with a LED that can be used for any purpose e.g. informing the user that the PIC is running correctly.
- Since each PIC has its own crystal, they might not run at the exact same frequency.
- To allow future extensions, remaining pins of both PICs are wired to side connectors on the PCB, and the I^2C bus leads to an extension connector.
- To remove electrical noise, decoupling capacitors are placed closed to each component's supply voltage track.

Note: PCB complete schematic is available as an annex to this report.

ANGULAR ORIENTATION MEASUREMENTS

Keeping the helicopter horizontal requires knowledge of its inclination (or tilt) and orientation. The TCM2 sensor fits these needs because it can measure its absolute compass, roll and pitch angles.

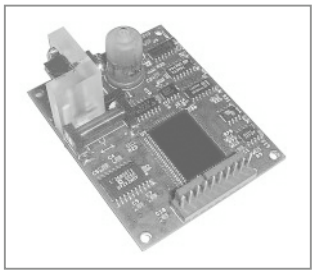
Angular orientation sensors

There are two kinds of angular orientation sensors: the first one measures angular velocities and magnetic fields, then integrates these values to compute an absolute orientation[1]; the second one directly measures absolute angles using "mechanical" compass and tilt sensors.

The second type of sensors is more interesting for our project because they cannot drift under normal conditions. In our circumstances, drifting sensor values would certainly cause a crash.

Precision Navigation's TCM2 compass and tilt sensor

We eventually decided to use an absolute orientation sensor that could be easily connected to the microcontrollers. We found the TCM2 from Precision Navigation, Inc (PNI)[2], which was the most extensive sensor with an acceptable weight. This sensor seems to be widely used in the automobile industry, and it has been successfully used in a similar project [2].



The TCM2 is essentially a compass sensor, based on a three-axis magnetometer augmented with an electrolytic two-axis tilt sensor (more information later). This tilt sensor acts as an inclinometer and allows the TCM2 to correct its compass measurement depending on its inclination. The sensors are mounted on an electronic board with a microprocessor that computes the compass, roll and pitch angles. The board has both RS232 and analog output (for heading only).
The TCM2 can output any combination of compass, roll, pitch, magnetometer and temperature measurements as ASCII data.

The TCM2 exists in three versions: the TCM2-30, the TCM2-50 and the TCM2-80. The major difference is the range of the pitch and roll measurements: ±30°, ±50° or ±80°. This range influences the global accuracy of the sensor: a wider range means a lower accuracy. For our project, we have chosen the TCM2-50 version, which is the best compromise.

The TCM2 also offers iron distortion correction systems and optional damping for compass measurements (through a configurable IRR filter).

[1] For example sensors of this type, check InterSense's web site: http://www.isense.com
[2] Web site: http://www.pnicorp.com

These are the main characteristics of the TCM2-50:

Heading accuracy	1.5° RMS
Heading resolution	0.1°
Tilt accuracy	±0.4°
Tilt resolution	0.3°
Maximal sampling rate	30Hz in "normal" mode 40Hz in "fast" mode (±0.3° noise added)
RS232 connection	up to 38400bps
Supply voltage	+5V DC (regulated) or 6-18V DC (unregulated)
Current	7 to 20mA

The TCM2's main disadvantages are:
- its weight (around 50g),
- the format it uses to send the data – since it sends pure ASCII instead of directly usable binary data, these ASCII strings have to be converted to binary numbers first,
- its latency[1] which is around 76ms in "normal" mode and 56ms in "fast"mode (on a 38400bps RS232 link).

Note: despite its high latency, the TCM2 is still able to work at 40Hz because computation is overlapped with measurement of the sensors.

How PNI magneto-inductive compass work

PNI's patented magneto-inductive compasses evaluate their angle to the magnetic north by measuring the earth's magnetic field vector. Three magneto-inductive sensors perpendicular to each other are used to compute this vector.

A magneto-inductive sensor is made of a single solenoidal winding, whose inductance change with different applied magnetic field strengths. These sensors are well suited for portable electronic navigation and measurement systems because they provide highly accurate magnetic field information, consume little power and are less expensive than alternative technologies. [11]

How electrolytic tilt sensors work

Electrolytic tilt sensors are made with a glass (or ceramic) envelope, partially filled with a conductive fluid. Vertical platinum electrodes go through the envelope, into the fluid.
The fluid moves due to tilting, under the influence of earth's gravity or by acceleration. When the sensor is at zero position, the electrical impedance of the fluid from the center electrode to the side electrodes is equal. Tilting the sensor disturbs this balanced condition and the impedance changes in proportion to the angle of tilt. Since the center of gravity of the volume

[1] For a detailed explanation, refer to the "Output response" paragraph in [11].

of fluid remains fixed, and the electrodes move with the envelope, the sensor acts therefore as liquid potentiometer.

An AC excitation voltage is applied between the side electrodes. When the sensor is horizontal, the voltage at the center electrode is 50% of the excitation voltage. Tilting the sensor will cause a proportional voltage variation around the 50% point.

Note: applying DC voltage to the electrodes would damage the sensor because of the electroplating action that would occur.

Tilt sensors electrical characteristics normally depend on temperature, therefore the need for temperature compensation. However, none is necessary in the TCM2 according to the documentation [11] - the temperature sensor was left for backward compatibility with the first-generation TCM1.

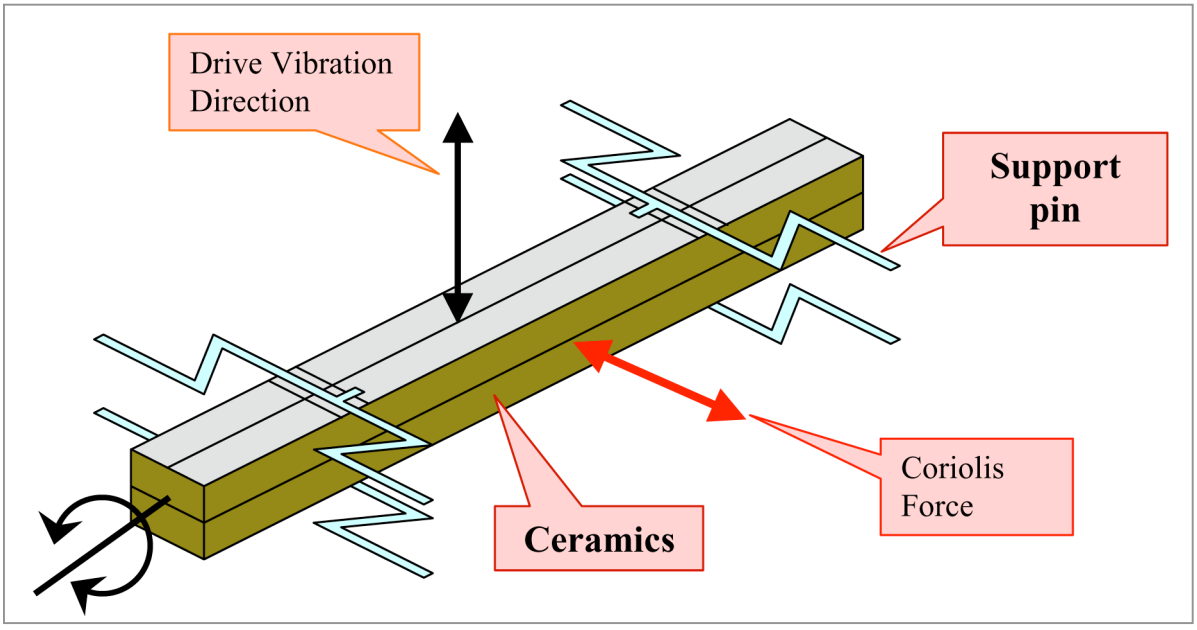
ANGULAR VELOCITY MEASUREMENTS

Because the TCM2 is not designed to evaluate angular speeds, we had to rely on specific sensors to achieve angular speed measurement: piezoelectric gyroscopes.

How piezoelectric gyroscopes work

Piezoelectric gyroscopes employ the principle that a Coriolis force results if an angular velocity is applied to a vibrating object. Murata's Gyroscope Application Guide [7] describes exactly how it works:

The piezoelectric ceramic vibrates at the resonant frequency in a vertical direction. When a rotational velocity is introduced into the system, the Coriolis force causes the ceramic to vibrate in the horizontal direction. This horizontal vibration causes the ceramic material to distort from the left to the right of the material. This distortion causes the piezoelectric material change the phase angle of the inputted voltage from the left side to the right side. […] The sensors on the top of the Bimorph material read the analog output voltages. These forces are in relation to the angular velocity only. The Coriolis effect delays the signal by 90 degrees at the full force.



Since each sensor can only measure angular velocity in one dimension (or around one axis), 3 sensors mounted perpendicularly to each other are required to measure reliably the rotation speed.

Murata's ENC-05 piezoelectric gyroscope

We ended up with Murata sensors because they have been used successfully in similar projects (Keyence Engager GSIII or [2]), and we simply did not find any better product.
Murata produces several types of piezoelectric gyroscopes depending on the application. We chose the ENC-05E because it is the best compromise between dimensions, weight, precision and maximal angular speed:

	ENC-05E	ENV-05D	ENC-03J
Maximal angular speed (°/s)	90	80	300
Dimensions (mm)	21.5 x 8.5 x 7.1	37.2 x 29.8 x 17.8	15.5 x 8.0 x 4.3
Weight (g)	2.7	50	1
Price (CHF per unit)	26[1]	115	35

Only the ENV-05D outputs a signal ready to be read by an A/D converter, since it includes internal circuitry to amplify and filter the signal. The ENC-05E and ENC-03J gyroscopes require dedicated amplifier circuits.

Note: the ENC-05E is available in two variants: A, with a resonance frequency of 25kHz, and B with a resonance frequency of 26.5kHz. When using several ENC-05E, A and B variants should be used to limit any possible perturbation.

ENC-05 output signal amplifiers

According to the ENC-05 datasheet, the sensor measures angular velocities up to 90°/s with a scaling factor of 1.11mv/°/s (±20%), while its output voltage (Vout) is centered around a reference voltage (Vref) - which is approximately 2.3V.
Therefore, Vout should vary with amplitude of ±100mV around Vref. However, experimental measures have shown variations up to ±600mV. Furthermore, the example ENC-05E amplifier circuit we were able to find [2] has gain of about 9 only – a bit small to fit a 0-5V A/D converter… Since we had no time to characterize the sensor precisely (to measure its "true" output range and scaling factor), we finally decided to amplify the signal with a gain of about 10 times - this allows a sensor output variation of ±250mV approximately.

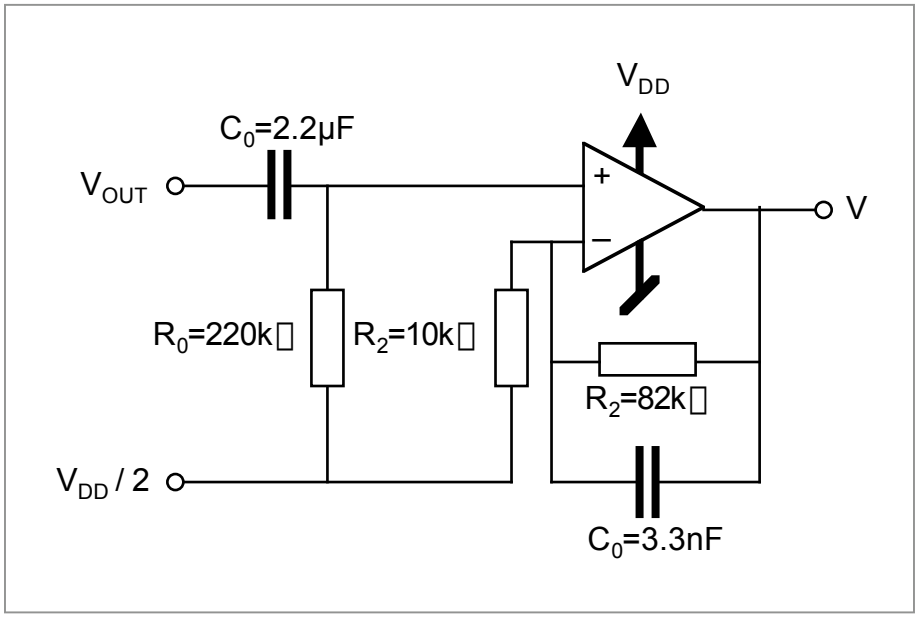
Since the ENC-05 outputs both Vout and Vref on separated pins, properly amplifying and filtering the output signal would have required a differential amplifier (to compute the voltage Vout – Vref), followed by an active 2nd order[2] Band-Pass filter to remove the possible output drift and the noise around the frequency response (about 25kHz). In the current context, such design is a "luxury" since no symmetrical power is available on the PCB (required for

[1] For undisclosed reasons, ENC-05E sensors are not available through Murata Switzerland and we had to bought them from K-Team (http://www.k-team.com).
[2] A 2nd order filter has a stronger slope and minimizes the phase shift.

differential amplifiers) and dozens of operational amplifiers, resistors and capacitors would have been necessary.

We found experimentally that Vref remained stable when the sensor was moved - according to the datasheet it should vary only with the temperature. It's also important to remember that 1) we do not need information about the very low angular speed since we already have it from the TCM2 sensor, and 2) since we won't likely sample the gyroscopes above 100Hz, we do not really need frequencies above 50Hz (Nyquist's law).

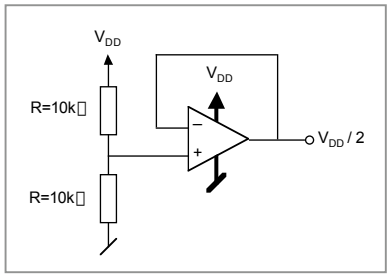
Considering these facts, we built an amplifying and filtering circuit inspired from the sample circuit proposed by Murata for its ENC-03J. [8]



This circuit achieves the 4 requested functions:

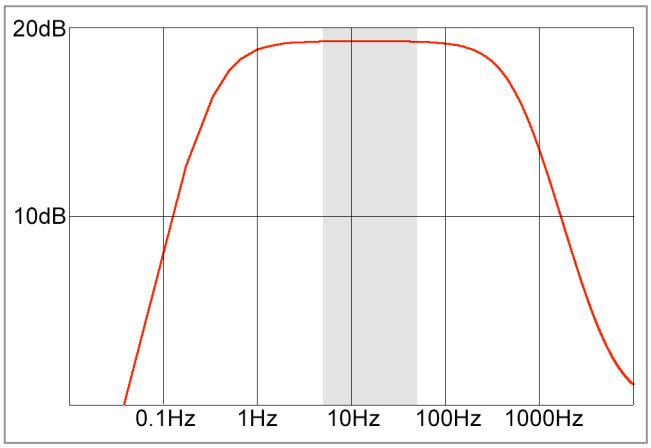
- The passive High-Pass filter has a cutoff frequency of 0.33Hz and removes the DC component of the signal (which is actually Vref), along with any eventual drift.
- The active Low-Pass filter has a cutoff frequency of 588Hz and removes the high-frequency noise - a cutoff frequency of more than 10 times the target top-frequency guaranties a pure signal.
- The active Low-Pass filter also amplifies the signal with a gain of 9.2.
- The amplified and filtered signal is centered on $V_{DD}/2$ in order to fit exactly the middle of the 0-V_{DD} A/D converter inputs, while remaining independent from any drift of V_{DD}.

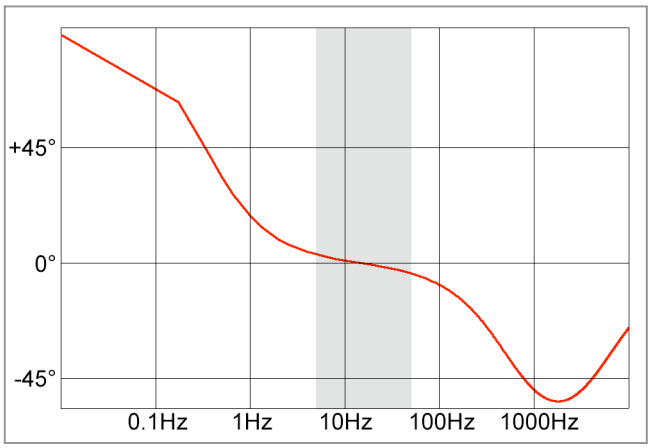
The $V_{DD}/2$ reference voltage is produced by a simple voltage divider, connected to a voltage follower OA that guarantees a minimal output resistance.



Operational amplifiers are LMC6582 one, because they perfectly fit our needs and are available in stock at the ASL. The LMC6582 is a rail-to-rail operational amplifier able to output voltage from 0.2V to 4.85V approximately. [9]

These two graphs show the theoretical frequency response of the amplifier to an AC signal, where the gray area figures the 5-50Hz band.

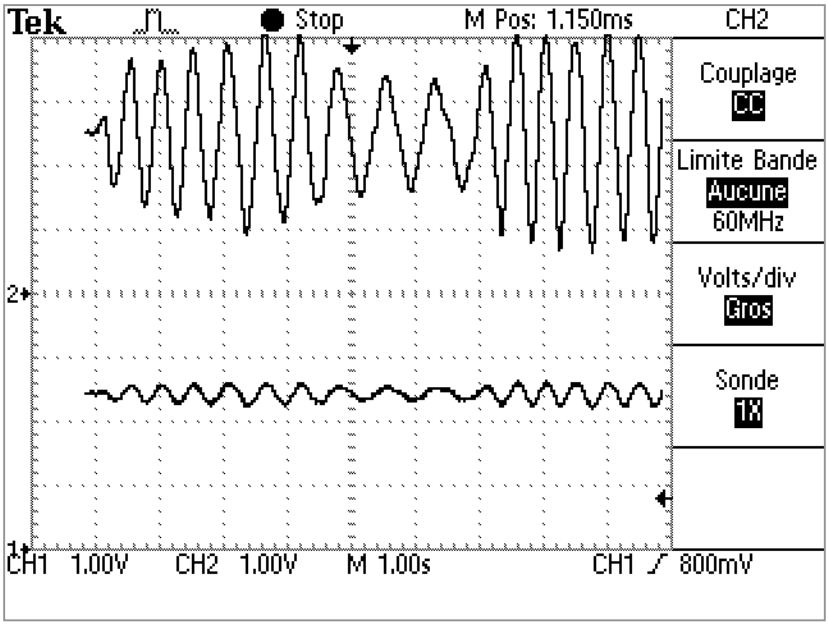




This circuit also spots some strong advantages:

- a very-low phase shift in the frequency band we are interested in,
- a very low output resistance (virtually zero)[1],
- a good input resistance[2] (virtually infinite for AC and around 220kΩ for DC),
- it does not inverse the signal,
- it requires few components.

This oscilloscope screen capture shows the action of the circuit on the final PCB. The top signal is the output of the circuit while the bottom signal is the raw output of the Vout sensor's pin.



[1] The PIC 16F876 tolerates a maximum load resistance of 10kΩ on it's A/D inputs. [10]
[2] The ENC-05E requires a minimal output load resistance of 50kΩ.

MOTOR SPEED CONTROL

Since we decided to keep the motor amplifiers from the original PCB, the problem was to find how to drive these amplifiers from the PICs.

PWM motor controllers

The electrical motors are driven through 4kHz Pulse Width Modulated (PWM) amplifiers (see annex E). PWM amplifiers are low-cost amplifiers based on transistors that are switched rapidly ON and OFF. When the transistor is ON, a current flow goes through the motor that makes it rotate; when the transistor is OFF, no current goes through the motor and it is stopped. Speed control is achieved by varying the ratio between ON and OFF times (also called "duty-cycle"). Finally, the higher the PWM frequency is, the smoother the rotation will be.

Note: it is not possible to use any PWM frequency, since the transistors have finite switch times (when changing their ON-OFF state).

Controlling the motor speeds

The PWM amplifiers are located on the lower PCB board and expect 0-10V 4kHz PWM input signals. However, we have found experimentally that 1) it is possible to feed the PWM amplifiers with PWM signals as low as 0.7V (or even from a TTL output), and 2) a 4kHz PWM is way from being optimal: a higher frequency PWM make the motors less noisy while the voltage on the motor pins is "smoother".

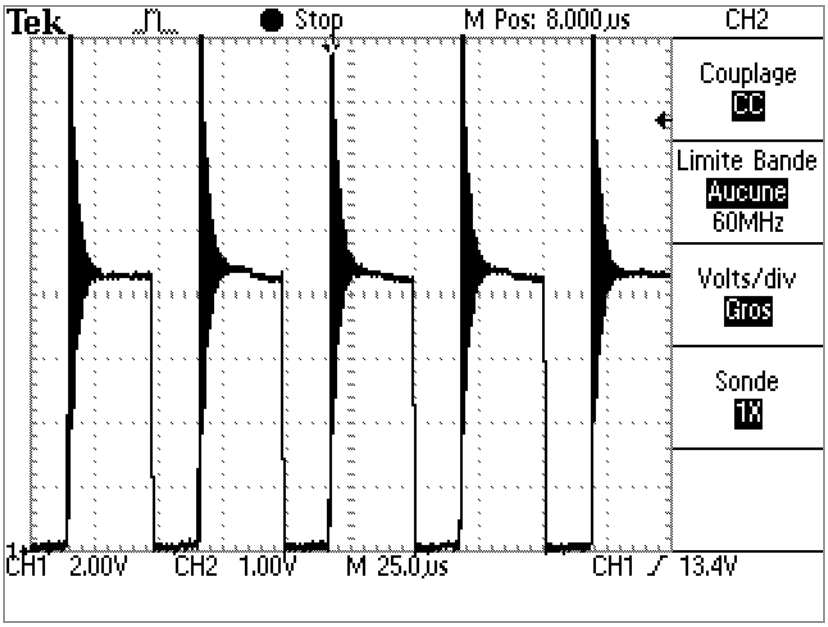
Since the PWM amplifiers have high-input resistances, we decided to drive them directly from the PIC's 0-5V PWM outputs[1]. Producing PWM signals on the PIC is all about finding a compromise between the frequency and the resolution of the duty cycle: the higher the frequency, the less the resolution.

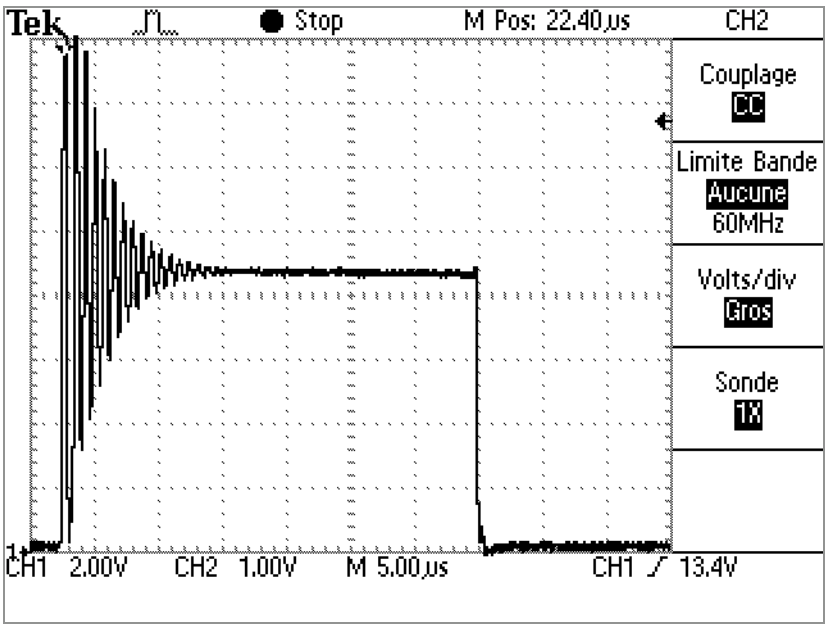
We decided to use an 8bits resolution with the maximal possible frequency i.e. 19531Hz. Our tests have shown that the PWM amplifiers were able to work fine with such signals.

Note: the motors start when the PWM duty cycles reach about 20 – maximal duty cycle value is 255.

[1] PIC's pins may source up to 25mA. [10]

These two oscilloscope screenshots show the voltage on the pin of one of the motors. As said earlier, we can clearly see that a 20kHz PWM is perfectly supported by the amplifiers and motors.



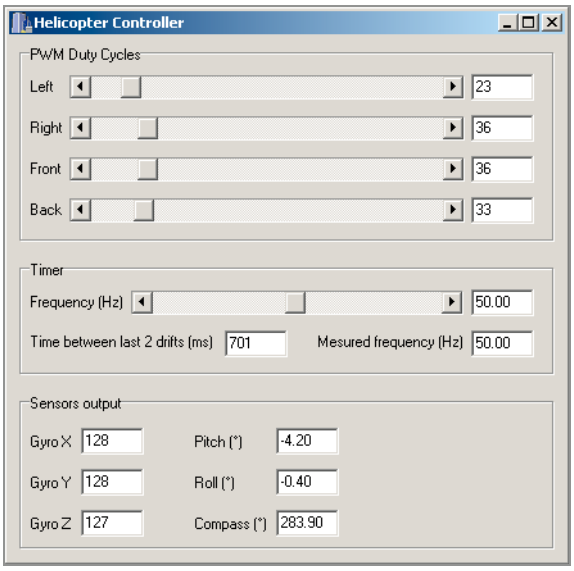


PC-BASED CONTROLLER SOFTWARE

To simplify the development of the helicopter controller, our idea is to run it remotely on the PC. Once the controller is completed and tested, it would be rewritten to be executed on the PCB's microcontrollers.

Overview

The program was written in C++ using Borland C++ 5. Using a Rapid Application Development environment enables fast creation of programs with user-friendly graphical interfaces. Furthermore, the core controller code can be written in pure C to be easily portable on the PICs (see annex F).



The program interface displays the values of the sensors on the PCB (gyroscopes and TCM2) and also offers controls to set each motor rotation speed (refer to annex B for more information). The controller's bandwidth (the rate at which the sensors values are read and the motor speeds are set) can be changed from 1 to 100Hz.

The application implements a preemptive multi-threaded structure:

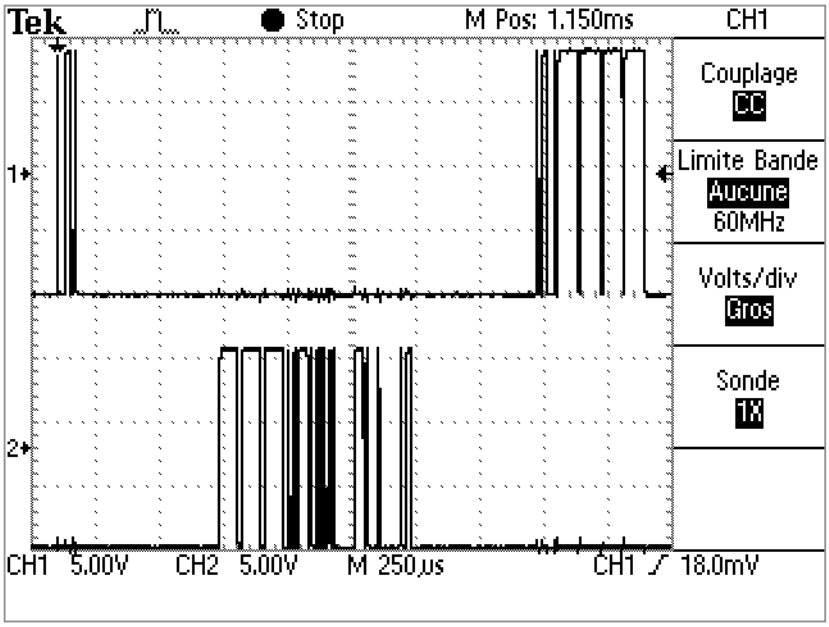
- The main thread handles all OS events and updates the display window with the latest sensor values (refresh rate is 25Hz).
- The controller thread handles communication with the PCB and has strictly no user interface. This thread is defined as "time-critical" to ensure it gets maximal processing power. The yet-to-be-written code to stabilize the helicopter will have to be inserted in this thread. This thread can *theoretically* run at frequencies up to 500Hz approximately.

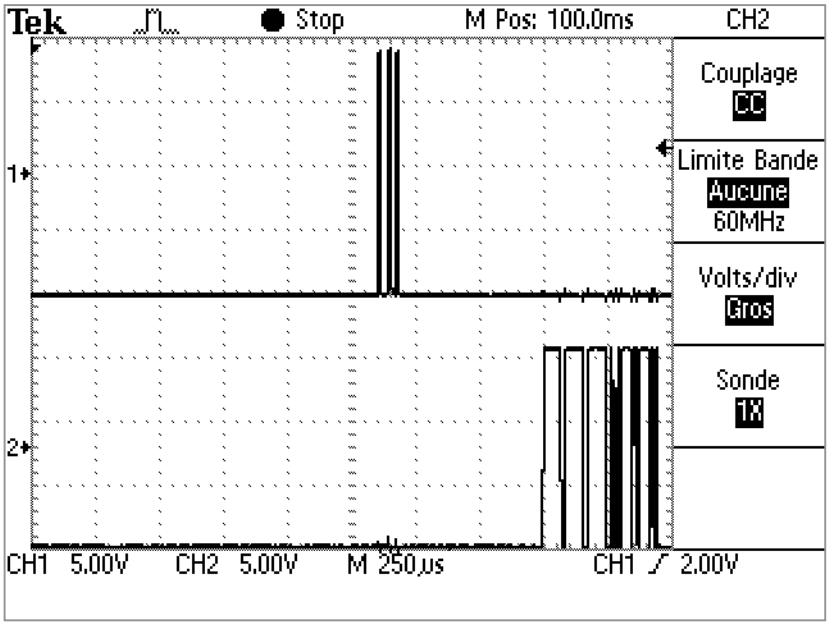
Timing accuracy

A controller must have a cycle time as stable as possible i.e. 20ms if the controller's frequency is set to 50Hz. Having the controller being executed on the PC make the task more complex since it has to deal with a heavy operating system.
To ensure the validity of the "controller-on-PC" concept, several measures have been performed, by monitoring the RS232 link, while the controller was set to run at 50Hz.

Important: the current controller code simply requests sensor values from the PCB and sends back the motor speeds defined by the user. Adding code to effectively control the helicopter will change the thread execution's duration and may invalidate these results.

In these two oscilloscope screen captures, the top and bottom signals respectively represent the PC transmit and receive RS232 pins.
We first evaluated the round-trip time of a complete controller sequence i.e. requesting the sensor values from the PCB (first pulse sequence), receiving them (second one) and sending back new motor speed values (third one): the result is about 2.5ms. Even if it will increase when the controller thread will implement actual computations, it should remain a few ms close to this very correct value – especially considering a 20ms controller's cycle time.





Secondly, by measuring the start offset of the sequence located at +100ms, we can compute a mean value on the last 5 cycle times; the result is 49.95Hz.

The fact that we do not have the requested 50Hz is very likely due to the execution time of the thread itself.

Finally, it is necessary to evaluate the stability of the controller's cycle time. According to timing measures done in the program itself, every 680 to 700ms, there is a drift of approximately 1 ms in the cycle time. We have no idea regarding the cause of this drift although properly reconfiguring the PC might help.
Further tests showed that under heavy OS load (for example, when moving around the program window very quickly), the frequency varies between the usual 49.95Hz and 50.25Hz.

Note: for unknown reasons, some frequencies cannot be reached by the controller thread e.g. it's impossible to have a frequency between 50 and 100Hz: only 50Hz and 100Hz are actually available.

CONCLUSION

Although we did not have time to test it extensively, the computer-based control system we have built seems to meet fully the initial requirements:

- angular orientation and angular velocity are successfully measured on all 3 axes,
- the motor speeds are controlled from the on-board microcontroller,
- we have a working two-ways RS232 wired link between the helicopter and the PC,
- the PC computer is able run the controller with acceptable timing precision,
- major processing power is remaining on the Master PIC and on the PC to do further computations,
- not all the helicopter payload has been used.

Even if we did not go as far as initially planned, the result of this project should be strong base for future autonomous helicopter projects at the ASL.

From a personal point of view, I have to say I have been very satisfied with this project since I have learned much in various areas through the development of this complete system from the ground up: choosing sensors and microcontrollers, designing and building a PCB, programming microcontrollers…

I'd like to thank Professor Roland Siegwart for setting up this project at my suggestion, Daniel Burnier and Jean-Christophe Zufferey for their time and precious help through the semester.
Many thanks to all other members of the ASL who took the time to answer my numerous questions.

ANNEX A: USING THE CONTROL SYSTEM

This page is intended as "Getting started" guide to the computer-based control system.

Starting the system

To use the helicopter control system, proceed as follow:
- Connect the RS232 cable to the PC (use the COM2 port).
- Power-up the PCB – use a power source between 7 and 9V DC that can source up to 20A current.
- Wait a few seconds to ensure complete startup of all the PCB components.
- Launch the control software.
- If all sensors values are updated correctly in the window, you may start using the system.

Warning: because the motors get warm rapidly, do not fly the helicopter for a longer time than 5 to 10mn. Then wait several minutes until the motors get back to normal temperature.

Troubleshooting

In case the system is not working, check the following issues:
- When the PCB is powered up, each PIC's LED should light up during 1 second,
- The Slave PIC's LED turns on when it detects the beginning of the data sent by the TCM2 and turns off when it detects the last byte. As a consequence, it should blink at the same clock rate than the TCM2 (normally 40Hz), as soon as the PCB is powered.
- The Master PIC's LED light up when it receives a "Get sensor values" command and turns off when it receives "Set motor speeds" command. Therefore it should blink at the controller's frequency.
- Try to communicate "manually" with the PCB through an RS232 terminal (refer to annex B).
- Make sure TCM2 settings are correct (refer to annex D).
- If the I^2C extension connector is not used, ensure the termination plug[1] is present.

Note: resetting only one PIC may put the PCB software in an undefined state (especially because of the I^2C communication between the 2 PICs). In case a reset is requested, it is highly recommended to reset both PICs simultaneously (simply disconnecting and reconnecting the PCB from power will do it).

[1] The termination is simply pull-up 10k resistances connected on the clock and data lines.

ANNEX B: COMMUNICATION PROTOCOLS

This sections describes in details the communication protocols used by the system, so that new functionalities may added later.

Communication between PICs

The Slave and Master PICs are connected through an I^2C bus at 400kBits. There is no real communication protocol between them since only two cases may occur:

- if the Slave is requested for data (I^2C slave transmitting mode), it returns the latest TCM2 values from its buffers,
- if the Slave receives data (I^2C slave receiving mode), it assumes this data contains the new PWM duty cycles values.

Bytes received from the Slave by the Master		
Byte #	**Format**	**Description**
1	Unsigned (0 -> 255 value)	Compass high-byte
2	Unsigned (0 -> 255 value)	Compass low-byte
3	Signed (-128 -> +127 value)	Pitch high-byte
4	Unsigned (0 -> 255 value)	Pitch low-byte
5	Signed (-128 -> +127 value)	Roll high-byte
6	Unsigned (0 -> 255 value)	Roll low-byte

Bytes sent to the Slave by the Master		
Byte #	**Format**	**Description**
1	Unsigned (0 -> 255 value)	Left PWM duty cycle
2	Unsigned (0 -> 255 value)	Right PWM duty cycle

Note: the PICs also implement a very basic communication scheme through hardware interrupts (refer to the end of the "Control system design" section).

Communication between PCB and PC

The PC is connected to the Master PIC through a 115'200bps RS232 link. The communication protocol is very simple: the PIC receives *commands* from the PC and possibly replies to them. There is no acknowledge nor checksum to ensure data reception and integrity.

A *command* is a simple byte possibly followed by data bytes. Depending on the command, the PIC will reply by sending back a predefined number of data bytes.

Warning: since the PIC does not queue commands but processes them immediately, sending commands too quickly[1] may overflow the RS232 reception buffer and paralyze the PIC into an undefined state.

The following commands are currently defined:

Name	Command byte	Command description
Get sensor values	'g'	Asks the PCB to return the current sensors values.
Set motor speeds	's'	Sets the duty cycles of the 4 motors PWM.[2]

Bytes to send with the "Set motor speeds" command		
Byte #	**Format**	**Description**
1	Unsigned (0 / 255 value)	Front PWM duty cycle
2	Unsigned (0 / 255 value)	Back PWM duty cycle
3	Unsigned (0 / 255 value)	Left PWM duty cycle
4	Unsigned (0 / 255 value)	Right PWM duty cycle

Bytes returned by the "Get sensor values" command		
Byte #	**Format**	**Description**
1	Unsigned (0 / 255 value)	X-axis gyroscope value
2	Unsigned (0 / 255 value)	Y-axis gyroscope value
3	Unsigned (0 / 255 value)	Z-axis gyroscope value
4	Unsigned (0 / 255 value)	Compass high-byte
5	Unsigned (0 / 255 value)	Compass low-byte
6	Signed (-128 / +127 value)	Pitch high-byte
7	Unsigned (0 / 255 value)	Pitch low-byte
8	Signed (-128 / +127 value)	Roll high-byte
9	Unsigned (0 / 255 value)	Roll low-byte

Important: compass, pitch and roll values are returned in a custom fixed-point format with one decimal digit for the fractional part e.g. the value "–34.2" will be sent as "-342".

[1] Successful tests have been performed with a controller frequency up to 100Hz i.e. 100 "Get sensor values" and "Set motor speeds" command are sent and received each second. Support for higher frequencies have not been tested.

[2] Depending on the options defined when the PIC source code was compiled, the new duty cycles values may not be effective immediately, but only on the next "Get sensor values" command (check source code for more information).

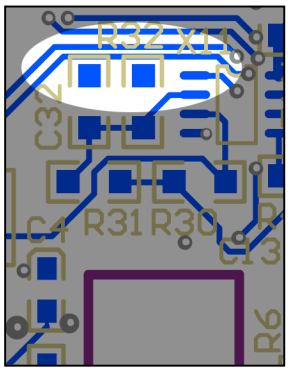
ANNEX C: PCB DESIGN ERRORS

Once the PCB was printed and we started mounting the components, we realized there were still some mistakes that were not found during the final check.

In the eventuality of the design of a new version of the PCB based on the layout of the current one, the following issues should be fixed (by order of importance):

- **The gyroscope footprint is wrong**: it was designed viewed from bottom - as a consequence, the footprint is mirrored vertically. Fortunately, we found a workaround by mounting "modified" connectors on the PCB that swap the pins 2-3 and 1-4. The gyros are then normally inserted into these connectors. This incorrect footprint only affects the horizontally mounted gyroscopes.

- As seen on this picture, **the closed-loop track on the Operational Amplifier X11 is missing** (it was accidentally deleted before the PCB layout was sent to printing).



- The 14-pins connector's position is slightly incorrect.

- The programming connector for the Slave PIC is a bit too close from the spacer that connects to the TCM2.

- The MAX232 and the C5 capacitor components are a bit too close from the spacer that connects to the lower PCB.

- There should be separate ground planes for both analogical and digital parts of the schematics to ensure the best possible electrical signal quality.

- The 16F876 PIC has 5 A/D inputs on pins RA0, RA1, RA2, RA3 and RA5, but only certain combinations of A/D and I/O inputs are allowed e.g. RA0, RA1 and RA5 as A/D and RA2 and RA3 as I/O. Therefore, the incoming gyro signals should have been connected to pins RA0, RA1 and RA5 instead of pins RA0, RA1 and RA2. The current situation requires all pins to be set as A/D since RA3 and RA5 cannot be set to I/O while the other pins are set to A/D.

ANNEX D: REQUIRED TCM2 SETTINGS

It's important that the TCM2 parameters are set correctly when the system is powered up because the onboard PCB will not check for incorrect settings nor change them.

Since these settings are stored in the sensor's EEPROM, they can safely be set by connecting the TCM2 to a PC computer and configure it through an RS232 terminal. The settings will remain even if the TCM2 is disconnected from power.
The only command that the PCB sends to the TCM2 at startup is the "Enter continuous sampling mode" to make sure it starts sending measures.

The TCM2 parameters should be set as follow (the command to set these parameters can be found in the TCM2 Users Manual):

Parameter	Value
Fast sampling	Enabled
Clock rate	40Hz [1]
Damping	Disabled
Inclinometer clipping value	0.0
Sampling period divisor	1
Baud rate	38400
Compass units	Degrees
Inclinometer units	Degrees
RS232 output word format	Standard
Compass data for output word	Enabled
Pitch data for output word	Enabled
Roll data for output word	Enabled
Magnetometer data for output word	Disabled
Temperature data for output word	Disabled
Magnetic distortion alarm	Enabled
Analog output	Disabled
Low power mode	Disabled [2]
New "Halt" command	Enabled
Magnetic north or true north	Magnetic
Declination Angle	0.0 [3]

Warning: modifying the format of the data sent by the TCM2 will very likely crash the PCB software.

[1] Any clock rate should work fine though.
[2] This feature was not tested.
[3] This value was left to the default setting.

ANNEX E: ORIGINAL PCB REVERSE-ENGINEERING

In order to minimize development time, we decided to re-use partially the original PCB of the Keyence Engager GS-III. This annex summarizes the reverse-engineering work that was done on this PCB.

The original PCB is made of 2 separate boards maintained together by mechanical connection, and electrically connected through a set of 14 pins.

- The lower board is essentially analogical: it provides 5V and 10V DC power, receives and possibly partially decodes radio signals, and hosts the PWM amplifiers to control the 4 electrical motors.
- The upper board contains the microcontroller and the angular velocity sensors (Murata ENC-05E). It handles radio signals decoding, PWM generation, gyroscope signal amplification and sampling.

The electrical motors are powered by 4kHz PWM amplifiers, which are controlled by 0-10V PWM signals. These are generated by the upper board, according to the radio control orders and gyroscopes outputs.

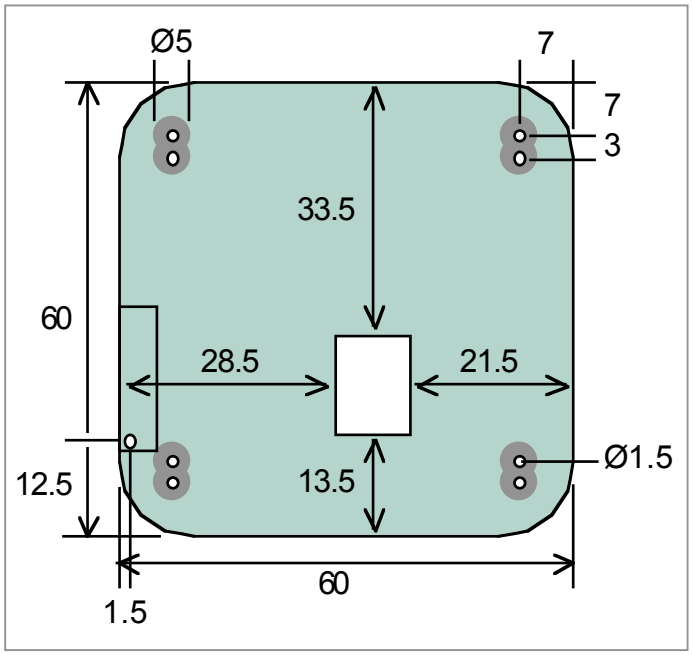
Here's the layout of the 14-pins connector:

Pin #	Signal (radiocontrol OFF)	Signal (radiocontrol ON)	Possible meaning
1	5V DC	0V	Radio control signal received correctly[1]
2	110mV DC	4kHz 10V PWM	Front motor control
3	110mV DC	4kHz 10V PWM	Back motor control
4	110mV DC	4kHz 10V PWM	Left motor control
5	110mV DC	4kHz 10V PWM	Right motor control
6	5V DC	5V DC	5V DC power
7	2.65V DC	Series of pulses	
8	Noise 0-4.8V	5V PWM	*Decoded radio signal?*
9	2.35V DC	2.35V DC	Raw vertical gyroscope sensor output
10	Ground	Ground	Ground
11	1.80V DC	Series of pulses	
12	4V	3V PWM	*Radio signal envelope?*
13	3.90V DC	Series of pulses	
14	10V DC	10V DC	10V DC power

Notes: we haven't measured how much current it is possible to sink from pins 6 and 14, however, these voltages remain stable as long as the PCB is powered with a voltage between 3.2V and 9V.

[1] There is a 2 seconds delay before the pin voltage drops to 0V once the radio control is turned ON.

This graph shows the mechanical layout of the original PCB upper board (dimensions are in mm).



ANNEX F: PICC COMPILER – PROS & CONS

To reduce development time, we decided to program the PIC microcontroller using a C-compiler from Custom Computer Services, Inc[1] instead of using raw assembly language. Although this compiler effectively helped software development, it has some serious flaws.

I did not find any problem or bug with the C-compiler itself: all C conventions seem to be perfectly implemented. The IDE is very decent and stable, compiles the source code quickly and spots many useful tools (Decimal / Hexadecimal converter, RS232 terminal, mixed C-code / generated assembly listing…). The PICC compiler also provides support for floating-point operations, trigonometric operations, string manipulations and more…

Assembly coding in the middle of C code is supported, but in a limited way, since register names are not predefined and no macros are available, e.g. for register bank selection. It's more efficient to define manually registers as C-variables (using the "#byte" preprocessor command) and manipulate them using standard C-code.

The provided documentation is sufficient, but far from being exhaustive: many functions (like the I^2C ones for example) would benefit from detailed explanations on how they exactly work.
Because of this limited documentation, and because you do not have access to the assembly code generated for the PICC built-in functions, you have no way to know how these functions handle the various special cases that may occur: buffer overflow, calling the function at interrupt time, successive calls without delays between them…

For example, in this project, we had to write custom versions of the I^2C routines, because the ones provided by PICC definitely did not work when doing multiple reads/writes (the example source code did not work either). We also had to write a special routine to handle the RS232 buffer overflow case, which was not handled at all by PICC. And finally, replacing the PICC 10bits to 8bits A/D conversion routines provided better results: the PICC routines simply truncate the original value, while the new routine rounds it to the nearest 8bits value.

As a conclusion, regarding the use of the PICC compiler versus assembly coding, I strongly recommend it, since it speeds up a lot development and debugging time - and above all, C code is way easier to read than assembly.
However, I suggest the creation, at the ASL, of custom replacement libraries for critical functions (I^2C, RS232, A/D conversions…) to ensure complete control over the generated assembly code.

[1] PICC IDE version 3.11 - Web site: http://www.ccsinfo.com

ANNEX G: TO DO LIST

Here's a list of ideas, in no particular order, which may be used as a starting point by the people who will continue on this project.

Helicopter

- Characterize the Murata ENC-05 gyroscopes[1] (angular velocity range, linearity, scale factor, bandwidth...).
- Characterize the TCM2-50 (latency, bandwidth, sensibility to vibrations…).
- Experiment with TCM2 digital damping settings.
- Test TCM2 low-power mode.
- Build a testing platform to safely try out the helicopter hovering controller.
- Use a BlueTooth module to communicate with the PC instead of a wired-RS232 connection.
- Very important: design a better fuselage, more robust, possibly lighter, and above all, with "true" fixations for the engines and future sensors.
- Replace motors and gears with more effective ones.
- Add feedback control to motors to increase speed control accuracy.
- Build a differential amplifier (with 2 or more operational amplifiers) that uses the Vref and Vout outputs from the Gyrostar ENC-05, to replace the current amplifiers.
- Replace the lower board of the PCB with one designed specifically for the project.

PCB Software

- Add better error handling e.g. RS232 buffer overflow, errors reported by the TCM2, I^2C writes not being acknowledged…
- Implement a security check that would stop all motors if no commands is received from the PC in a given laps of time.
- Implement a "Shutdown" command ('d') that would be sent by the PC to allow the PCB to perform clean up before being powered-down e.g. to stop the TCM2 continuous sampling.
- Implement an "Emergency stop" command ('h') that would immediately stop the motors and put the PCB software into an idle mode, virtually disconnected from the PC.
- Write custom versions of the RS232 routines from PICC, as it has been done for the I^2C routines, in order to have a better control over the USART unit.
- Implement a more robust communication protocol between PCB and PC: checksums, acknowledges…
- Write a custom interrupt handler that allows interrupt reentrancy with interrupts levels.
- On the Slave PIC, handle the RS232 communication with the TCM2 at interrupt level – this would free the main task for other computations.

[1] Check with the K-Team company which is also using these sensors (http://www.k-team.com).

PC Software

- Reformat the PC and re-install a clean OS with only strictly required applications and background services.
- Tune C++ code for maximal performance. This include setting compiler optimizations, removing debugging code if any, improve memory management, use as few as possible calls to system API into the controller thread…
- Use low-level Windows APIs to control the controller thread periodicity (find a way so that the thread gets called by the OS at regular times instead of putting itself into "sleep" until the next cycle starts).
- Add better error handling e.g. COM port not available, no more data received from the PCB, etc…
- Increase the controller frequency from 50Hz to 100Hz in order to improve the sampling quality of the gyroscope outputs (Nyquist's law). It may be necessary to smooth (interpolate) the TCM2 returned values, which would be still received at 40Hz only (Kalmann filter?).
- Use an asynchronous RS232 communication library to avoid blocking the controller thread if anything goes wrong on the RS232 link.

BIBLIOGRAPHY

Related projects

[1] Johann Borenstein, Advanced Technologies Lab, University of Michigan, "*The Hoverbot: An Electrically Powered Flying Robot*"
http://www-personal.engin.umich.edu/~johannb/hoverbot.htm

[2] Urs Baumann, Institut für Automatik, Diplomarbeit, ETHZ, "Fliegende Plattform" (PDF version of the report must be requested on the lab's web site)
http://www.ethz.ch

[3] Stephan Marti, Media Lab class project, MIT, "*The Zero-G Eye*"
http://web.media.mit.edu/~stefanm/HowTo/ZeroGEye.html

[4] Stephan Marti Master's Thesis, "*Autonomously flying micro robot*"
http://web.media.mit.edu/~stefanm/FFMP/Proposal.html

[5] Student project, Real Time Systems & Robotics, TU, "*Autonomously Operating Flying Robot*"
http://pdv.cs.tu-berlin.de/MARVIN/index.html
http://user.cs.tu-berlin.de/~remuss/marvin.html

[6] Willie Hazin, Intelligence Machine Design Laboratory, University of Florida, "*Autonomous vertical flight control system for helicopters*"
http://www.mil.ufl.edu/imdl/papers/IMDL_Report_Fall_98/Willie_Hazen/Cobra.pdf

Technical references

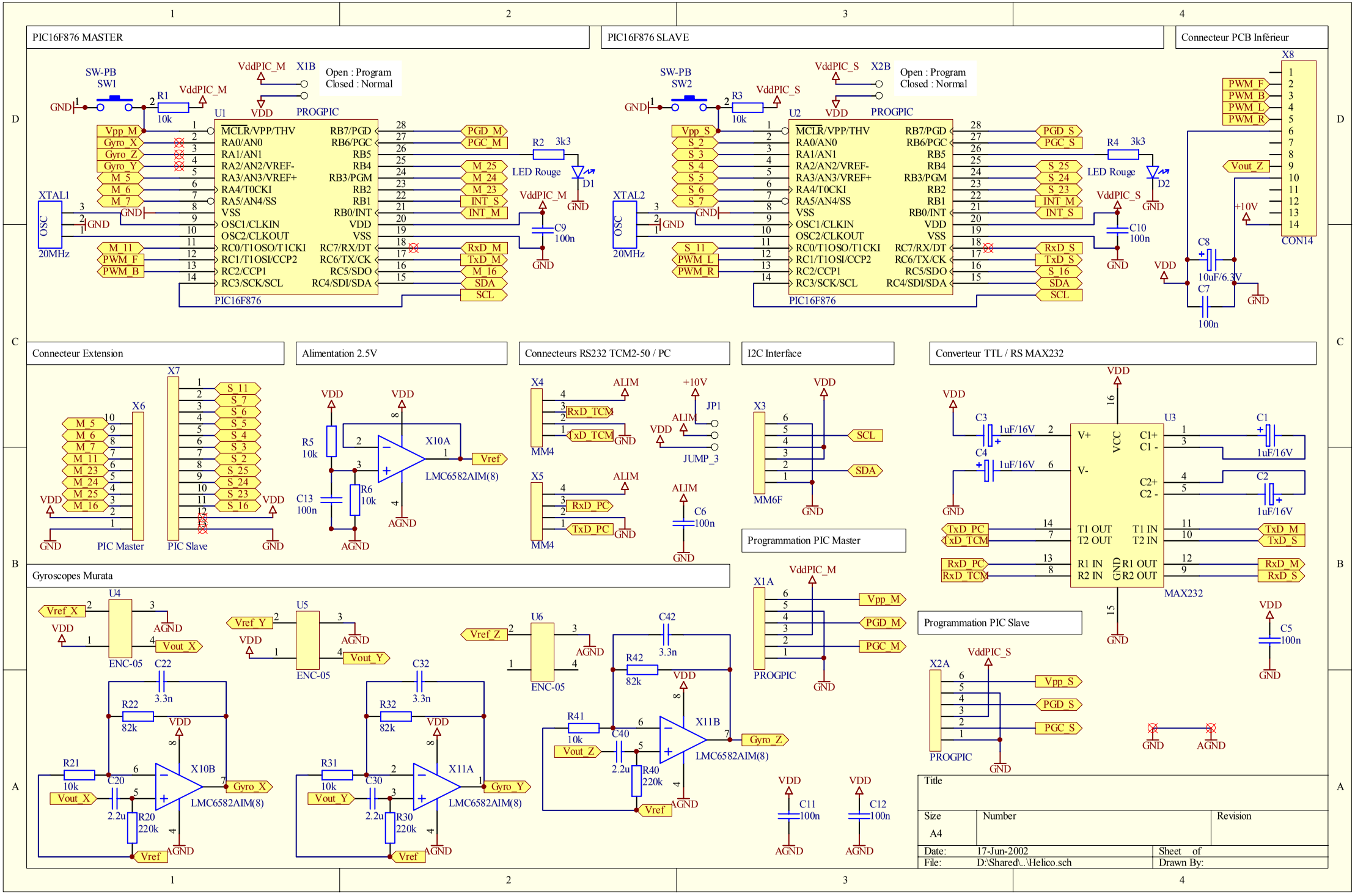
[7] Murata Electronics North America, "*Application guide for Gyrostar*"
http://www.murata.com

[8] Murata Manufacturing Co., Ltd., "Gyrostar ENC-03J datasheet"
http://www.murata.com

[9] National Semiconductor, "*LMC6582 Dual/LMC6584 Quad Low Voltage, Rail-To-Rail Input and Output CMOS Operational Amplifier*"
http://www.national.com/pf/LM/LMC6582.html

[10] Microchip, "*PIC16F87X Data Sheet*"
http://www.microchip.com

[11] Precision Navigation Inc., "TCM2 Manual"
http://www.pnicorp.com/technical-information/pdf/tcm2-50.pdf


PIC16F876 MASTER
PIC16F876 SLAVE
Connecteur PCB Inférieur
Connecteur Extension
Alimentation 2.5V
Connecteurs RS232 TCM2-50 / PC
I2C Interface
Converteur TTL / RS MAX232
Gyroscopes Murata
Programmation PIC Master
Programmation PIC Slave
SW-PB
SW1
SW2
R1
R2
R3
R4
10k
3k3
LED Rouge
D1
D2
U1
U2
PROGPIC
VddPIC_M
VddPIC_S
X1B
X2B
Open : Program
Closed : Normal
MCLR/VPP/THV
RA0/AN0
RA1/AN1
RA2/AN2/VREF-
RA3/AN3/VREF+
RA4/T0CKI
RA5/AN4/SS
VSS
OSC1/CLKIN
OSC2/CLKOUT
RC0/T1OSO/T1CKI
RC1/T1OSI/CCP2
RC2/CCP1
RC3/SCK/SCL
RB7/PGD
RB6/PGC
RB5
RB4
RB3/PGM
RB2
RB1
RB0/INT
VDD
VSS
RC7/RX/DT
RC6/TX/CK
RC5/SDO
RC4/SDI/SDA
PIC16F876
XTAL1
XTAL2
OSC
20MHz
C9
C10
100n
C8
10uF/6.3V
C7
X8
CON14
PWM_F
PWM_B
PWM_L
PWM_R
Vout_Z
+10V
X6
X7
PIC Master
PIC Slave
X4
X5
MM4
MM6F
JUMP_3
ALIM
RxD_TCM
TxD_TCM
RxD_PC
TxD_PC
R5
R6
C13
C6
X10A
LMC6582AIM(8)
Vref
AGND
X3
SCL
SDA
C3
C4
C1
C2
C5
1uF/16V
U3
MAX232
V+
V-
VCC
C1+
C1 -
C2+
C2 -
T1 OUT
T2 OUT
T1 IN
T2 IN
R1 IN
R2 IN
R1 OUT
R2 OUT
GND
X1A
X2A
U4
U5
U6
ENC-05
Vref_X
Vref_Y
Vref_Z
Vout_X
Vout_Y
C22
C32
C42
3.3n
R22
R32
R42
82k
R21
R31
R41
C20
C30
C40
2.2u
R20
R30
R40
220k
X10B
X11A
X11B
Gyro_X
Gyro_Y
Gyro_Z
C11
C12
Title
Size
A4
Number
Revision
Date:
17-Jun-2002
Sheet of
File:
D:\Shared\..\Helico.sch
Drawn By:

```
/*****************************************************************************/
/*  This header contains common definitions for both Master and Slave PICs   */
/*  EPFL IMT/ISR/ASL - Pierre-Olivier Latour 2002                            */
/*****************************************************************************/

/*
Preprocessor settings

PWM Mode 1: 100Khz PWM with duty cycle from 0 to 200
PWM Mode 2: 25Khz PWM with duty cycle from 0 to 200
PWM Mode 3: 4883Hz PWM with duty cycle from 0 to 256
PWM Mode 4: 19531Hz PWM with duty cycle from 0 to 256
PWM Mode 5: 19531Hz PWM with duty cycle from 0 to 1024 - NOT SUPPORTED!

Sync duty cycles: set to 1 to change the PWM duty cycles at the same time
on Master and Slave PICs (through a HW interrupt signal sent when the
GET_DATA command is received)
*/
#define __PWM_MODE__ 4
#define __SYNC_DUTY_CYCLES__ 1

/*
Global PIC settings
*/
#include <16F876.h>
#device *=16 ADC=10 //16 bits pointers and 10 bits ADC values
#use DELAY(CLOCK=20000000) //PIC is running at 20MHz
#fuses HS,NOWDT,PUT,NOBROWNOUT

/*
LED settings
*/
#define LED 53
#define LED_On() Output_High(LED);
#define LED_Off() Output_Low(LED);

/*
HW interrupt between Master and Slave
Send interrupt signal from B0
Receive interrupt signal on B1
*/
#define INT_OUT 49
#define Interrupt_Send() Output_High(INT_OUT);
#define Interrupt_Clear() Output_Low(INT_OUT);

/*
Set IO pins directions manually
*/
#use FAST_IO(A)
#use FAST_IO(B)
#use FAST_IO(C)

/*
I2C settings
*/
#define SLAVE_ADDRESS 0x30 //Last bit must be 0!

/*
A/D Converters settings
*/
#define ADC_CLOCK ADC_CLOCK_INTERNAL
#define ADC_DELAY 10 //in us - Delay time after changing the ADC channel to get a valid rea

/*
Startup delay
*/
#define STARTUP_DELAY 1000 //ms
```

```
/*
Motor PWM settings
(PWM period) = (1 / clock) * 4 * scaler * (Period + 1)
(PWM duty cycle max) = (PWM period) * clock / scaler = 4 * (Period + 1)
*/
#if __PWM_MODE__ == 1
#define kPWMScaler  T2_DIV_BY_1
#define kPWMPeriod  49
#elif __PWM_MODE__ == 2
#define kPWMScaler  T2_DIV_BY_4
#define kPWMPeriod  49
#elif __PWM_MODE__ == 3
#define kPWMScaler  T2_DIV_BY_16
#define kPWMPeriod  63
#elif __PWM_MODE__ == 4
#define kPWMScaler  T2_DIV_BY_4
#define kPWMPeriod  63
#elif __PWM_MODE__ == 5
#define kPWMScaler  T2_DIV_BY_1
#define kPWMPeriod  255
#endif

/*
IO pins directions macros
*/
#define kPinInput   1
#define kPinOutput  0
#define kPinADC     kPinInput //Pins A0,A1,A2,A3 and A5 only!
#define kPinPWM     kPinOutput //Pins C1 and C2 only!
#define kPinInINT   kPinInput //Pin B0 only!
#define kPinOutINT  kPinOutput //Pin B1 only!
#define BuildDirectionByte(p0,p1,p2,p3,p4,p5,p6,p7) (((p7) << 7) | ((p6) << 6) | ((p5) << 5)
#define SetPortDirections_A()  Set_Tris_A(BuildDirectionByte(kPortA_0, kPortA_1, kPortA_2, kI
#define SetPortDirections_B()  Set_Tris_B(BuildDirectionByte(kPortB_0, kPortB_1, kPortB_2, kI
#define SetPortDirections_C()  Set_Tris_C(BuildDirectionByte(kPortC_0, kPortC_1, kPortC_2, kI

/*
16Bits to 8Bits macros
*/
#define HiByte(v) (((v) >> 8) & 0xFF)
#define LoByte(v) ((v) & 0xFF)
```

```
/*****************************************************************************/
/*  This file contains source code for Master PIC                            */
/*  EPFL IMT/ISR/ASL - Pierre-Olivier Latour 2002                            */
/*****************************************************************************/

#include "Header.h"
#include "I2C.h"

/*****************************************************************************/
/*                                  DEFINITIONS                              */
/*****************************************************************************/

//RS232 parameters for communicating with PC
#use  RS232(baud=115200,parity=N,xmit=PIN_C6,rcv=PIN_C7)

//I2C parameters for communicating with PIC Slave
#use  I2C(MASTER,sda=PIN_C4,scl=PIN_C3,FORCE_HW,FAST)

//IO pins directions
enum {
    kPortA_0 = kPinADC, //Gyro X
    kPortA_1 = kPinADC, //Gyro Y
    kPortA_2 = kPinADC, //Gyro Z
    kPortA_3 = kPinADC, //ADC channel - UNUSED
    kPortA_4 = kPinOutput,
    kPortA_5 = kPinADC, //ADC channel - UNUSED
    kPortA_6 = kPinOutput,
    kPortA_7 = kPinOutput
};
enum {
    kPortB_0 = kPinOutput,
    kPortB_1 = kPinOutINT, //Send HW interrupt signal to slave
    kPortB_2 = kPinOutput,
    kPortB_3 = kPinOutput,
    kPortB_4 = kPinOutput,
    kPortB_5 = kPinOutput, //LED
    kPortB_6 = kPinOutput,
    kPortB_7 = kPinOutput
};
enum {
    kPortC_0 = kPinOutput,
    kPortC_1 = kPinPWM, //PWM 2 (Forward motor)
    kPortC_2 = kPinPWM, //PWM 1 (Backward motor)
    kPortC_3 = kPinInput, //I2C clock
    kPortC_4 = kPinInput, //I2C data
    kPortC_5 = kPinOutput,
    kPortC_6 = kPinOutput, //RS232 transmit
    kPortC_7 = kPinInput //RS232 receive
};

/*
RS232 protocol between PC and PIC Master

-> Byte #1: command = COMMAND_SETPWM
-> Byte #2: PWM forward
-> Byte #3: PWM backward
-> Byte #4: PWM left
-> Byte #5: PWM right

-> Byte #1: command = COMMAND_GETDATA
<- Byte #2..n: gyro values and TCM values (see format below)

*/
#define  COMMAND_GETDATA 'g'
#define COMMAND_SETPWM  's'

//Format of the data sent to the PC
enum {
```

```
    kData_GyroX = 0,
    kData_GyroY,
    kData_GyroZ,
    kData_CompassHB,
    kData_CompassLB,
    kData_PitchHB,
    kData_PitchLB,
    kData_RollHB,
    kData_RollLB,
    kDataSize
};

/*****************************************************************************/
/*                                GLOBAL VARIABLES                           */
/*****************************************************************************/

long        dutyForward = 0,
            dutyBackward = 0;

/*****************************************************************************/
/*                                    ROUTINES                               */
/*****************************************************************************/

/*
This routine send a buffer over the RS232 to the PC
*/
void SendBuffer(char* buffer, int size)
{
    while(size) {
        PutC(*buffer);
        --size;
        ++buffer;
    }
}

/*
This routine reads a 10bits value on an ADC channel and returns an 8bits value
*/
int Read_ADC_Channel(int num)
{
    long        value;
    int         result;

    //Select channel and wait a short time to make sure we will get a valid ADC read
    Set_ADC_Channel(num);
    Delay_US(ADC_DELAY);

    //Read ADC
    value = Read_ADC();

    //Round value to nearest 8bits value according to last 2 bits
    if(value & 0x0002)
    result = (int) (value >> 2) + 1;
    else
    result = (int) (value >> 2);

    return result;
}

/*
This routine is called when data is received over RS232 from the PC

    - To access the slave in transmitting mode, last bit of the Slave address must
      be set to 1.
    - In Master-receiving mode, last read must have acknowledge bit set to 0 to
      indicate end of transfer so that slave can start again watching for
      "Start" conditions on the I2C bus.
*/
```

```
#INT_RDA
void  RS232_Interrupt()
{
    byte        incoming;
    int         buffer[kDataSize];
    int         dutyLeft,
                dutyRight;

Start:
    incoming = GetC();
    switch(incoming) {

        case  COMMAND_GETDATA:
        //Flash LED
        LED_On();

        //Update the PWM duty cycles
#if  __SYNC_DUTY_CYCLES__
        Interrupt_Send();
        Set_PWM1_Duty(dutyBackward);
        Set_PWM2_Duty(dutyForward);
        Interrupt_Clear();
#endif

        //Read values from gyros X,Y and Z - Y and Z are swapped on PIC pins!
        buffer[kData_GyroX]  =  Read_ADC_Channel(0);
        buffer[kData_GyroZ]  =  Read_ADC_Channel(1);
        buffer[kData_GyroY]  =  Read_ADC_Channel(2);

        //Read values from TCM
        I2CMaster_Start();
        I2CMaster_Write(SLAVE_ADDRESS  |  1);
        buffer[kData_CompassHB]  =  I2CMaster_Read(1);
        buffer[kData_CompassLB]  =  I2CMaster_Read(1);
        buffer[kData_PitchHB]  =  I2CMaster_Read(1);
        buffer[kData_PitchLB]  =  I2CMaster_Read(1);
        buffer[kData_RollHB]  =  I2CMaster_Read(1);
        buffer[kData_RollLB]  =  I2CMaster_Read(0);
        I2CMaster_Stop();

        //Send data to PC
        SendBuffer(buffer,  kDataSize);
        break;

        case  COMMAND_SETPWM:
        //Flash LED
        LED_Off();

        //Get PWM duty cycles from PC
        dutyForward = GetC();
        dutyBackward = GetC();
        dutyLeft = GetC();
        dutyRight = GetC();

        //Send PWM duty cycles to Slave - FIXME: check for ack bit!
        I2CMaster_Start();
        I2CMaster_Write(SLAVE_ADDRESS);
        I2CMaster_Write(dutyLeft);
        I2CMaster_Write(dutyRight);
        I2CMaster_Stop();

#if  !__SYNC_DUTY_CYCLES__
        //Set duty cycles
        Set_PWM1_Duty(dutyBackward);
        Set_PWM2_Duty(dutyForward);
#endif
        break;
```

```
    }

    //Make sure there is no remaining data to be processed
    if(KBHit())
    goto Start;
}

/*
Main routine
*/
void main()
{
    //Flash LED
    LED_On();

    //Setup both PWMs
    Setup_CCP1(CCP_PWM);
    Setup_CCP2(CCP_PWM);
    Setup_Timer_2(kPWMScaler, kPWMPeriod, 1);
    Set_PWM1_Duty(0);
    Set_PWM2_Duty(0);

    //Setup ADCs
    Setup_ADC_Ports(ALL_ANALOG); //See possible values in 16F876.h
    Setup_ADC(ADC_CLOCK);

    //Setup IO pins directions
    SetPortDirections_A();
    SetPortDirections_B();
    SetPortDirections_C();

    //Startup delay
    Delay_MS(STARTUP_DELAY);
    LED_Off();

    //Set interrupts
    Enable_Interrupts(GLOBAL);
    Enable_Interrupts(INT_RDA);

    //Run...
    while(1)
    ;
}
```

```
/*******************************************************************************/
/*  This file contains source code for Slave PIC                               */
/*  EPFL IMT/ISR/ASL - Pierre-Olivier Latour 2002                              */
/*******************************************************************************/

#include "Header.h"
#include "I2C.h"
#include "RS232.h"

/*******************************************************************************/
/*                                  DEFINITIONS                                */
/*******************************************************************************/

//RS232 parameters for communicating with PC
#use  RS232(baud=38400,parity=N,xmit=PIN_C6,rcv=PIN_C7)

//I2C parameters for communicating with PIC Slave
#use  I2C(SLAVE,scl=PIN_C3,sda=PIN_C4,address=SLAVE_ADDRESS,FORCE_HW,FAST)

//Interrupts priority
#priority  EXT,SSP,RDA

//IO pins directions
enum {
    kPortA_0 = kPinOutput,
    kPortA_1 = kPinOutput,
    kPortA_2 = kPinOutput,
    kPortA_3 = kPinOutput,
    kPortA_4 = kPinOutput,
    kPortA_5 = kPinOutput,
    kPortA_6 = kPinOutput,
    kPortA_7 = kPinOutput
};
enum {
    kPortB_0 = kPinInINT, //Receive HW interrupt signal from slave
    kPortB_1 = kPinOutput,
    kPortB_2 = kPinOutput,
    kPortB_3 = kPinOutput,
    kPortB_4 = kPinOutput,
    kPortB_5 = kPinOutput, //LED
    kPortB_6 = kPinOutput,
    kPortB_7 = kPinOutput
};
enum {
    kPortC_0 = kPinOutput,
    kPortC_1 = kPinPWM, //PWM 2 (Left motor)
    kPortC_2 = kPinPWM, //PWM 1 (Right motor)
    kPortC_3 = kPinInput, //I2C clock (in Slave mode, this pin must be set as input)
    kPortC_4 = kPinInput, //I2C data (in Slave mode, this pin must be set as input)
    kPortC_5 = kPinOutput,
    kPortC_6 = kPinOutput, //RS232 transmit
    kPortC_7 = kPinInput //RS232 receive
};

//TCM2 data formating
#define     TCM_MAX_DATASIZE    32
#define     TCM_START_CHAR      '$'
#define     TCM_END_CHAR        '\n'
#define     TCM_ERROR_CHAR      'E'

//TCM2 commands
#define     TCMCOMMAND_START    "go\r"
#define     TCMCOMMAND_STOP     "h" //"h\r" (new halt command in TCM2 firmware 1.07 is one

/*******************************************************************************/
/*                                GLOBAL VARIABLES                             */
/*******************************************************************************/
```

```
long        dutyLeft = 0,
            dutyRight = 0;
long        compass = 0;
signed long pitch = 0,
            roll = 0;
int         readIndex = 0;
int         writeIndex = 0;

/*****************************************************************************/
/*                                    ROUTINES                               */
/*****************************************************************************/

/*
This routine is called when the external interrupt is trigered from the Master PIC
*/
#if __SYNC_DUTY_CYCLES__
#INT_EXT
void External_Interrupt()
{
    //Update PWM duty cycles
    Set_PWM1_Duty(dutyRight);
    Set_PWM2_Duty(dutyLeft);
}
#endif

/*
This routines sets a byte from the global variables
*/
void SetByte(int index, int value)
{
    switch(index) {

        case 0:
        dutyLeft = value;
        break;

        case 1:
        dutyRight = value;
        break;

    }
}

/*
This routines gets a byte from the global variables
*/
int GetByte(int index)
{
    switch(index) {

        case 0:
        return (int) HiByte(compass);
        break;

        case 1:
        return (int) LoByte(compass);
        break;

        case 2:
        return (int) HiByte(pitch);
        break;

        case 3:
        return (int) LoByte(pitch);
        break;

        case 4:
        return (int) HiByte(roll);
```

```
        break;

        case 5:
        return (int) LoByte(roll);
        break;

    }
}

/*
This routine is called when data is received over I2C from the Master PIC
It must be very fast since it interrupts RS232 decoding
*/
#INT_SSP
void I2C_Interrupt()
{
    int         flags;
    int         incoming;

    //Get immediately useful bits from SSPSTAT register
    flags = SSPSTAT & (BF | R_W | S | P | D_A);

    //I2C write operation, last byte was an address byte, buffer is full
    if(flags == (BF | S)) {
        incoming = I2CSlave_Read(); //Receive byte #0 (it is actually the slave address)
        writeIndex = 0;
    }
    //I2C Write operation, last byte was data, buffer is full
    else if(flags == (BF | S | D_A)) {
        incoming = I2CSlave_Read(); //Receive byte #1..(n-1)
        SetByte(writeIndex++, incoming);
    }
    //I2C Write operation, last byte was data, buffer is full
    else if(flags == (BF | P | D_A)) {
        incoming = I2CSlave_Read(); //Receive byte #n
        SetByte(writeIndex++, incoming);

#if !__SYNC_DUTY_CYCLES__
        //Update PWM duty cycles
        Set_PWM1_Duty(dutyRight);
        Set_PWM2_Duty(dutyLeft);
#endif
    }
    //I2C read operation, last byte was an address byte, buffer is empty
    //(it actually contains the slave address)
    else if(flags == (R_W | S)) {
        readIndex = 0;
        I2CSlave_Write(GetByte(readIndex++)); //Transmit byte #0
    }
    //I2C read operation, last byte was a data byte, buffer is empty
    else if(flags == (R_W | S | D_A)) {
        I2CSlave_Write(GetByte(readIndex++)); //Transmit bytes #1..n
    }
#if 0
    //Slave I2C logic reset by NACK from master
    //Slave logic is reset in this case and starts waiting for next START bit
    else if(flags == (S | D_A)) {
        printf("CATCH!\n"); //Do nothing
    }
    //Undefined status!
    else
    printf("FATAL ERROR: %2X\n", flags); //Fatal error!
#endif
}

/*
Convert a string with a decimal part to a number x 10
Input string must be of the form -xxx.xxx
```

```
Input string must use only '0..9','.' or '-' character
Input string may end with any other character
The returned value is an integer number with last digit being the decimal part
The returned value is between -3276.8 and +3276.7
*/
signed long  StringToNum(char* c)
{
    signed long     num = 0;
    int             fractional = FALSE,
                    minus;

    //Check for minus
    if(*c == '-') {
        minus = TRUE;
        ++c;
    }
    else
    minus = FALSE;

    //Read digits
    while(1) {
        if((*c >= '0') && (*c <= '9')) {
            num *= 10;
            num += *c - '0';
            if(fractional)
            break;
        }
        else if(*c == '.')
        fractional = TRUE;
        else
        break;

        ++c;
    }

    //Check if any fractional part was found
    if(!fractional)
    num *= 10;

    //Return value
    if(minus)
    return -num;
    else
    return num;
}

/*
This routine returns the first occurence of a given character in a string
This routine is case-sensitive
*/
char*  ScanForChar(char* buffer, char c)
{
    while(*buffer != c)
    ++buffer;

    return buffer;
}

/*
This routine parses the TCM output data and extract compass, pitch and roll values

At 20MHz, a PIC executes between 2.5x10^6 and 5.0x10^6 instructions per second
    -> 0.4us or 0.2us per instruction
ScanForChar is 5+14*n instructions (n is the number of chars to skip - 5 maximum) -> 75
StringToNum is 79+46*n instructions worst case (n is the number of digits in the
    number - 5 maximum) -> 309
Parse_TCMData is 114 instructions long -> 114 + 3 * (75 + 309) = 1266 -> 0.5ms
    worst case approximately while TCM2 runs at 40Hz=1/25ms
```

```
*/
void Parse_TCMData(char* buffer, int size)
{
    long        tempCompass;
    signed long tempPitch,
                tempRoll;

    //Extract compass value
    buffer = ScanForChar(buffer, 'C') + 1;
    tempCompass = StringToNum(buffer);

    //Extract pitch value
    buffer = ScanForChar(buffer, 'P') + 1;
    tempPitch = StringToNum(buffer);

    //Extract roll value
    buffer = ScanForChar(buffer, 'R') + 1;
    tempRoll = StringToNum(buffer);

    //Update global variables - prevents I2C from reading them in the meanwhile
    Disable_Interrupts(INT_SSP);
    compass = tempCompass;
    pitch = tempPitch;
    roll = tempRoll;
    Enable_Interrupts(INT_SSP);
}

/*
This routines waits for a string formatted as "$C328.2P-15.3R20.7*XX\r\n" from the
TCM2 and extracts the values
*/
void Process_TCMData()
{
    char        buffer[TCM_MAX_DATASIZE];
    int         size = 0;

    //Wait for start char
    while(1) {
        buffer[0] = GetC();
        if(buffer[0] == TCM_START_CHAR)
        break;
    }
    ++size;

    //Flash LED
    LED_On();

    //Copy data and wait for end char
    while(1) {
        buffer[size] = GetC();
        if(buffer[size] == TCM_ERROR_CHAR) //Check for error returned by the TCM2
        goto End;
        ++size;
        if(size == TCM_MAX_DATASIZE) //Check for data overflow
        goto End;

        if(buffer[size - 1] == TCM_END_CHAR)
        break;
    }

    //Parse data
    Parse_TCMData(buffer, size);

End:
    //Flash LED
    LED_Off();
}
```

```
/*
Main routine
*/
void main()
{
    //Flash LED
    LED_On();

    //Setup both PWMs
    Setup_CCP1(CCP_PWM);
    Setup_CCP2(CCP_PWM);
    Setup_Timer_2(kPWMScaler, kPWMPeriod, 1);
    Set_PWM1_Duty(0);
    Set_PWM2_Duty(0);

    //Setup IO pins directions
    SetPortDirections_A();
    SetPortDirections_B();
    SetPortDirections_C();

    //Stop TCM2
    printf(TCMCOMMAND_STOP);

    //Startup delay
    Delay_MS(STARTUP_DELAY);
    LED_Off();

    //Reset the USART if we have overrunned
    //(possible because PICC starts the USART reception at the beginning of "main")
    RS232_ClearOverRun();

    //Start TCM2 - FIXME: check for acknowledge from TCM2!
    printf(TCMCOMMAND_START);

    //Set interrupts
#if __SYNC_DUTY_CYCLES__
    EXT_INT_Edge(0, L_TO_H);
#endif
    Enable_Interrupts(GLOBAL);
    Enable_Interrupts(INT_SSP);
#if __SYNC_DUTY_CYCLES__
    Enable_Interrupts(INT_EXT);
#endif

    //Run...
    while(1)
    Process_TCMData();
}
```

```
/************************************************************************************/
/*  This header contains code for I2C support on PIC 16F876                         */
/*  EPFL IMT/ISR/ASL - Pierre-Olivier Latour 2002                                   */
/*                                                                                  */
/*  IMPORTANT NOTES:                                                                */
/*  _ Do not enable the SSP interrupt in Master mode since this code was designed  */
/*      to work with SSP interrupt OFF in Master mode.                              */
/*  _ All routines in this file are synchronous (i.e. they block until execution is */
/*      complete) but I2CSlave_Write which returns before the byte is sent.         */
/*  _ In slave-receiving mode, the ack bit is sent as 1 automatically by the SSP    */
/*      hardware if BF and SSPOV bits are cleared.                                  */
/*  _ In slave-transmitting mode, the ack bit is always sent as 1 by the SSP        */
/*      hardware.                                                                   */
/*  _ A call to I2CMaster_Start() or to I2CMaster_Restart() may generate an bus     */
/*      collision interrupt in certain circonstances.                               */
/*  _ This code is not Multi-Master aware nor General Call aware.                   */
/*  _ This code was only tested in Fast mode on PIC 16F876 (400KBits).              */
/*  _ Pass a non-zero value to I2CMaster_Read() to send an Acknowledge to the slave.*/
/*  _ I2CMaster_Write() returns a 1 if the slave has acknowledged.                  */
/*  _ In the slave code, the SSP interrupt routine must be as fast as possible to   */
/*      avoid dropping bytes if the master does successive read/write without any   */
/*      delay between them.                                                         */
/*  _ DO NOT MIX calls to these routines with calls to PICC I2C routines - only the */
/*      use of #USE I2C(...) is permitted.                                          */
/************************************************************************************/

/*
SSP registers
*/
#byte SSPSTAT = 0x0094 //SSP status register
#byte SSPCON = 0x0014 //SSP control register
#byte SSPCON2 = 0x0091 //SSP control register 2
#byte SSPBUF = 0x0013 //serial receive/transmit buffer

/* SSP status register bits (SSPSTAT)

BF - Receive:           1 -> Receive complete, SSPBUF is full
                        0 -> Receive not complete, SSPBUF is empty
BF - Transmit:          1 -> Data transmit in progress (does not include the ACK
                             and STOP bits), SSPBUF is full
                        0 -> Data transmit complete (does not include the ACK and
                             STOP bits), SSPBUF is empty
R_W - Slave mode:       1 -> Read
                        0 -> Write
R_W - Master mode:      1 -> Transmit is in progress
                        0 -> Transmit is not in progress
S:                      1 -> Indicates that a START bit has been detected last
                             (this bit is '0' on RESET)
                        0 -> START bit was not detected last
P:                      1 -> Indicates that a STOP bit has been detected last
                             (this bit is '0' on RESET)
                        0 -> STOP bit was not detected last
D_A:                    1 -> Indicates that the last byte received or
                             transmitted was data
                        0 -> Indicates that the last byte received or
                             transmitted was address
*/
#define BF      (1 << 0) //Buffer Full Status bit
#define R_W     (1 << 2) //Read/Write bit Information
#define S       (1 << 3) //START bit
#define P       (1 << 4) //STOP bit
#define D_A     (1 << 5) //Data/Address bit

/* SSP control register bits (SSPCON)

CKP - Master mode:      UNUSED!
CKP - Slave mode:       1 -> Enable clock
                        0 -> Holds clock low (clock stretch)
```

```
  SSPOV:                    1 -> A byte is received while the SSPBUF is holding the
                                 previous byte (SSPOV is a "don't care" in Transmit)
                            0 -> No overflow
  WCOL - Master mode:       1 -> A write to SSPBUF was attempted while the I2C
                                 conditions were not valid
                            0 -> No collision
  WCOL - Slave mode:        1 -> SSPBUF register is written while still transmitting
                                 the previous word
                            0 -> No collision
  */
  #define CKP     (1 << 4) //SCK release control
  #define SSPOV   (1 << 6) //Receive Overflow Indicator bit
  #define WCOL    (1 << 7) //Write collision detect bit

  /* SSP control register 2 bits (SSPCON2) - Bits 0-6 are valid only in Master mode!

  SEN:                      1 -> Initiate START condition on SDA and SCL pins
                                 (Automatically cleared by hardware)
                            0 -> START condition idle
  RSEN:                     1 -> Initiate Repeated START condition on SDA and SCL pins
                                 (Automatically cleared by hardware)
                            0 -> Repeated START condition idle
  PEN:                      1 -> Initiate STOP condition on SDA and SCL pins
                                 (Automatically cleared by hardware)
                            0 -> STOP condition idle
  RCEN:                     1 -> Enables Receive mode for I2C
                            0 -> Receive idle
  ACKEN - Receive mode:     1 -> Initiate Acknowledge sequence on SDA and SCL pins and
                                 transmit ACKDT data bit
                                 (Automatically cleared by hardware)
                            0 -> Acknowledge sequence idle
  ACKDT - Receive mode:     Value that will be transmitted when the user initiates an
                                 Acknowledge sequence at the end of a receive.
  ACKSTAT - Transmit mode 1 -> Acknowledge was not received from slave
                            0 -> Acknowledge was received from slave
  */
  #define SEN     (1 << 0) //START Condition Enable bit
  #define RSEN    (1 << 1) //Repeated START Condition Enable bit
  #define PEN     (1 << 2) //STOP Condition Enable bit
  #define RCEN    (1 << 3) //Receive Enable bit
  #define ACKEN   (1 << 4) //Acknowledge Sequence Enable bit
  #define ACKDT   (1 << 5) //Acknowledge Data bit
  #define ACKSTAT (1 << 6) //Acknowledge Status bit

  /*
  This routine writes a byte on the I2C bus
  */
  void I2CSlave_Write(int value)
  {
      //Wait until the buffer is free
      while(SSPSTAT & BF)
      ;

  Write:
      //Clear the write collision flag
      SSPCON &= ~WCOL;

      //Write byte to buffer
      SSPBUF = value;

      //Check for write collision
      if(SSPCON & WCOL)
      goto Write;

      //Release the clock
      SSPCON |= CKP;
  }
```

```
/*
This routine returns the content of the SSP buffer
*/
int  I2CSlave_Read()
{
    //Clear overflow bit - FIXME: it's better to handle the overflow case properly
    //inside the SSP interrupt routine
    SSPCON &= ~SSPOV;

    //Return content of SSP buffer - this also clears the buffer full (BF) bit
    return SSPBUF;
}

/*
This routine initiates a transfer on the I2C bus
*/
void  I2CMaster_Start()
{
    //Enable START bit
    do {
        SSPCON2 |= SEN;
    } while(!(SSPCON2 & SEN));

    //Wait for completion
    while(SSPCON2 & SEN)
    ;
}

/*
This routine re-initiates a transfer on the I2C bus
*/
void  I2CMaster_Restart()
{
    //Enable RESTART bit
    do {
        SSPCON2 |= RSEN;
    } while(!(SSPCON2 & RSEN));

    //Wait for completion
    while(SSPCON2 & RSEN)
    ;
}

/*
This routine finishes a transfer on the I2C bus
*/
void  I2CMaster_Stop()
{
    //Enable STOP bit
    do {
        SSPCON2 |= PEN;
    } while(!(SSPCON2 & PEN));

    //Wait for completion
    while(SSPCON2 & PEN)
    ;
}

/*
This routine writes a byte on the I2C bus and returns the ACK bit
*/
int  I2CMaster_Write(int value)
{
    //Wait until the buffer is free
    while(SSPSTAT & BF)
    ;

Write:
```

```
    //Clear the write collision flag
    SSPCON &= ~WCOL;

    //Write byte to buffer
    SSPBUF = value;

    //Check for write collision
    if(SSPCON & WCOL)
    goto Write;

    //Wait until the byte has been sent
    while(SSPSTAT & BF)
    ;

    //Return the value of the ack bit - FIXME: may not be set at this time?
    if(SSPCON2 & ACKSTAT)
    return 0;
    else
    return 1;
}

/*
This routine reads a byte on the I2C bus and send the ACK bit
*/
int I2CMaster_Read(int acknowledge)
{
    int         value;

    //FIXME: check for overflow bit!

    //Enter reception mode
    do {
        SSPCON2 |= RCEN;
    } while(!(SSPCON2 & RCEN)); //FIXME: correct?

    //Wait for data
    while(!(SSPSTAT & BF))
    ;

    //Reads content of SSP buffer - this also clears the buffer full (BF) bit
    value = SSPBUF;

    //Send acknowledge bit
    if(acknowledge)
    SSPCON2 &= ~ACKDT;
    else
    SSPCON2 |= ACKDT;
    SSPCON2 |= ACKEN;

    //Wait until acknowledge has been sent
    while(SSPCON2 & ACKEN)
    ;

    return value;
}
```

```
/*****************************************************************************/
/*  This header contains code for RS232 support on PIC 16F876                */
/*  EPFL IMT/ISR/ASL - Pierre-Olivier Latour 2002                            */
/*                                                                           */
/*  IMPORTANT NOTES:                                                         */
/*  _ PICC RS232 routines DO NOT handle the case where the USART FIFO buffer gets   */
/*      full therefore the need for this file.                               */
/*****************************************************************************/

/*
USART registers
*/
#byte RCSTA = 0x0018 //receive status and control register
#byte RCREG = 0x001A //receive register

/* SSP status register bits (SSPSTAT)

OERR:               1 -> Overrun error (can be cleared by clearing bit CREN)
                    0 -> No overrun error
CREN:               1 -> Enables continuous receive
                    0 -> Disables continuous receive
*/
#define OERR    (1 << 1) //Overrun Error bit
#define CREN    (1 << 4) //Continuous Receive Enable bit

/*
This routines checks if the OERR bit is set.
The OERR bit is set if the USART FIFO buffer is full and further reception is
disabled.
Since OERR is read-only, it has to be cleared by disabling and re-enabling
the CREN bit.
*/
void  RS232_ClearOverRun()
{
    int         value;

    if(RCSTA & OERR) {
        //Disable receive
        RCSTA &= ~CREN;

        //Dummy read of the receive FIFO buffer to empty it
        value = RCREG;
        value = RCREG;

        //Re-enable receive
        RCSTA |= CREN;
    }
}
```