UNIVERSITÀ DI PADOVA  FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria dell'Informazione

Scuola di Dottorato di Ricerca in Ingegneria dell'Informazione
Indirizzo: Ingegneria Informatica ed Elettronica Industriali
XX Ciclo

Software and Control Architectures for Autonomous and Racing Vehicles

Dottorando: LUCA BURELLI

Supervisore: Prof. RUGGERO FREZZA

Direttore della Scuola: Prof. MATTEO BERTOCCO

Padova, 31 Luglio 2008

Ai miei genitori.

"But remember, the brick walls are there for a reason. The brick walls are not there to keep us out. The brick walls are there to give us a chance to show how badly we want something. Because the brick walls are there to stop the people who don't want it badly enough. They're there to stop the *other* people."[1]

– prof. Randy Pausch, *Last Lecture at CMU*

[1]Ricordate, i muri sono lì per un motivo. I muri non sono fatti per tenerci lontani. I muri sono fatti per darci la possibilità di capire quanto veramente vogliamo qualcosa: i muri servono per fermare le persone che non la vogliono abbastanza. Servono per fermare *gli altri*.

Sommario

Questo lavoro presenta i risultati ottenuti in alcuni progetti di ricerca che sono stati avviati all'Università di Padova nel campo dei sistemi di controllo per veicoli, e in particolare delle loro architetture software.

Inizialmente verrà illustrata una raccolta di algoritmi che sono attualmente allo stato dell'arte per i vari sistemi che compongono un veicolo autonomo, e saranno messe in luce le possibili scelte architetturali per lo sviluppo del software di controllo di un dispositivo di tale complessità. Per meglio comprendere le scelte che sono state fatte durante questi progetti, sarà inoltre necessario approfondire alcuni dettagli tecnici; essi si riveleranno molto interessanti per capire il lavoro che al giorno d'oggi viene dato per scontato da chi non sia uno sviluppatore di *middleware*.

All'altro estremo dello spettro delle possibili applicazioni dell'elettronica nei veicoli, sarà presentato un dispositivo per la registrazione ed elaborazione dati per veicoli da gara. Lo sviluppo di questo oggetto compatto ha coinvolto scelte in diversi campi dell'ingegneria, dalla meccanica all'elettronica all'informatica. Nonostante possa sembrare un'applicazione completamente differente, dalla discussione emergeranno alcuni punti di contatto tra i progetti presentati.

L'attività a monte di questi progetti copre svariate problematiche di ricerca applicata nel campo delle tecnologie dell'informazione. Grazie a questo lavoro sarà possibile trasportare sui veicoli reali gli algoritmi innovativi sviluppati per l'ambiente *automotive* presso l'Università di Padova, in modo tale da poterli confrontare con tutte le non idealità inevitabilmente trascurate dalle simulazioni.

Abstract

This work presents the achievements that have been obtained in a number of ongoing research projects, at the University of Padova, in the field of automotive software and control architectures.

An overview of current state-of-the-art algorithms for the various elements of an autonomous vehicle will be given, along with a discussion at the current software options available for designing such a complex device. To fully understand the choices that have been made during the development, it will be necessary to show also a number of technical implementation details; these provide interesting insight into the background work which is nowadays considered as given by anybody who is not a middleware developer.

On the other end of the automotive electronics spectrum, a data logger and co-processor for racing vehicles will be presented. This compact device required interdisciplinary design decisions in both the mechanical, electronics and software engineering fields. Despite being a radically different application, a few key convergence points will emerge from the discussion.

The activity behind these projects covers several issues of applied research within the information technology tradition, and paves the way for the experimentation on live devices, with all their not-really-ideal behaviors, of advanced algorithms in the automotive fields at the University of Padova.

Contents

Chapter 1

Introduction

Automotive electronics has seen an explosive evolution. Functions that were handled by stand-alone systems only a few years ago are now part of a complex networked system of great intricacy and pronounced internal reciprocal effects. Engine control, vehicle safety, driving assistance, multimedia features—what was once considered a luxury option is now standard issue in every vehicle class. Nowadays, in a car these functions are handled by more than a dozen dedicated micro-controllers, and electronic components account for up to 25 % of the overall vehicle production cost.

The use of electronic components combined with mechanical, electrical, or hydraulic systems offers numerous benefits in areas such as reliability, cost, weight and space. Advances in electronics design and integration has pushed toward the migration from mechanical and hydraulic servo systems to systems commanded by "smart" computational units, at lower and lower costs. This also means, however, that vehicle (and passenger) safety is now even more in the hands of the software developer. From the choice of algorithms, to their implementation, and finally to the overall interconnection of these parts, everything must be both error-free and error-resilient.

Advances in computing power and electronics have also lowered the entry barrier for researchers: algorithms that were once impractical on dedicated hardware are now evaluated in real-time by commodity personal computers. New research efforts in the field of autonomous vehicle control have been encouraged by the recent DARPA Grand Challenge initiative.

In 2005 the University of Padova joined this trend by starting an

autonomous vehicle project, with the aim of studying vehicle control algorithms and the underlying software architectures. This project has been successful, developing a complete software structure and a suite of algorithm implementations for autonomous vehicles.

Continuing the research in the field of automotive electronics and software systems, a number of other applications have been studied; in this Thesis, a complete real-time data logger/processor solution will be described in detail.

1.1 Structure of the Thesis

This work is divided in two parts.

Part I will deal with the algorithmic areas of the current research, presenting the algorithms that have been employed in the Autonomous Vehicle project at the University of Padova. Chapter 2 will serve as an introduction to this field. The following two chapters will explain the various algorithms in detail, Chapter 3 focusing more on the control systems and Chapter 4 on the algorithms used for world sensing and for high-level reasoning.

Part II deals with the software aspects of some of the implementations that have been completed for this research. Again, after an introduction, in Chapter 5, that will provide some background on the discussed topics, Chapter 6 will detail what has been used as the backbone of the Autonomous Vehicle project, and how that has been extended and customized to this specific application. Chapter 7 will provide another example of software-hardware interaction, specifically, a real-time data acquisition and processor project.

Part I

Algorithms for Autonomous Vehicles

Chapter 2

Introduction

When focusing the attention to the leading application of the design of an autonomous vehicle, it can be easily understood and will be stated clearly in next chapter, that the starting point for the unmanned system to behave properly is to sense the environment around. In other terms, the intelligence of the system has to be provided as a first issue with the capability not only of sensing its own state, but also of understanding that of the surroundings.

Ideally, this can be done simply by instrumenting the system with a wide variety of sensors, according to the application needs and the envisaged interactions between the system and the world it is interfacing. This aspect, though, raises issues of two kinds:

- in the first instance, sensors can be affected by electronic or measurement noise, errors, drift in time, and many other problems related to the communication of the measured quantity (e.g.: lossy communication link, data packet drop, or desynchronization between the transmitter and the receiver);

- secondly, but often more importantly, it may happen that the specific variable is not a measurable quantity, be it for difficulty in retrieving its measure within the constraints imposed by the diagnostics structure or for the the lack of the diagnostics device itself.

Remarkably, the solution to both these issues can be obtained by resorting to the same methodology, namely the estimate of the required variable supported by the knowledge provided by the available data. The

rationale behind the concept is to implement a predictor corrector procedure, where a specific quantity value is first estimated by prior knowledge on the system and the environment, and then updated by using the data gathered from available measurements. The commonly agreed reference for this family of algorithms is the popular Kalman filter, whose details are briefly recalled in the following section.

2.1 Kalman Filtering background

The Kalman filter [1] consists in a set of equations that provide the estimation of the state of a stochastic linear process. In particular, the Kalman filter is the optimal linear estimator, having the following properties:

- it is a data processing algorithm, combining measurement data and a priori estimates in order to minimize the statistical error;
- it includes all available knowledge on the process and the measurements;
- it is recursive, not requiring the knowledge of all the past each time a new sample is acquired;
- it refers to gaussian noise, which is an overall good approximation of measurement and process noises.

The process of interest can be described in state space form as

$$\begin{aligned} x(t+1) &= Fx(t) + Gu(t) + w(t) \\ z(t) &= Hx(t) + v(t), \end{aligned}$$

being x and z respectively the process state and the output measurement, (F, G, H) the system matrices, and be the system affected by both process measurement w and measurement noise v. Moreover, the noise variables are considered as zero mean, independent (one on the other), and gaussian, with

$$\begin{aligned} w &\in \mathcal{N}(0, Q), \quad Q = \mathbb{E}\left[ww^T\right] \\ v &\in \mathcal{N}(0, R), \quad R = \mathbb{E}\left[vv^T\right]. \end{aligned}$$

Be the a priori and a posteriori estimates of $x(t+1)$ respectively $\hat{x}_-(t+1)$ e $\hat{x}_+(t+1)$, the correspondent errors with respect to the real value of the state $x(t+1)$ can be introduced as

$$\begin{aligned} e_-(t+1) &= x(t+1) - \hat{x}_-(t+1) \\ e_+(t+1) &= x(t+1) - \hat{x}_+(t+1). \end{aligned}$$

The rationale behind the procedure is the minimization of the a posteriori prediction error variance

$$P_+(t+1) = \mathbb{E}\left[\hat{x}_+(t+1)\hat{x}_+(t+1)^T\right],$$

by defining the a posteriori estimate $\hat{x}_+(t+1)$ as a linear combination of the a priori estimate $\hat{x}_-(t+1)$ and the innovation given by the difference between the measurement and its expected value $z(t+1) - H\hat{x}_-(t+1)$, that is

$$\hat{x}_+(t+1) = \hat{x}_-(t+1) + K(t+1)\left[z(t+1) - H\hat{x}_-(t+1)\right],$$

where the weighting matrix $K(t+1)$ assumes the name of Kalman gain.

After some calculations, the error variance $P_+(t+1)$ results (omitting all time dependence to simplify the notation)

$$P_+(\cdot) = P_-(\cdot) - K(\cdot)HP_-(\cdot) - P_-(\cdot)H^TK(\cdot)^T + K(\cdot)HP_-(\cdot)H^TK(\cdot)^T + K(\cdot)RK(\cdot)^T,$$

whose minimization with respect to $K(t+1)$ leads to the following expression for the Kalman gain

$$K(t+1) = P_-(t+1)H^T\left(HP_-(t+1)H^T + R\right)^{-1}.$$

It is worth noticing that when $R \approx 0$ it follows that $K \approx H^{-1}$ implying $\hat{x}_+(t+1) \approx z(t+1)$ (measurements are trustworthy); conversely, when $P_-(t+1) \approx 0$ it follows that $K \approx 0$ and $\hat{x}_+(t+1) \approx \hat{x}_-(t+1)$ (measurements are not trustworthy).

The algorithm implements a two step procedure:

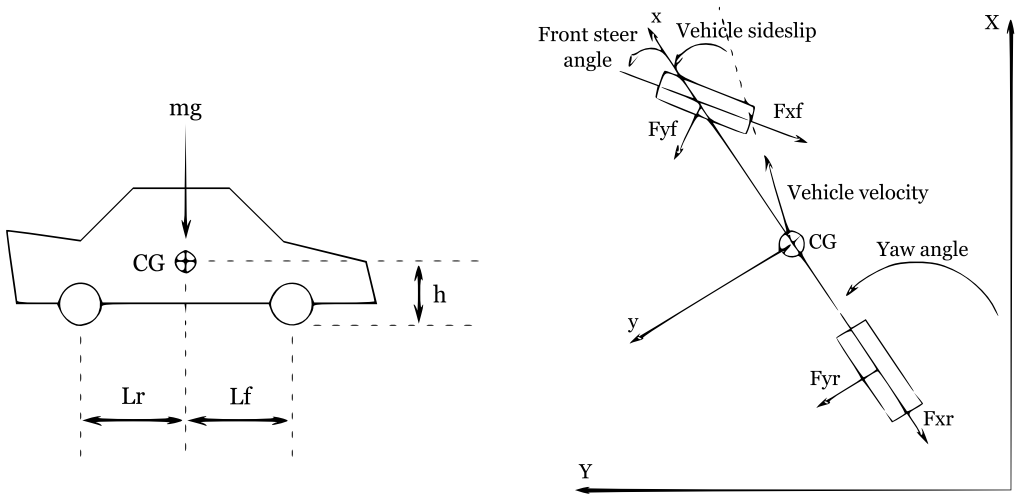
1. prediction phase:
   (a) the a priori estimate is computed:
   $$\hat{x}_-(t+1) = F\hat{x}(t) + Gu(t);$$
   (b) the a priori prediction error variance is computed:
   $$P_-(t+1) = FP_-(t)F^T + Q;$$
2. correction phase:
   (a) the Kalman gain is computed from $P_-(t+1)$:
   $$K(t+1) = P_-(t+1)H^T \left(HP_-(t+1)H^T + R\right)^{-1}.$$
   (b) the a posteriori estimate is obtained as a linear combination:
   $$\hat{x}_+(t+1) = \hat{x}_-(t+1) + K(t+1)\left[z(t+1) - H\hat{x}_-(t+1)\right],$$
   (c) the error variance is updated:
   $$P_+(t+1) = \left(I - K(t+1)H\right)P_-(t+1).$$

2.2 The bicycle model

In the following discussion, we will make use of the *bicycle model*, depicted in Figure 2.1. This is a widely-used approximation (see [2]) for the cinematic motion of a car-like vehicle, which makes the assumptions that

1. when turning, the slip angles on the inside and outside wheels are approximately the same;
2. the effect of the vehicle roll is negligible.

These constraints hold well for most typical driving situations, notably excluding rally driving conditions (high speed, high slip angle).



(a) Side view (b) Top view

Figure 2.1: The bicycle model

Given these assumptions, it is thus possible to collapses left and right tires into one, reducing the car to a bicycle. Furthermore, assumption 2 guarantees the stability in the lateral dynamics, reducing the dynamics only on the forward axis of the vehicle.

This model has the nice property of being *differentially flat*: once the evolution of the state variables over time is known, it is possible to invert the dynamic equations and recover the originating control inputs. This is quite useful because, given a (feasible) trajectory, the control inputs for such a model can be readily calculated.

Chapter 3

Control systems

When in the Eighties the microcomputer era advanced, it started to revolutionize not only the way of designing already existing or new products, but also the technology content of the products themselves, suggesting the possibility of introducing intelligence in systems, for example in the form of advanced control algorithms.

In the automotive, this new perspective have been supported by always more stringent constraints over the car system efficiency, from the energetic and environmental point of view (oil consumption and pollution reduction), from the mere performance (in terms of speed, acceleration, ...), and from the comfort (smoothness in the drive, support in emergency or critical situations).

These issues in recent times have been transferred to what is called *drive-by-wire*[1], meaning the possibility of replacing the traditional mechanical or hydraulic links between the vehicle commanding drives and the parts that physically execute these commands, with an intelligent system that retrieves all the command inputs and acts onto the local devices to produce the desired effect. The main advantage and rationale behind the idea is that in doing so the several independent devices operating in the vehicle and determining its behavior (brakes, engine, steer...) can act in a coordinated way so as to reach a behavior regime unreachable by the collection of isolated actions: In practical terms, this translates in enhanced drive security, better performance, and overall car system optimization.

[1] More informations and pointers on `http://en.wikipedia.org/Drive_by_wire`.

The main ingredients for a drive-by-wire systems are:

- a set of sensors, to gather information from the environment and the system itself, and measuring a wide range of quantities, from the speed and the acceleration of the vehicle to the temperature or the presence of rain;
- a microprocessor, representing the intelligence of the system;
- a set of actuators.

The drive-by-wire paradigm has been introduced in this context to draw the focus to what have been identified with the intelligence of the system, meaning the control algorithms supervising and determining the vehicle behavior according to the external conditions and disturbances (e.g. road and weather conditions). And indeed, the attention of engineers and academicians from the control and system community is increasingly addressing applications related to the automotive field. Just to give an example, recently the IEEE Control Systems Magazine, which is traditionally careful on technological as well as scientific state of the art issues in control, has published a special issue on control techniques and methodologies applied to motorcycle design [3].

Pushing just a little forward the imagination along these tracks, it is not distant to set the basis for research programmes for the autonomous vehicle, such as the DARPA Grand Challenge: In this case, a crucial role is played by the sensors to "view" everything, from the road conditions, to the planned trajectory, the the vehicle conditions. In the remainder of the part, several algorithms will be presented, all devoted to infer the conditions in which the vehicle is acting. The reference to the leading example of the autonomous vehicle allows to discuss these aspects, without loss of generality.

3.1 Position and attitude

The main positioning system used in several autonomous car projects is the now-ubiquitous GPS/IMU combination [4] [5] [6].

The GPS (Global Positioning System) is a positioning system based on satellite data. The system is based on the measurement of the time of flight of a radio signal traveling from the satellite to the receiver at

the unknown location: from at least four[2] of these measurements and the knowledge on the exact position of the satellite sources, geometric methods such as the trilateration or the triangulation procedures allow to determine the unknown location.

The IMU (Inertial Measurement Unit) uses a combination of accelerometers and gyroscopes in order to detect the current rate of acceleration and changes in rotational attributes, including pitch, roll and yaw. This information is then integrated by a processing unit so as to produce an estimate of the IMU current position and velocity.

It is very well known that these two technologies complement each other: GPS measures are stable, but the update rate is low and furthermore subject to outages; the data acquired from inertial sensors, on the other hand, is continuously available at a high rate, but has fundamental long-term drift issues. Thus, the combination of the two can provide a very good absolute, high-speed estimate of the current position. It is in fact the main method used for navigation purposes in aircrafts, where INS systems complement the GPS measurements and provide dead reckoning [3] when the GPS signal is not available. These tactical-grade devices are not suitable for all applications, though, because of their high costs (up to tens of thousands of Euros) [7].

3.1.1 Notation and reference systems

The Cartesian coordinate frame of reference used by the GPS is called Earth-Centered, Earth-Fixed (ECEF). ECEF uses 3-D XYZ coordinates (in meters) to describe the location of a GPS user or satellite. The term "Earth-Centered" comes from the fact that the origin of the axis (0,0,0) is located at the mass center of gravity (determined through years of tracking satellite trajectories). The term "Earth-Fixed" implies that the axes are fixed with respect to the earth (that is, they rotate with the Earth).

This reference system is not very useful, however, since most taskS require location of a point on (or close to) the surface of the Earth. For de-

[2]Counter-inuitively, four signals are needed for triangulation. This is because in addition to the X, Y and Z coordinates, the unknown delta between the receiver clock and the GPS constellation's must be estimated.

[3]Dead reckoning is a procedure that estimates one's current position based upon a previously determined position, called fix, and advancing that position based upon known speed, elapsed time, and course.

scribing that, the old Longitude, Latitude and Altitude (LLA) reference is much better suited. This representation relies on a "simple" ellissoid representation of the Earth's much more complex shape.

For global applications, the geodetic reference (datum) used for GPS is the World Geodetic System 1984 (WGS84) [8]. This ellipsoid has its origin coincident with the ECEF origin. The X-axis crosses the Greenwich meridian (where longitude $\lambda = 0$ degrees) and the XY plane coincides with the equatorial plane (latitude $\varphi = 0$ degrees). Altitude is described as the perpendicular distance above the ellipsoid surface.

Note that there are two definitions of latitude: the usual meaning is more precisely referred to by the term *geodetic latitude*, and takes into account the flattening parameter of the ellipsoid describing the Earth. *Geocentric latitude* (indicated with φ'), instead, considers the Earth as a perfect sphere and measures angles from that assumption.

Finally, latitude and longitude are useful to rotate the ECEF frame to "local" coordinates, so that the measurement axes are pointing East, North and Up (ENU). This is also sometimes known as the Local Tangent Plane (LTP) reference.

Converting between coordinate systems The parameters specified by WGS84 for the ellipsoid are the following:

$$a = 6378137 \qquad b = a(1-f) = 6356752.31424518 \qquad f = 298.257223563$$

To obtain LLA coordinates from their ECEF representation, first compute longitude as

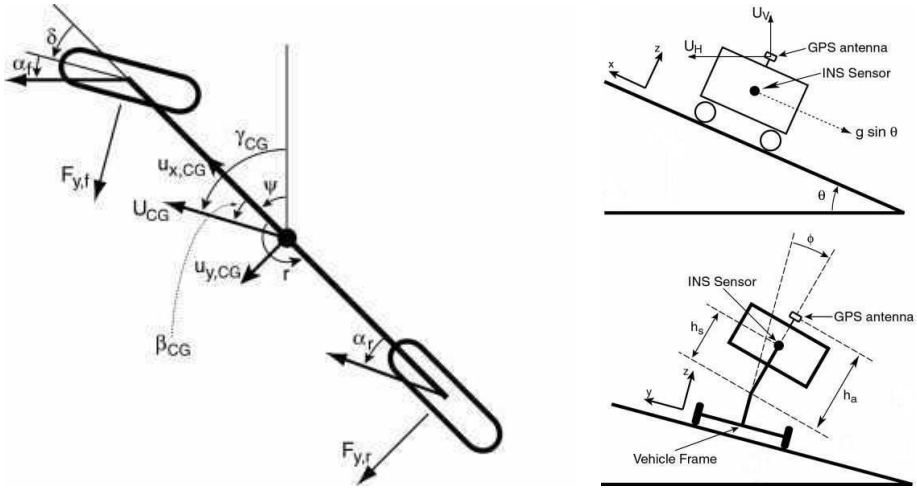
$$\lambda = \arctan \frac{Y}{X} \tag{3.1}$$

Geodetic latitude is then expressed by

$$\varphi = \arctan \frac{Z + (a^2 - b^2)(\sin\theta)^3}{p - (a^2 - b^2)(\cos\theta)^3} \tag{3.2}$$

where the auxiliary values p and θ are defined as

$$p = \sqrt{X^2 + Y^2} \qquad \text{and} \qquad \theta = \arctan \frac{Za}{pb}$$



(a) Top (b) Side

Figure 3.1: Vehicle reference axes. Roll, pitch and jaw angles are shown w.r.t. the road as (ϕ, θ, Ψ).

while geocentric latitude is simply obtained by

$$\varphi' = \arctan \frac{Z}{p} \quad . \tag{3.3}$$

Finally, to convert from ECEF coordinates to ENU, only a simple rotation $\mathbf{R}_{\text{ENU}}^{\text{ECEF}}$ is required:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}^{\mathbf{ENU}} = \mathbf{R}_{\text{ENU}}^{\text{ECEF}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}^{\mathbf{ECEF}} \tag{3.4}$$

where

$$\mathbf{R}_{\text{ENU}}^{\text{ECEF}} = \begin{bmatrix} -\sin\lambda & \cos\lambda & 0 \\ -\sin\varphi'\cos\lambda & -\sin\varphi'\sin\lambda & \cos\varphi' \\ \cos\varphi'\cos\lambda & \cos\varphi'\sin\lambda & \sin\varphi' \end{bmatrix}$$

In the following talk, vectors and measurements relative to the ECEF reference frame will be denoted with the superscript e (as in v_x^{e}), while items referring to the body frame will have the superscript b (as in v_x^{b}); some specific axes will be referred to similarly.

Figure 3.1 shows the bicycle model with the relevant vectors that will be used in the discussion below.

For the GPS/INS integration, several possible algorithms have been investigated; in particular, the Ryu-Gerdes [9] and Qi-Moore [10] algorithms have been reviewed.

3.1.2 Ryu-Gerdes algorithm

The algorithm Ryu and Gerdes proposed in [9] uses a bicycle model for the vehicle, with an IMU and two GPS antennas as the inputs to the model. This configuration is devised to obtain absolute velocity and attitude measurements, so that drift errors on IMU sensors can be estimated and removed from the measurements, to achieve better accuracy.

The yaw angle Ψ, for example, is measured directly from the orientation of the vector connecting the reconstructed position of the two GPS antennas, relative to the ENU x-y plane, and can be modeled with a simple additive noise:

$$\Psi_m^{GPS} = \Psi + \text{noise}$$

The IMU measurements, on the other hand, are of the yaw rate $\dot{\Psi}$, and incur significant but relatively stable bias $\dot{\Psi}_{bias}^{IMU}$:

$$\dot{\Psi}_m^{IMU} = \dot{\Psi} + \dot{\Psi}_{bias}^{IMU} + \text{noise}$$

This makes it easy to derive a first-order linear model for combining both measurements:

$$\begin{bmatrix} \dot{\Psi} \\ \dot{\Psi}_{bias} \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \Psi \\ \Psi_{bias} \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \dot{\Psi}_m^{IMU} + \text{noise} \tag{3.5}$$

When GPS attitude measurements are available, the estimate is updated with:

$$\Psi_m^{GPS} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \Psi \\ \Psi_{bias} \end{bmatrix} + \text{noise} \tag{3.6}$$

A Kalman filter can be constructed from (3.5) and (3.6), that can be used to obtain the estimated vehicle yaw angle.

The GPS can also provide velocity information, so another Kalman filter can be devised to filter the data coming from IMU linear accelerometers and estimate biases. In fact, the measured acceleration is related

to the velocity at the point where the sensor is placed by

$$\begin{aligned} a^{\mathrm{b}}_{x,m} &= \dot{v}^{\mathrm{IMU}}_{x} - \dot{\Psi} \cdot v^{\mathrm{IMU}}_{y} + a^{\mathrm{b}}_{x,bias} + \text{noise} \\ a^{\mathrm{b}}_{y,m} &= \dot{v}^{\mathrm{IMU}}_{y} + \dot{\Psi} \cdot v^{\mathrm{IMU}}_{x} + a^{\mathrm{b}}_{y,bias} + \text{noise} \end{aligned} \tag{3.7}$$

where

$$\begin{aligned} a^{\mathrm{b}}_{x,m},\ a^{\mathrm{b}}_{x,bias} &\quad \text{longitudinal accelerometer measurement and bias} \\ v^{\mathrm{IMU}}_{x} &\quad \text{longitudinal velocity at sensor position} \\ a^{\mathrm{b}}_{y,m},\ a^{\mathrm{b}}_{y,bias} &\quad \text{lateral accelerometer measurement and bias} \\ v^{\mathrm{IMU}}_{y} &\quad \text{lateral velocity at sensor position} \end{aligned} \tag{3.8}$$

These velocities can be derived from one of the GPS antenna's measurements, but first sideslip angle (β_{CG}) must be computed by measuring the angle between the GPS velocity vector $\mathbf{v}^{\mathrm{GPS}}_{m}$ and the estimated vehicle yaw Ψ. The GPS velocities, referred to the vehicle frame, can then be expressed as

$$\begin{aligned} v^{\mathrm{b}}_{x,m} &= \left\|\mathbf{v}^{\mathrm{GPS}}_{m}\right\| \cos\beta_{CG} \\ v^{\mathrm{b}}_{y,m} &= \left\|\mathbf{v}^{\mathrm{GPS}}_{m}\right\| \sin\beta_{CG} \end{aligned} \tag{3.9}$$

Now, if the GPS antenna providing velocity measurements and IMU are placed exactly one above the other, the velocities seen by the sensor are simply

$$\begin{aligned} v^{\mathrm{GPS}}_{x,m} &= v^{\mathrm{IMU}}_{x,m} + \text{noise} \\ v^{\mathrm{GPS}}_{y,m} &= v^{\mathrm{IMU}}_{y,m} + \text{noise} \end{aligned} \tag{3.10}$$

Using (3.10), the final Kalman filter for analyzing the linear accelerometer biases can thus be written as

$$\begin{bmatrix} \dot{v}^{\mathrm{IMU}}_{x} \\ \dot{a}^{\mathrm{b}}_{x,bias} \\ \dot{v}^{\mathrm{IMU}}_{y} \\ \dot{a}^{\mathrm{b}}_{y,bias} \end{bmatrix} = \begin{bmatrix} 0 & -1 & r & 0 \\ 0 & 0 & 0 & 0 \\ -r & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} v^{\mathrm{IMU}}_{x} \\ a^{\mathrm{b}}_{x,bias} \\ v^{\mathrm{IMU}}_{y} \\ a^{\mathrm{b}}_{y,bias} \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a^{\mathrm{b}}_{x,m} \\ a^{\mathrm{b}}_{y,m} \end{bmatrix} + \text{noise} \tag{3.11}$$

The state vector can be updated by the GPS measurements, when avail-

able, by

$$\begin{bmatrix} v_{x,m}^{\mathrm{GPS}} \\ v_{y,m}^{\mathrm{GPS}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} v_x^{\mathrm{IMU}} \\ a_{x,bias}^{\mathrm{b}} \\ v_y^{\mathrm{IMU}} \\ a_{y,bias}^{\mathrm{b}} \end{bmatrix} + \text{noise} \tag{3.12}$$

If the road is not flat, but has significative roll and/or bank grades, a more complex solution must be adopted.

Three different Kalman filters are described: one for yaw angle, one for lateral velocities, and one for roll and pitch angles. Yaw is computed from the GPS heading information (integrated with the relevant angular acceleration by the IMU), while roll and pitch angles are computed from the differential position of the two GPS antennas, and also from the current horizontal and vertical velocity components.

3.1.3 Qi-Moore algorithm

The Qi-Moore algorithm [10], instead, uses a Direct Kalman Filter to deal with model nonlinearities, resulting in a low-order linear filter that has very good performance. This filter estimates the current position, speed and current linear/angular accelerometer offsets.

Skew-symmetric matrices indicated with Ω are matrices of the form

$$\Omega = \begin{bmatrix} 0 & -\omega_z & -\omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix}$$

which is the form used to simplify notation when dealing with angular velocities, hiding cross products.

The current IMU accelerometer measurement $\mathbf{a}_m^{\mathrm{b}}$ can be expressed in the ECEF frame as $\mathbf{a}_m^{\mathrm{e}} = \mathbf{R}_{\mathrm{b}}^{\mathrm{e}}\mathbf{a}_m^{\mathrm{b}}$. The current angular velocity, $\Omega_{\mathrm{e}}^{\mathrm{b}}$, is the skew symmetric matrix associated with $\omega_{\mathrm{e}}^{\mathrm{b}} = \omega_{\mathrm{e,m}}^{\mathrm{b}} - \mathbf{R}_{\mathrm{e}}^{\mathrm{b}}\omega_{\mathrm{ie}}^{\mathrm{e}}$: from the actual measurement $\omega_{\mathrm{e,m}}^{\mathrm{b}}$, the always present Earth angular velocity $\omega_{\mathrm{ie}}^{\mathrm{e}}$ must be removed.

The continuous time dynamical system equations are of the form

$$\begin{bmatrix} \dot{\mathbf{p}}^{\mathrm{e}} \\ \dot{\mathbf{v}}^{\mathrm{e}} \\ \dot{\mathbf{R}}_{\mathrm{b}}^{\mathrm{e}} \end{bmatrix} = \begin{bmatrix} \mathbf{v}^{\mathrm{e}} \\ \mathbf{R}_{\mathrm{b}}^{\mathrm{e}}\mathbf{a}^{\mathrm{b}} - 2\Omega_{\mathrm{ie}}^{\mathrm{e}}\mathbf{v}^{\mathrm{e}} + \mathbf{g}^{\mathrm{e}}(\mathbf{p}^{\mathrm{e}}) \\ \mathbf{R}_{\mathrm{b}}^{\mathrm{e}}\Omega_{\mathrm{eb}}^{\mathrm{b}} \end{bmatrix} \tag{3.13}$$

where $\mathbf{R}_{\mathrm{b}}^{\mathrm{e}}$ is the rotation matrix from ECEF (e-frame) to body frame (b-frame), $\mathbf{g}^{\mathrm{e}}(\mathbf{p}^{\mathrm{e}})$ is the gravity vector (which depends on the current position $\mathbf{p}^{\mathrm{e}}$),

Once the system (3.13) is solved, the vehicle attitude is represented by the matrix $\mathbf{R}_{\mathrm{b}}^{\mathrm{e}}$, and can be decomposed in the three independent yaw, pitch and roll rotations.

The article then derives a discrete time representation of the model shown in (3.13), noting that the attitude can be obtained by directly integrating part of the discretized model (3.13), while using the estimates of $\mathbf{p}^{\mathrm{e}}$ and $\mathbf{v}^{\mathrm{e}}$ obtained from a position-only Direct Kalman Filter.

The resulting state space is $\xi = \begin{bmatrix}\mathbf{p}^{\mathrm{e}\prime} & b & \dot{\mathbf{p}}^{\mathrm{e}\prime} & \dot{b} & \Delta\mathbf{a}^{\mathrm{e}\prime} & \Delta\dot{\mathbf{a}}^{\mathrm{e}\prime}\end{bmatrix}'$, where b and $\dot{b}$ are the GPS clock range bias and drift, and $\Delta\mathbf{a}^{\mathrm{e}\prime}$ and $\Delta\dot{\mathbf{a}}^{\mathrm{e}\prime}$ are the acceleration bias and drift, expressed in the ECEF reference frame via the relations

$$\begin{aligned}\Delta\mathbf{a}^{\mathrm{e}}(t+1) &= \mathbf{R}_{\mathrm{e}}^{\mathrm{b}}(t+1)\Delta\mathbf{a}^{\mathrm{b}}(t+1)\\ \Delta\dot{\mathbf{a}}^{\mathrm{e}}(t+1) &= \mathbf{R}_{\mathrm{e}}^{\mathrm{b}}(t)\Delta\Omega_{\mathrm{eb}}^{\mathrm{b}}(t)\left(\mathbf{a}_{m}^{\mathrm{b}}(t+1)+\Delta\mathbf{a}^{\mathrm{b}}(t+1)\right)\end{aligned} \tag{3.14}$$

The matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in the obtained DKF are not fixed, because they take into account the presence or absence of GPS measurements for the current step. If the sample time of the GPS is ΔT which is N times the sample time δT of the INS, the following structure is defined:

$$\mathbf{A}(t) = \begin{bmatrix}\mathbf{A}_{11} & \mathbf{A}_{12}(t)\\ \mathbf{0} & \mathbf{A}_{22}(t)\end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix}\mathbf{0}_{4\times 3}\\ \mathbf{I}_{3\times 3}\delta T\\ \mathbf{0}_{4\times 3}\end{bmatrix}$$

where

$$\mathbf{A}_{11} = \begin{bmatrix} \mathbf{I}_{4\times4} & \mathbf{I}_{4\times4}\delta T \\ \mathbf{0}_{4\times4} & \mathbf{I}_{4\times4} \end{bmatrix}$$

$$\mathbf{A}_{12}(t) = \begin{bmatrix} \mathbf{0}_{4\times3} & \mathbf{0}_{4\times3} \\ \mathbf{I}_{3\times3}\Delta T & \mathbf{I}_{3\times3}\Delta T^2 \\ \mathbf{0}_{1\times3} & \mathbf{0}_{1\times3} \end{bmatrix} \text{ when } t = kN, \mathbf{0}_{8\times6} \text{ otherwise}$$

$$\mathbf{A}_{22}(t) = \begin{bmatrix} c_1\mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & c_2\mathbf{I}_{3\times3} \end{bmatrix} \text{ when } t = kN, \mathbf{0}_{6\times6} \text{ otherwise}$$

$$\mathbf{C}(t) = \begin{bmatrix} \mathbf{I}_{8\times8} & \mathbf{0}_{8\times6} \end{bmatrix} \text{ when } t = kN, \mathbf{0}_{8\times14} \text{ otherwise}$$

In this formulation, c_1 and c_2 are the forgetting factors for the accelerometer bias and drift estimates (so $0 < c_1, c_2 \leq 1$).

A reduced-order DKF can be obtained by removing GPS clock range bias and drift from the estimation state vector. The resulting error in the estimation is small, especially considering that "smart GPS" antennas do already employ GPS clock rate estimation and take this factor into account to provide better location estimates.

The implemented algorithm

Details on the implemented algorithm, which merges both of these ideas, are given in Section 6.3.2.

3.2 Horizon detection

Detecting the horizon from the camera images provides an absolute reference measurement for the vehicle roll angle. This result can be obtained through a procedure [11] that begins from the acquisition of a video stream from a camera, to proceed with a series of post-processing steps, namely:

1. edge detection: edges characterize object boundaries whose detection represents a canonical problem in image processing.

2. Hough transform: the Hough transform in its original formulation is intended for the recognition of rectilinear segments in an image, and in its later formulation the procedure has been extended to

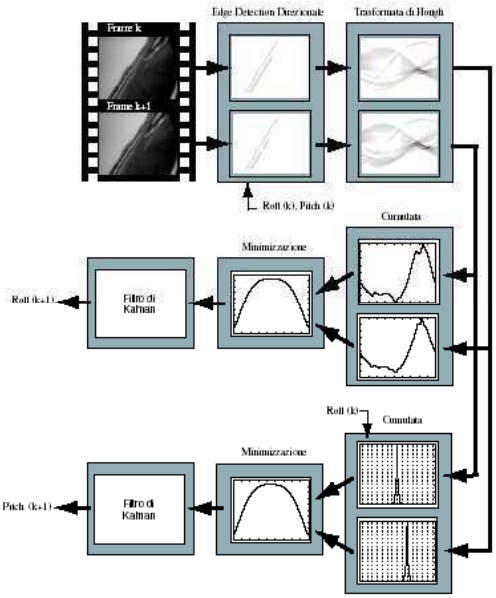


Figure 3.2: Procedure for the horizon estimation.

the recognition of any curve and is based on the validation of a recognition hypothesis;

3. Roll and Pitch estimation and integration.

The flowchart of the procedure is given in Figure 3.2, while some details over the mentioned algorithms are now in order and given in the following.

3.2.1 Canny Edge Detection

The benefit of the application of edge detection to an image of interest is that it significantly reduces the amount of data and filters out application-specific useless information, while preserving an important set of structural properties (interesting features) in an image. Edges in images are areas characterized by strong intensity contrasts, meaning a jump in the image intensity function detected between adjacent pixels.

The leading criteria in the derivation of the Canny algorithm are:

- good detection: low probability of not marking real edge points, and falsely marking non-edge points;

- good localization: the detected edge should be close to the center of the true edge;
- edge detection uniqueness: only one response to a single edge, obtained by explicit elimination of multiple responses.

In this sense, the Canny edge detection algorithm [12] is known to many as the optimal edge detector. In general, the application of the Canny edge detector is preceded by the image convolution with a gaussian filter, producing a slightly more blurred image but not affected by a single noisy pixel, in order to gain in noise reduction. Then, the main part of the algorithm is the directional edge detection, obtained by resorting to an oriented Gaussian filter.

3.2.2 Hough Transform

The Hough transform, originally patented by P.V. Hough in 1962 and later much generalized and improved [13] [14], is used in this context to filter out all edges that do not represent straight lines. The algorithm consider sets of points belonging to the edges detected in an image, and deduces the geometric parameters of the straight lines that better fit the chosen points. For computational reasons, the polar representation of lines is used,

$$l_{(\sigma,\rho)} = \{(x,y)|\rho = x\cos\sigma + y\sin\sigma\}\,, \tag{3.15}$$

being ρ the distance between the frame origin and the line, and σ its orientation (angle) with respect to the horizontal. Using this representation, the range of parameters is limited[4], since

$$0 \leq \sigma < \pi \quad \text{and} \quad -\frac{\sqrt{2}}{2}\left(\frac{h}{2}+\frac{w}{2}\right) \leq \rho \leq +\frac{\sqrt{2}}{2}\left(\frac{h}{2}+\frac{w}{2}\right),$$

(h,w) being the frame dimension.

The coordinates of each point of edge segments (x,y) serve as constants in Eq. 3.15, while searching for the pair (ρ,σ). These (ρ,σ) appears as sinusoids in the Hough parameter space, defining in this way a point-to-curve map between the cartesian image space and the Hough parameter space, that is the Hough transformation $\mathcal{H}_(\rho,\sigma)$ for straight

[4]In contrast with the more common line representation $y = mx + q$, where the angular coefficient varies to infinity in correspondence to vertical lines.

lines. In the Hough parameter space points that belong to straight lines yield curves which intersect at a common (ρ, σ) point.

The accumulated values defined by:

$$\begin{aligned} \mathcal{H}^{\rho}_{(}\sigma) &= \sum_{\rho} \mathcal{H}_{(}\rho, \sigma) \\ \mathcal{H}^{\sigma}_{(}\rho) &= \sum_{\sigma} \mathcal{H}_{(}\rho, \sigma), \end{aligned}$$

can also be computed.

In the application to horizon detection, ρ and σ are the y-offset in the image and the horizon angle.

3.2.3 Pitch, Roll Estimation

The measurements of roll and pitch (respectively ϕ and θ) and their variations (respectively $\Delta\phi$ and $\Delta\theta$) can be obtained by resorting to the difference of the norm of the accumulated Hough transforms (respectively $\mathcal{H}^{\rho}_{(}\sigma)$ and $\mathcal{H}^{\sigma}_{(}\rho)$). In other words, $\mathcal{H}^{\rho}_{(}\sigma)$ and $\mathcal{H}^{\sigma}_{(}\rho)$ can be compared from one frame to the next, and by minimizing the Euclidean distance between each of the cumulated transforms at time t and at time $t+1$ it is possible to calculate $\Delta\rho$ and $\Delta\sigma$, which are directly related to, respectively, roll and pitch variations.

Finally, for each parameter, a simple random-walk Kalman filter is used to update the estimate of the "real" value. The state of the model to be estimated consist in pitch and pitch variation and in roll and roll variation, taking into account the fact that pitch and roll are independent. Therefore, the whole system is basically formed by two independent subsystems:

$$\begin{aligned} \begin{bmatrix} \phi(t+1) \\ \Delta\phi(t+1) \end{bmatrix} &= F \begin{bmatrix} \phi(t) \\ \Delta\phi(t) \end{bmatrix} + w(t) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \phi(t) \\ \Delta\phi(t) \end{bmatrix} + w(t) \\ \begin{bmatrix} \tilde{\phi}(t) \\ \widetilde{\Delta\phi}(t) \end{bmatrix} &= H \begin{bmatrix} \tilde{\phi}(t) \\ \widetilde{\Delta\phi}(t) \end{bmatrix} + v(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{\phi}(t) \\ \widetilde{\Delta\phi}(t) \end{bmatrix} + v(t), \end{aligned}$$

where $\tilde{\phi}$ and $\widetilde{\Delta\phi}$ are the measurements of roll and roll variation, and

equivalently

$$\begin{bmatrix} \theta(t+1) \\ \Delta\theta(t+1) \end{bmatrix} = F \begin{bmatrix} \theta(t) \\ \Delta\theta(t) \end{bmatrix} + w(t) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta(t) \\ \Delta\theta(t) \end{bmatrix} + w(t)$$
$$\begin{bmatrix} \tilde{\phi}(t) \\ \widetilde{\Delta\phi}(t) \end{bmatrix} = H \begin{bmatrix} \tilde{\phi}(t) \\ \widetilde{\Delta\phi}(t) \end{bmatrix} + v(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{\phi}(t) \\ \widetilde{\Delta\phi}(t) \end{bmatrix} + v(t).$$

3.3 Maximum tire force

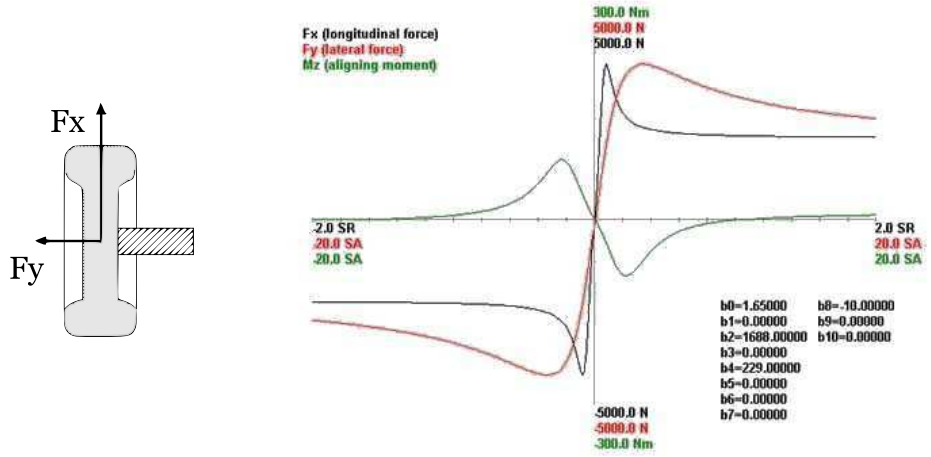
The interaction between road and tires is extremely complex, and an accurate description is impractical but in a very few corner cases (car races being one of these, where all the data about the tires, the road and the environment is known).

Unfortunately, an estimate of the maximum force tires can sustain while keeping contact with the terrain is essential to avoid maneuvers that can cause wheel slippage, and eventually the complete loss of control: The forces exerted on the tire as well as those impressed by the tire onto the ground are the origin of all vehicle behaviors, ranging from braking to steering, from accelerating to sideslipping. In general, from the representation point of view, these forces can be decomposed into longitudinal and lateral forces.

From the Seventies, to estimate these forces the Pacejka method [15] has been proposed: in his technique, some “magic formulas” are introduced that allow obtaining the longitudinal and lateral forces, starting from a set of parameters describing the road surface, the tire type, the normal forces acting onto the ground, and the angular and longitudinal slip of the tire.

The Pacejka method provides a set of curves (like the ones in Figure 3.3(b)) describing the interaction of the tire with the road surface: In particular, three curves are obtained related to longitudinal (forward) force F_x, lateral (sideways) force F_y (see Figure 3.3(a)), and aligning moment M_z (the torque felt at the steering wheel during a driving manoeuver). The independent variables are the slip ratio Sr, defined as the ratio between the wheel spin velocity and the ground velocity, and the slip angle Sa, that is the angle between the wheel heading and the actual vehicle velocity.

These curves can be described through a set of parameters, namely



(a) Tire forces decomposition

(b) A sample set of Paceijka curves

Figure 3.3: The Paceijka Tire-Road model.

the peak value of the curve D, a shape factor C determining the shape of the peak, shifting values S_h and S_v related to manufacture features of the tire. They all are presented in the following form, with y being F_x, F_y, or M_z, and u being Sa or Sr according to the quantity y of interest[5]:

$$y = D\sin(C\arctan(Bu - E(Bu - \arctan(Bu)))) + S_v;$$

the undefined parameters B, E, as well as C, D, S_h, and S_v, are functions depending on the load F_z and a set of road descriptors and their formulations differ if considering the F_x, F_y, or M_z equation.

A workable algorithm to calculate the slip coefficients, using least squares estimation, was presented in [16] and was used in this project.

3.4 Path planner

Path planning is a canonical problem in mobile robot applications in order to navigate an autonomous unit in the environment and avoid obstacles, be they fixed (structured environment) or mobile (other similar robots, people, automatic devices).

[5]In particular, lateral force and aligning moment depend on the slip angle, while the longitudinal force is related to the slip ratio.

In automotive applications and in the framework of autonomous vehicle design, the information about current vehicle position, nearby obstacles, and overall goal is used to generate a possible path for the vehicle to follow. Also, the path planner should output safe speed clues for the vehicle controller, since it can take into account a wider span of information. In other words, the problem can be stated as: *given a representation of the environment (map), and a set of aim-points to be reached, find a path suitable for the specific vehicle and provide a feasible trajectory to be followed*.

The practical solution to the problem proceeds through the following steps:

- extraction from the input map of the information useful for the path planning operation;
- given the map, planning of a suitable path from the starting point to the final destination;
- given the path, provide for every point along the trajectory feasible speed profile.

For the path planning problem, many approaches have been investigated, that can be classified at various level between exact and heuristic algorithms. The first type of approach tends to formally demonstrate (or not) the existence and find (if there is one) the exact solution to the path planning problem but at the same time tends to have high complexity. Conversely, heuristic procedures simplify the problem and the context of its definition (for example approximating the shapes of involved objects) and in some case may fail to find a solution although there is one. Other solutions try to find intermediate solutions capturing advantages from both worlds, and to do so are often organized into two (or more) step procedures.

All these methodologies make use of a further wide variety of techniques, from Voronoi diagrams [17], to potential fields [18] [19], to Freeman chains [20]. In particular, two methods are of interest in this framework, namely the potential fields approach and the Delayed D* algorithm.

The potential fields approach [18] describes the domain where the subject of interest in a similar fashion as the potential field generated

by an electric charge: according to the other charges moving in the area, the field can exert repulsive or attractive forces, which are simply the negative or positive gradient of the potential field. If thinking of the path planning problem as an obstacle avoidance problem in the first instance, it appears clear how the potential field approach allows a global representation of the space and the obstacles within. In this way, a first coarse planning can be produced (at global level) as a trajectory passing through minimum potential valleys whose boundaries are generated by the presence of obstacles and reproduce their shapes. The definition of this first-guess path is computationally simplified by choosing piecewise linear trajectories and a node-edge tree representation. The global planner finds the shortest collision-free path or the more convenient according to a heuristic cost function whose contributing terms are associated to the costs of nodes and edges composing the chosen path. As a second step, a local planner intervenes that modifies at local level the global path by adjusting path length and smoothness of motion while preserving the collision-free feature. This operation is performed within the neighborhood of the initial estimate. Interestingly, the tree formulation of the path allows the definition of forward tree (going from the starting point towards the final destination) and backward tree (going in the opposite direction), and the definition of states along the trajectory that are forward reachable. A possible end-to-end path (connecting start to finish) of feasible configurations (for the moving object attitude and orientation) is composed of forward reachable states.

The Delayed D* algorithm [21] employs a description of the path in terms of arcs, which allows an easy formulation of the path cost as the sum of the cost of the arcs encountered from the initial to the final position. It often happens that the initial path envisaged for achieving the goal position has to be corrected, so that a replanning operation is required. This procedure is usually carried out in an heuristic way because of computational convenience. The D* algorithms basically employ a smart strategy to perform the replanning, obtained by focusing the search for the path update on a restricted set of states whose contribution could be relevant for the determination of the new path. This is obtained by identifying two classes of states to be considered: those whose arc associated cost decreases, which appear as good candidate to be included in the path planning (lowering the total path cost), and those whose arc associated cost increases, which conversely need to be poten-

tially excluded from current path solution. Iterating this procedure at any change in arc cost completes the path replanning and consequently the path definition.

3.5 Vehicle Controller

The task of calculating the controls that keep a vehicle on a given path is widely studied in literature [2] [22] [23]. The approach presented here is based on the work in [22].

Before trying to generate the actual controls, an important thing to note is that the vehicle may be completely off the requested path (which is often the case, when delays in processing or localization errors are considered). Thus it is initially necessary to develop a feasible trajectory, or *connecting contour*, between the current position and orientation and a suitable point on the requested trajectory.

Another important factor that has had pivotal importance in the choice of the control model is that the environment in which the vehicle is going to move is very much unstructured and subject to rapid variations (for example, due to reconstruction errors); this means that the calculated reference trajectory may change abruptly. Therefore, the driving strategy employed must be very conservative and slow.

It is thus possible to use the *bicycle model* presented in Section 2.2, which well describes the vehicle's cinematic response when the wheel's slip angle is low, and is very effective at slow speeds.[6]

Having introduced this assumption makes it possible to separately consider the lateral and forward dynamics of the vehicle, as if the two were not influencing each other. Another way to put this is that we decouple the steering capacity of the vehicle from its forward speed and vice-versa, leading to two separate controllers for the steering angle and the thrust control.

[6]Determining the actual value of this limit is not a trivial task, but the vehicle was never planned to resort to high speed, high slip angles driving, such as rally drivers do. In fact, the wheel slip is computed on the fly so that the controller will be alerted when the vehicle is driving close to the safety envelope.

3.5.1 Endpoint selection for the connecting contour

While theoretically the vehicle orientation may be completely unrelated to the requested path, it is safe to assume that at least a point of the requested trajectory falls in the forward half-plane of the vehicle. (If that wasn't the case, since path planning is executed often and always creates a path starting from the current vehicle position, it would mean that an external perturbation—like a slippage—unexpectedly modified the vehicle position, and the path planner has not yet decided on a different path to follow. In any case a new, forward-facing path is going to be generated soon.)

The algorithm translates the path in body coordinates, so that the vehicle is considered to be at the origin of the plane, looking towards the increasing x axis. It then looks for an endpoint that matches the following conditions:

- has index greater or equal than that of the closest one to the vehicle's current position;
- has the highest x-coordinate which is less than L, where L is a value which increases with speed;
- the x-coordinates of the points between the closest one and the selected endpoint define an increasing function.

This last property ensures that, in the case of an "$\supset$"-shaped path, the vehicle always connects with the path without cutting the path.

3.5.2 Connecting contour generation

The selection of a connecting path has a fundamental implication in the definition of the steering control strategy, since these are directly derived from the bicycle model inversion properties. For this work, polynomial functions have been selected due to their regularity (they are C^∞) and simple, understood mechanics.

To calculate the coefficients, it is necessary to place a number of conditions on the polynomial function $P(x)$:

1. have value 0 at the axis origin (starting vehicle position);
2. have tangent 0 at the axis origin (starting vehicle orientation);
3. pass by the chosen endpoint (end condition);

4. at the endpoint, have the same tangent the path has (straight connection at endpoint);
5. at the origin, have the same curvature the vehicle currently has (starting steer continuity).

This set of constraints translates in a minimum satisfying polynomial $P(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$ of grade 4, whose parameters can be easily obtained by applying the above conditions. In particular, the first two require the zeroing of a_0 and a_1, since

$$P(0) = 0 \quad \Rightarrow \quad a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4|_{x=0} = 0 \quad \Rightarrow \quad a_0 = 0 \tag{3.16}$$

$$P'(0) = 0 \quad \Rightarrow \quad a_1 + 2a_2 x + 3a_3 x^2 + 4a_4 x^3|_{x=0} = 0 \quad \Rightarrow \quad a_1 = 0 \tag{3.17}$$

The remaining three conditions become

$$P(x_r) = y_r \quad \Rightarrow \quad a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4|_{x=x_r} = y_r \tag{3.18}$$

$$P'(x_r) = m_r \quad \Rightarrow \quad a_1 + 2a_2 x + 3a_3 x^2 + 4a_4 x^3|_{x_r} = m_r \tag{3.19}$$

$$P''(0) = c \quad \Rightarrow \quad 2a_2 + 6a_3 x + 12a_4 x^2|_{x=0} = c \quad \Rightarrow \quad a_2 = c/2 \tag{3.20}$$

where x_r, y_r, m_r are the coordinates and the angular coefficient of the path at the endpoint, and c is the current vehicle path curvature.

3.5.3 Control values

The motion of a vehicle described by the bicycle model on a plane is bound by the following equations:

$$\begin{cases} \dot{x} &= u_1 \cos\psi \\ \dot{y} &= u_1 \sin\psi \\ \dot{\psi} &= \frac{u_1 \tan\delta}{l} \end{cases} \tag{3.21}$$

where the input u_1 represents the forward speed and the second input δ is the wheel steering angle.

So, once $x(t)$ and $y(t)$ are known, $u_1(t)$ is easily expressed as

$$u_1(t) = \sqrt{\dot{x}(t)^2 + \dot{y}(t)^2}$$

while $\delta(t)$ is instead obtained by the ratio of the first two equations in

(3.21):

$$\begin{array}{rcl} \frac{\dot{y}(t)}{\dot{x}(t)} & = & \tan(\psi(t)) \\ \psi(t) & = & \arctan(\frac{\dot{y}(t)}{\dot{x}(t)}) \end{array} \tag{3.22}$$

and finally expressing δ from the second one as

$$\delta(t) = \arctan(\frac{l\dot{\psi}(t)}{u_1}) \tag{3.23}$$

3.5.4 Steer and throttle calculation

The actual steer and brake/throttle controls must be derived from the connecting function obtained above. Since the control cycle is executed at high speed (20 Hz), only a single value needs to be calculated for each step: the resulting vehicle movement will be considered when choosing the controls at the next iteration, and the actual path will be an "envelope" of all the instantaneous control commands.

The steer value is directly obtainable by calculating the path curvature at the next update cycle; indeed, condition (3.20) already implies that the curvature at $t = 0$ is the current one.

To calculate the speed controls, a number of safety conditions must be taken into account:

- At a given point, the maximum speed is inversely related to the path curvature. The exact value is calculated by knowing the physical properties of the vehicle (mass, inertia, etc. . .), and the current estimation of the maximum tire force, computed in Section 3.3.
- The maximum speed is reduced linearly at a safe rate, so that the vehicle will stop at the end of the path. This has been done because if the controller for any reason does not get any updates from the path planner, the last path received is considered the only safe driving possibility.
- Finally, as an additional safety measure, the actual point that is considered for velocity calculation is not the current position p_0, but the position p_a the car would reach if doing an emergency brake manuever starting from the current point and the current steer angle. This effectively penalizes trajectories that, while being close to the requested path, have the potential to drive the vehicle away from it.

Once the actual position for which the velocity needs to be computed is known, the closest point p_p on the path is calculated. The velocity can then be obtained as a function of the current distance d of p_a from p_p:

$$v = \begin{cases} v_p & d < 1 \\ v_p \left(\frac{1}{2} + \frac{1}{2} \cos \left(\frac{d-1}{\varepsilon_p - 1} \pi \right) \right) & d > 1 \end{cases} \tag{3.24}$$

where v_p is the "speed limit" calculated above for p_p, and ε_p is the distance of the closest obstacle from that point, as returned by the path planner. This has the effect of creating a central zone, around the reference trajectory, where the speed is unrestricted. When the vehicle moves away from it, the speed is linearly reduced down to zero.

These values, appropriately scaled, can be used to generate commands for the vehicle's mechanical controls.

Chapter 4

World sensing and behaviors

One evidently basic sub-system that is central to the design of the autonomous vehicle is the sensing system: the design has to provide the vehicle with “eyes” to see the world around, so as to provide data for all intelligent activities, from obstacle avoidance, to motion planning and navigation, that have already been explained in the previous Sections.

Of course all these activities are based on the derivation of a map of the world around starting from observations (in the actual sense) of the environment.

In more detail, the current map of the surroundings is obtained via temporal integration of preprocessed data coming from several sources, namely:

- a stereo, long range, fixed camera pair;
- a shorter range, stereo camera pair mounted on a pan-tilt platform;
- a LIDAR sensor mounted above the vehicle and looking slightly down, at the road ahead.

A high-level, rule based fuzzy logic decision system monitors the overall behavior of the system and sends the appropriate requests to the path planner.

4.1 Stereo vision

For this project, two different algorithms have been implemented for extracting obstacle information from stereo cameras: using the classic

Bayesian MAP approach on the disparity map, and the one Bertozzi, Broggi and Fascioli proposed in [24].

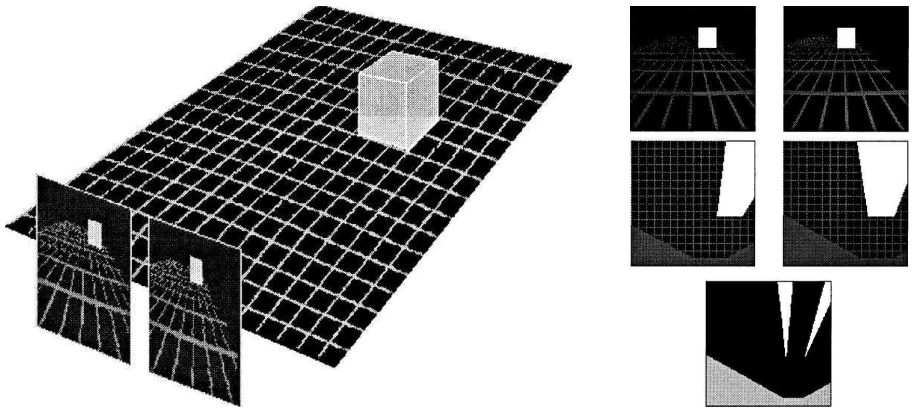
4.1.1 Bayesian MAP

The first approach tries to exploit the parallax effect, so that one can reconstruct an objects depth by observing where it appears in the two images. To be effectively used, the two images need to be distortion-rectified and warped so that the projected images appear to be coming from ideal pinhole cameras looking along parallel lines. Experiments have been made with both local methods (looking for patterns in one image and trying to match these on the other), and global methods (looking for the sequence of disparity changes that best matches the image correspondences). In a set of real-world tests, local methods performed better, since they do make less hypothesis on the environment but just consider the actual image patterns.

4.1.2 Stereo IPM

The approach used by Bertozzi et al., instead, is designed for obstacle detection on roads or similar flat terrains. In fact, the algorithm works on the hypothesis that the cameras are looking at a mostly flat surface, of known orientation, which constitutes a sort of a priori information (scene model). Loosely speaking, all elements that significantly stand out of the normality of the flat road surface are considered as obstacles.

Based on this rationale, the algorithm proceeds by computing an Inverse Perspective Mapping on each of the cameras, to remove perspective effects, and the result is a kind of warped image where it looks as if the image was taken with a camera perpendicular to the surface. Objects that have different height, however, do appear distorted differently in the two cameras, due to their unique view angle. By analyzing the difference between the two IPM images, it is possible to obtain object shape and height information. For example, in Figure 4.1 the presence of an ideal obstacle is deduced by observing the triangular traces left by the obstacle walls in the image obtained as the difference between the camera pair views. Processing these data allows determining position and shape of the obstacle.



(a) World scene and cameras

(b) Inverse Perspective Mapping steps

Figure 4.1: Inverse Perspective Mapping. An ideal obstacle is seen from different point of view from a camera pair (left image). From the difference between the images it is possible to infer the obstacle features (right image).

Of course, the detection of a real obstacle poses further problems regarding both the not regular shape of the shapes in the difference image and the presence of noise in the source images, which can bring in the need for additional filtering procedures.

The downside of this algorithm is that it depends a lot on knowing the surface orientation precisely; as a matter of fact, a wrong IPM transformation would result in all the terrain being classified as obstacle. To improve error resiliency from the reconstructed vehicle attitude, a number of iterations of the algorithm are performed, while refining at each step the vehicles pitch angle.

4.2 LIDAR

The LIDAR (Light Detection and Ranging or Laser Imaging Detection and Ranging) is a well established technique used to determine the distance of an object by means of a laser impulse of known wavelength. The sensor generates high-speed depth readouts by sweeping a laser beam on a linear path and measuring the delay and intensity of the reflected light.

There are many motivations for using active optical sensors for per-

ception. Precision as well as reliability are often the cited reasons. Moreover, a significant amount of research has been conducted in LIDAR obstacle avoidance as the technology has become more affordable. However, the application of LIDAR to terrain characterization has only recently been suggested.

Henriksen and Krotkov, in their paper [25], suggest that there are three primary terrain hazards detectable with laser, and compare the relative detection rates between LIDAR and vision:

- Positive elevation hazards, also known as "steps", indicate an abrupt rise in the level of terrain. This class of hazards includes boulders, natural perforations in the landscape and broken rock surfaces. Both laser and vision perform equally well in detecting step hazards.
- Negative elevation hazards, also called "ditches", represent abrupt downgrades in the landscape such as craters and cliffs. Laser range finding presents a drastic improvement over vision-based methods in detecting negative hazards due to difficulties in identifying distances through vision.
- The last class of terrain hazard is known as the "belly" hazards. These represent areas of terrain, such as dunes, that are continuous but not traversable due to the clearance physics of the vehicle, and are equally likely to be detected by LIDAR and vision methods.

Roberts [26] proposes a basic, fixed-mount laser range finder, placed on the front of vehicles to detect immediate obstacles and terrain hazards. The sensor is able to gather terrain information only if the vehicle is moving, but the setup is very popular due to its ease of construction and use.

In the framework of the autonomous vehicle, a series of sweeps is made at different times, exploiting the car movement to provide a mesh of points on the road ahead. This point collection is filtered by a surface extraction algorithm tuned to find "passable terrain", by checking a number of conditions on the first and second-order derivatives on the mesh segments. To further reduce noise from this data, the estimated surface parameters are integrated by a Kalman filter. By differencing the expected surface from the actual point mesh and grouping outliers, obstacles can be classified depending on the height (positive or negative),

size, or shape, and the area ahead of the vehicle is segmented in “passable”, “unpassable” and “unknown”.

4.3 Local map and Global map

Obstacle maps come from different sensors, and each sensor is different in range, resolution and covered area. In the case of the pan-tilt cameras, the covered area is not even fixed w.r.t. the vehicle. To further complicate things, sensors have different update rates, and the processing delay is also not constant. All this information must however be fused into a common map to be used by the planning algorithms.

To create a common representation, each of the sensors is required to generate a grid map with the shape of a trapezoid; however, the number of cells, the overall size and position of the grid are parameters that are associated with the sensor and current measure. For each cell, the sensor must output if that small area has been measured, and if so the probability that an obstacle is present at that location.

A local map is generated by keeping track of all the measurements by timestamp; once two measurements for the same time are present, the grids are intersected and then merged together using a weight map based on the sensor’s characteristics (e.g. the LIDAR measurements have always the same uncertainty, while stereo vision resolution inherently worsens with distance).

This local map is updated in real-time with the car sensors current knowledge of the surroundings, and thus may miss, for example obstacles that went “out of scope” from all of the sensors, but still are relevant for the planner. For this reason, a higher-level description of the obstacles is obtained by extracting features (blobs) from the maps, analyzing the boundaries of these blobs and describing them in terms of an ellipse or a simplified polygon. These objects are then kept in a global map of all the possible obstacles, where each of them is tracked and updated independently (to improve accuracy) via Kalman filtering of its parameters from one observation to the next.

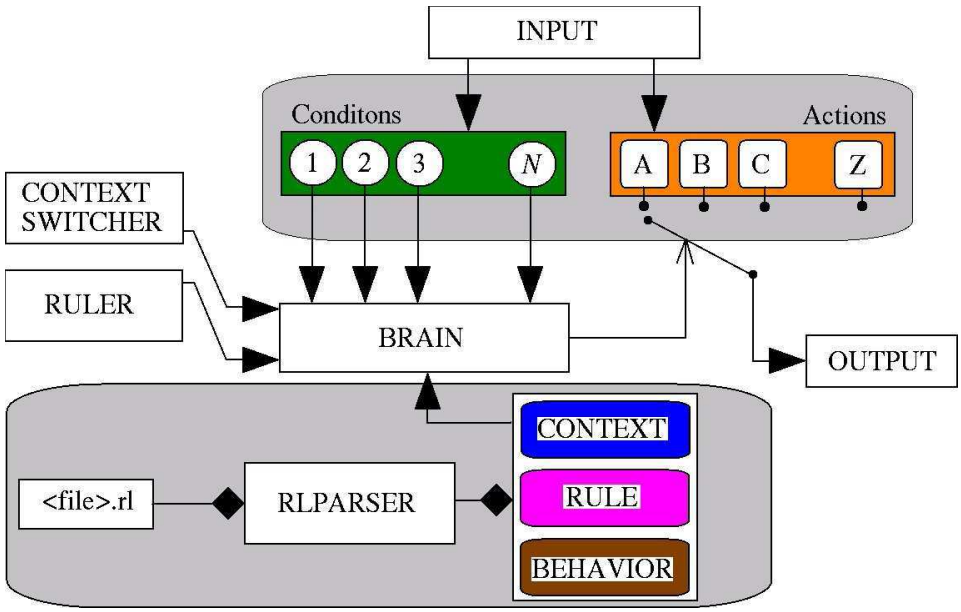


Figure 4.2: The fuzzy-logic decision system structure.

4.4 Decision system

The highest-level control scheme is based on a fuzzy-logic [27] system similar to ARTIFACT, the one developed for the Artisti Veneti Robocup team [28]. The most useful theorical abstraction here is that of a switching system [29], where the system evolves under the supervision of a controller, but it may instantaneously switch to a different control set at any time. Its functional diagram is shown in Figure 4.2.

This system is based upon four basic kinds of objects:

- *Conditions* are a floating point quantity in the range $[0, 1]$ that describes how probably true this particular condition is. A standard number of conditions are derived from vehicle-measurable quantities (e.g. `CloseToObstacle`, `OnPath`, ...), while custom ones can be easily defined by composing basic conditions with the normal fuzzy logic operators.
- *Actions* are the outputs of the fuzzy-logic system, and define the overall behaviour of the autonomous vehicle (such as `ReachNextPoint`, or `EmergencyStop`). Each defined action is associated to a (possibly complex) condition result; at each evaluation, the one that is associated with the highest condition value is selected for execution.

- *Rules* are the actual description of the link between an Action and its Condition. They specify a threshold value under which the Action is never taken.
- *Contexts* are useful to store fuzzy logic "state". Using Contexts, rules can be grouped in sets which can selectively enabled or disabled (again, depending on a condition). This is used, for example, to switch the controller from normal to emergency behaviour.

The Brain is the entity that evaluates all the conditions in the active contexts, and activates the most likely action. It can be however interrupted by the Ruler, which checks if any threshold has been met, or by the Context Switcher, which keeps track of the active contexts. If a new set of rules needs to be evaluated, after the updates the Brain is executed again.

Most fault conditions are also handled by the fuzzy logic system by employing contexts appropriately. If an unhandled software exception escapes from the cell-specific code, it is caught, serialized by the middleware, and sent to the master server. In this way the decision system is able to react and take the appropriate corrective action, depending on the error and the erratic cells duties. For more discussion on this topic, see Section 6.5.

Part II

Software Architectures

Chapter 5

Introduction

Advances in electronics design and integration has pushed toward the migration from mechanical and hydraulic servo systems to systems commanded by computational units at lower and lower costs, paving the pathway to the definition of the *drive-by-wire* paradigm and inspiring even more ambitious research challenges over the unmanned vehicle idea.

The increasing number of software-controlled functions and their interconnection, however, pose significant challenges in all the phases of a vehicle's lifecycle:

- From the early stages of the **design** phase, the addition of electronics and software calls for careful planning of the interactions, and generally the use of algorithms which are more complex than standard "analog" control techniques.

- Before and during development, recent advances in computer capabilities have made software **simulation** of these complex systems a given, since this greatly reduces final development costs, and also provides early detection of design errors.

- The demands made on testing methods during the **implementation** phase have become more stringent as well. Hardware-in-the-loop simulators and laboratory vehicle setups have become permanent fixtures in the development departments of many companies, especially in the early phases of development, when such installations serve to augment road testing.

- While most of the outside environment can be abstracted and simulated, a number of possible design errors or unexpected side-effects become apparent only during the **live testing** of the developed vehicle. To achieve the ultimate performance, final calibration and refinements of the control strategies can only be based on actual, sensor-recorded data.

Part I addressed some of the most recent algorithms that have been presented in the field of autonomous vehicles. The following chapters will instead provide some practical examples of architectures that can fulfill automotive software requirements.

5.1 Design decisions

When designing mixed electronic/software systems, depending on the task at hand, a number of different choices needs to be made. There is never a clear cut distinction between the options, however, but they serve as a general guidelines during the very first design decisions.

5.1.1 Size

One of the first issues a system developer has to make clear is the overall "size" of the problem. Some tasks may be solvable by using a simple, standalone CPU; some may benefit from cooperation with other processing equipment; some may require high computational power more akin to a server PC. This basic design decision is often a given, due to other constraints such as space, cost or reliability, but has nevertheless profound impacts on the overall system choices.

5.1.2 Operating systems

When dealing with a programmable processing unit, there are a number of basic operations that almost every programming paradigm requires. Issues such as timing/delays, resource allocation, multiprocessing are well understood and usually handled by an *operating system* (or OS). These provide a consistent API and robust, proven implementations of common interfaces (such as networking stacks), allowing developers to forget about some of the most low-level details and focus more on the

specific task at hand. This additionally allows higher code portability and generally forces better coding practices, such as privilege separation.

Traditionally, for smaller and specialized tasks OSes have been considered overkill and resource-hungry; however, ever-increasing silicon capabilities, along with specialized, low-footprint OSes, make this constraint less and less relevant.

While most OSes essentially provide the same set of basic primitives, they vary in platform support and the number of available flavors is huge. From commonplace operating systems such as Windows or Linux, to their more compact, but similar sibilings (Windows CE and uClinux) to low-footprint OSes (TinyOS) or hard real-time centric (VxWorks, QNX, ...), the choice is usually dictated by the platform (Windows, for example runs only on Intel x86 hardware), hardware compatibility or extensibility (an area where Linux shines, thanks to its Open Source licensing model), or real-time support. This last term needs to be defined precisely.

5.1.3 Real time

Generally speaking, a real-time system specification contains explicit timing conditions, usually externally measurable, that have to be met in order for the system to be useful. The most common example is the response of a control system, measured as the delay between inputs and outputs, which usually has to be time-bounded.

Real-time systems can be divided into three types, depending on the meaning given to the phrase "real-time" itself:

- Hard real-time

  The most taxing are *hard real-time* systems, where failure to meet a deadline can result in complete system failure. These systems cannot tolerate any variation of the requested timings. The classic (and most dramatic) example for this type of system is a nuclear reactor control system, where due to the inherent instability of the process, a single missed deadline could lead to catastrophic loss of control.
- Firm real-time

  Sometimes the system will gracefully accept "some" misses, up to

a certain number, percentage, or continuous time. Consider, for an example, the case of a VoIP phone application: since the network does have unknown (and potentially unlimited) latency, a short buffer can be implemented to overcome these conditions; once this is depleted, the system "fails" by cutting the sound—but this is a transient error from which the system can autonomously recover. Such systems are sometimes called *firm real-time*.

- Soft real-time

  Finally, often the phrase "real time" is mistakenly used when referring to systems that need to operate "very fast", but which have no strict timing requirements. These systems (more precisely called *soft real-time*) do not fail completely when a deadline is missed, but only suffer a reduced performance. This is the case, for example, of a text input terminal, where the delay between the actual keypress and the letter appearing on the screen should be as small as possible, but the system can tolerate arbitrary delays without any failure.

Most operating systems do have provisions for, at least, supporting soft real-time requests. Usually, however, OSes do not offer any guarantee of maintaining the user-requested deadline, and the reasons are twofold. Firstly, guaranteeing boundary fulfillment is extremely difficult, because this ultimately means analyzing each and every internal execution path in the OS, making sure that no one exceeds the specified timing constraints. Also, a lot of general-purpose operating system schedulers are tuned to achieve maximum throughput and general user-side responsiveness, which usually conflicts with the hard boundaries required by real-time constraints.

There are, however, extensions to standard operating systems that provide them with a real-time layer. These extensions usually work by introducing a micro-kernel between the original OS kernel and the underlying hardware, so that the OS becomes an "idle task", and gets CPU time only when no real-time processes are active. In Linux the *de facto* standard is Xenomai, while on Windows there are a number of competitors.

5.2 Middlewares

At an even higher abstraction level, there is often the need of having different objects cooperate in solving the task at hand. Sometimes they are just different software entities (processes, threads, etc) on a single computing device; or they might be applications running on similar computers, networked together; or even completely different electronic devices (such as an FPGA offloading some complex algorithm from the CPU).

5.2.1 History

These distributed computing ideas race back to the Cold War of the 1960s, when the U.S. government began to focus on computer science research as a means toward the latest and most secure communications technology which they would need in the event of war. Communications networks were essential for the military and needed to be able to withstand attack so that chains of command would not be broken.

One researcher, Paul Baran, developed the idea of a distributed communications network in which messages would be sent through a network of switching nodes until they reached their destination. The nodes would be computers (instead of the telephone switches used at the time), so that they would be intelligent enough to decide the best route for sending each message. Many nodes and connections would create redundancy so that messages could arrive through many different paths, and the messages would be broken up into blocks of uniform length to make transmissions simpler and more efficient. This idea of message block switching, or later packet switching, was his most important innovation and was the basis for the design of networks that would become the Internet.

The first node for the ARPAnet (the first network Baran developed with these ideas) was installed in UCLA in 1969 and the other three, at the Stanford Research Institute, UC Santa Barbara and Utah, were connected by the end of that year. Since then much work has been poured on protocol research, and later when that was estabilished, on the very base of distributed computing: the *middleware*.

The term itself is cited first in a NATO Software Engineering paper in 1968; however, the meaning then was still related to a single

execution environment. Later, it was associated mainly with relational databases for many practitioners in the business world through the early 1990s, but by the mid-1990s this was no longer the case [30, 31]. Concepts similar to today's middleware previously went under the names of "network operating systems", "distributed operating systems" and "distributed computing environments".

Cronus was the major first distributed object middleware system, and Clouds and Eden were contemporaries. RPC was first developed circa 1982, by Birrell and Nelson. Early RPC systems that achieved wide use include those by Sun in its Open Network Computing (ONC) and in Apollos Network Computing System (NCS). The Open Software Foundations Distributed Computing Environment (DCE) [32] included an RPC that was an adaptation of Apollos that was provided by Hewlett Packard (which acquired Apollo). Quality Objects (QuO) was the first middleware framework to provide general-purpose and extensible quality of service for distributed objects.

Interoperability issues with all these different technologies pushed for a common agreed standard. The OMG (Object Management Group) has been established in 1989 with just 8 members, and has since grown to be presently the largest industry consortium of any kind. The organization charter was to "provide a common architectural framework for object-oriented applications based on widely available interface specifications". Version 1.0 of the standard, named *CORBA* (from the initials of Common Object Request Broker Architecture), was introduced and adopted in December 1990, but it was still incomplete in many areas.[1] Version 2.0, released in 1994, was the first one to be feature-complete and ready for interoperability.

In 1998, TAO was the first major CORBA implementation to provide (if it was supported also by the host operating system) hard real-time performance directly in the ORB.

5.2.2 Categories and examples

One of the easiest, and yet quite appropriate, ways to define a modern middleware is "Middleware is the slash in the term client/server". Middleware frameworks are designed to mask some of the kinds of hetero-

[1] For example, the standard did not mention a way to describe the object features in a common language. This was going to be later standardized as IDL.

geneity that programmers of distributed systems must deal with. They always mask heterogeneity of networks and hardware. Most middleware frameworks also mask heterogeneity of operating systems or programming languages, or both. Open standards (without a reference implementation) such as CORBA also mask heterogeneity among vendor implementations of the same middleware standard. Finally, programming abstractions offered by middleware can provide transparency with respect to distribution in one or more of the following dimensions: location, concurrency, replication, failures, and mobility.

Middlewares can be classified in terms of the environment abstractions that they provide to the programmer.

Distributed Relational Databases

Distributed relational databases are the most widely deployed kind of middleware today. Its Structured Query Language (SQL) allows programmers to manipulate sets of entities (a database) in an English-like language yet with intuitive semantics and rigorous mathematical foundations based on set theory and predicate calculus. Distributed relational databases also offer the abstraction of a *transaction*. Distributed relational database products typically offer heterogeneity across programming languages, but most do not offer much, if any, heterogeneity across vendor implementations.

Linda is a framework offering a distributed entity abstraction called Tuple Space (TS). Lindas API provides associative access to TS, but without any relational semantics. Linda offers spatial decoupling by allowing depositing and withdrawing processes to be unaware of each others identities. It offers temporal decoupling by allowing them to have non-overlapping lifetimes.

Jini is a Java framework for intelligent devices, especially in the home. Jini is built on top of JavaSpaces, which is very closely related to Linda's TS.

Remote Procedure Call

Remote procedure call (RPC) middleware extends the procedure call interface familiar to virtually all programmers to offer the abstraction of being able to invoke a procedure whose body is across a network. RPC

systems are usually synchronous, and thus offer no potential for parallelism without using multiple threads, and they typically have limited exception handling facilities.

Message-Oriented Middleware

Message-Oriented Middleware (MOM) provides the abstraction of a *message queue* that can be accessed across a network. It is a generalization of a well-known operating system construct: the mailbox. It is very flexible in how it can be configured with the topology of programs that deposit and withdraw messages from a given queue. Many MOM products offer queues with persistence, replication, or real-time performance. MOM offers the same kind of spatial and temporal decoupling that Linda does.

Distributed Object Middleware

Distributed object middleware provides the abstraction of an object that is remote yet whose methods can be invoked just like those of an object in the same address space as the caller. Distributed objects make all the software engineering benefits of object-oriented techniques (encapsulation, inheritance, and polymorphism) available to the distributed application developer.

CORBA is a standard for distributed object computing, and is considered by most experts to be the most advanced kind of middleware commercially available and the most faithful to classical object oriented programming principles. Its standards are publicly available.

DCOM is a distributed object technology from Microsoft that evolved from its Object Linking and Embedding (OLE) and Component Object Model (COM). DCOM's distributed object abstraction is augmented by other Microsoft technologies, including Microsoft Transaction Server and Active Directory. DCOM provides heterogeneity across language but not across operating system or tool vendor. COM+ is the next-generation DCOM that greatly simplifies the programming of DCOM. SOAP is a distributed object framework from Microsoft that is based on XML and HyperText Transfer Protocols (HTTP). Its specification is public, and it provides heterogeneity across both language and vendor. Microsoft's distributed object framework .NET also has heterogeneity across language

and vendor among its stated goals.

Java has a facility called Remote Method Invocation (RMI) that is similar to the distributed object abstraction of CORBA and DCOM. RMI provides heterogeneity across operating system and Java vendor, but not across language. However, supporting only Java allows closer integration with some of its features, which can ease programming and provide greater functionality.

5.2.3 Practical middlewares

As can be seen from the previous introduction, the categories of middleware are blurred in the marketplace in a number of ways. Starting in the late 1990s, many products began to offer APIs for multiple abstractions, for example distributed objects and message queues. They often use RPC or MOM as an underlying transport while adding management and control facilities. Relational database vendors have been breaking the relational model and the strict separation of data and code by many extensions, including RPC-like stored procedures. To complicate matters further, Java is being used to program these stored procedures. Additionally, some MOM products offer transactions over multiple operations on a message queue. Finally, distributed object systems typically offer event services or channels which are similar to MOM in term of architecture, namely topology and data flow.

TAO

We initially experimented with the Open Source CORBA implementation called *TAO*. TAO provides one of the most complete implementations of the CORBA standard, including real-time support (where the underlying host OS supports it). A set of C++ wrapper objects and APIs simplify handling the different parts of the standard.

Still, however, the complexities that are hidden in this specification are almost unmanageable. Due to the huge size of the steering committee, the CORBA standard specifies multiple, overlapping ways to provide the same services. Some details in the memory management are quite arcane and require careful handling of even the most basic requests. For this reason, experiments with this implementation were abandoned.

ICE

ICE[2] (Internet Communications Engine) is a middleware developed by ZeroC and released under both Open Source and commercial licenses. The team that designed this middleware has been part of the CORBA steering committee, before abandoning it for the above reasons. A complete description of the features and benefits of this architecture is given in Section 6.1; in particular, the differences with CORBA will be analyzed in Section 6.1.6.

RTCF

Another middleware that has been studied is RTCF (Real-Time Computational Framework) from SES[3]. This is an advanced, hard real-time middleware that was specifically engineered from the ground up to meet the stringent needs of large, distributed simulations even considering hardware-in-the-loop.

This middleware is based on the Distributed Object Model, specifically, its basic abstraction is the *data point*, through which the various entities that are active for the simulation cooperate using a publish/subscribe model. The underlying transport is fully decoupled from the code, allowing the same binary code to receive simulation results from a local simulation process, a remote networked PC, an FPGA add-on card or even some other forms of communication such as replicated memory or *InfiniBand*. Thanks to advanced real-time components, the same binary code can also be ported to a real-time environment without any modification or rebuild process.

Since this middleware is geared to simulations and other distributed computations, the data types that are supported by these protocols are limited to the basic integer and real numbers, strings, and array or matrices of these types. All published (numeric) data points must clearly define their own engineering unit, so implicit conversion is performed by RTCF when the same data is required in another unit, eliminating possible errors. It even supports dynamic, user defined data point filtering, so that values are modified before being published (this is very useful for simulating failures or to test error resiliency).

[2]Available from `http://www.zeroc.com`.
[3]Website for contact information: `http://www.spenceengr.com`.

Synchronization is achieved in RTCF by the use of data FIFOs, as well as distributed mutex and semaphore abstractions.

The support for different hardware platforms is very wide, but on the PC it currently relies on Microsoft Windows to be the host operating system. Real-world applications of this technology are within the *Harris* high-end broadcast products[4], as well as several aviation and space projects[5].

[4]Details available on `http://www.broadcast.harris.com`.

[5]Details are covered by NDA agreements. A list of participating companies and agencies can be found at `http://www.spenceengr.com/resources.html`.

Chapter 6

The Ice Middleware, Applied

6.1 Introduction to Ice

Ice is an object-oriented middleware platform. Ice applications are suitable for use in heterogeneous environments: client and server can be written in different programming languages, can run on different operating systems and machine architectures, and can communicate using a variety of networking technologies. The source code for these applications is portable regardless of the deployment environment.

6.1.1 Terminology

Clients and Servers

The terms client and server are not firm designations for particular parts of an application; rather, they denote roles that are taken by parts of an application for the duration of a request:

- Clients are active entities. They issue requests for service to servers.
- Servers are passive entities. They provide services in response to client requests.

Frequently, servers are not “pure” servers, in the sense that they never issue requests and only respond to requests. Instead, servers often act as a server on behalf of some client but, in turn, act as a client to another server in order to satisfy their clients request.

Similarly, clients often are not "pure" clients, in the sense that they only request service from an object. Instead, clients are frequently client–server hybrids. For example, a client might start a long-running operation on a server; as part of starting the operation, the client can provide a callback object to the server that is used by the server to notify the client when the operation is complete. In that case, the client acts as a client when it starts the operation, and as a server when it is notified that the operation is complete.

Such role reversal is common in many systems, so, frequently, client–server systems could be more accurately described as peer-to-peer systems.

Objects

An *object* is a conceptual entity, or abstraction. An object can be characterized by the following points:

- An Ice object is an entity in the local or a remote address space that can respond to client requests.

- A single object can be instantiated in a single server or, redundantly, in multiple servers. If an object has multiple simultaneous instantiations, it is still a single object.

- Each object has one or more *interfaces*, or *facets*. An interface is a collection of named *operations* that are supported by an object. Clients issue requests by invoking operations.

- An operation may have parameters as well as a return value. Parameters and return values have a specific type. Parameters are named and have a direction: in-parameters are initialized by the client and passed to the server; out-parameters are initialized by the server and passed to the client. (The return value is simply a special out-parameter.)

- If an object has more than one interface, clients can select among the facets of an object to choose the interface they want to work with.

- Each Ice object has a unique object identity. An objects identity is an identifying value that distinguishes the object from all other objects. The Ice object model assumes that object identities are globally unique, that is, no two objects within an Ice communication domain can have the same object identity.

Proxies

For a client to be able to contact an Ice object, the client must hold a *proxy* for the Ice object.[1]

A proxy is an artifact that is local to the clients address space; it represents the (possibly remote) Ice object for the client. A proxy acts as the local ambassador for an Ice object: when the client invokes an operation on the proxy, the Ice run time:

1. Locates the Ice object
2. Activates the Ice object's server if it is not running
3. Activates the Ice object within the server
4. Transmits any in-parameters to the Ice object
5. Waits for the operation to complete
6. Returns any out-parameters and the return value to the client (or throws an exception in case of an error)

A proxy encapsulates all the necessary information for this sequence of steps to take place. In particular, a proxy contains addressing information (to contact the correct server for the remote object), an object identity (to identify which particular object in the server is the target) and an optional facet identifier, that determines which particular facet of an object the proxy refers to.

Proxies can be *direct* or *indirect*. A direct proxy embeds an object's identity, together with the address at which its server runs and protocol-specific information, so that the addressing information in the proxy is directly used to contact the server; the identity of the object is sent to the server with each request made by the client.

[1] A proxy is the equivalent of a CORBA object reference. We use "proxy" instead of "reference" to avoid confusion: "reference" already has too many other meanings in various programming languages.

Conversely, an indirect proxy may only provide an object's identity (optionally with an object adapter identifier), but no addressing information. To determine the correct server, the client-side run time passes the proxy information to a location service. In turn, the location service uses the object identity or the object adapter identifier as the key in a lookup table that contains the address of the server and returns the current server address to the client, which now knows how to contact the server and will dispatch the request as usual. The process of resolving the actual address from a proxy is called *binding*.

The entire process is similar to the mapping from Internet domain names to IP address by the Domain Name Service (DNS): when a domain name is used to look up a web page, the host name is first resolved to an IP address behind the scenes and, once the correct IP address is known, the IP address is used to connect to the server. With Ice, the mapping is from an object identity or object adapter identifier to a protocoladdress pair, but otherwise very similar. The client-side run time knows how to contact the location service via configuration (just as web browsers know which DNS to use via configuration).

The main advantage of indirect binding is that it allows to change the object's physical location without invalidating existing proxies that are held by clients. In other words, direct proxies avoid the extra lookup to locate the server, but no longer work if a server is moved to a different machine. On the other hand, indirect proxies continue to work even if the object moves to a different server.

Servants

An Ice object is a conceptual entity that has a type, identity, and addressing information. However, client requests ultimately must end up with a concrete server-side processing entity that can provide the behavior for an operation invocation. To put this differently, a client request must ultimately end up executing code inside the server, with that code written in a specific programming language and executing on a specific processor.

The server-side artifact that provides behavior for operation invocations is known as a *servant*. A servant provides substance for (or incarnates) one or more Ice objects. In practice, a servant is simply an instance of a class that is written by the server developer and that is

registered with the server-side run time as the servant for one or more Ice objects. Methods on the class correspond to the operations on the Ice object's interface and provide the behavior for the operations.

A single servant can incarnate a single Ice object at a time or several Ice objects simultaneously. If the former, the identity of the Ice object incarnated by the servant is implicit in the servant. If the latter, the servant is provided the identity of the Ice object with each request, so it can decide which object to incarnate for the duration of the request.

At-Most-Once Semantics

Ice requests have *at-most-once semantics*: the Ice run time does its best to deliver a request to the correct destination and, depending on the exact circumstances, may retry a failed request. Ice guarantees that it will either deliver the request, or, if it cannot deliver the request, inform the client with an appropriate exception; under no circumstances is a request delivered twice, that is, retries are attempted only if it is known that a previous attempt definitely failed.[2] At-most-once semantics are important because they guarantee that operations that are not idempotent[3] can be used safely.

Removing the protocol overhead for at-most-once semantics, we can build distributed systems that are more robust in the presence of network failures. However, realistic systems require non-idempotent operations, so at-most-once semantics are a necessity, even though they make the system less robust in the presence of network failures. Ice permits you to mark individual operations as idempotent. For such operations, the Ice run time uses a more aggressive error recovery mechanism than for non-idempotent operations.

Server-side and Client-side method calls

By default, the request dispatch model used by Ice is a synchronous remote procedure call: an operation invocation looks, to the client, like

[2] One exception to this rule are datagram invocations over UDP transports. For these, duplicated UDP packets can lead to a violation of at-most-once semantics.

[3] An *idempotent* operation is an operation that has the same effect if executed once or more times. For example, `x = 1` is an idempotent operation: if we execute the operation twice, the end result is the same as if we had executed it once. On the other hand, `x++` is not idempotent: if we execute the operation twice, the end result is not the same as if we had executed it once.

a local procedure call—that is, the client thread is suspended for the duration of the call and resumes when the call completes (and all its results are available).

Ice also supports asynchronous method invocation (AMI): clients can invoke operations asynchronously, that is, the client uses a proxy as usual to invoke an operation but, in addition to passing the normal parameters, also passes a callback object and the client invocation returns immediately. Once the operation completes, the client-side run time invokes a method on the callback object passed initially, passing the results of the operation to the callback object (or, in case of failure, passing exception information).

The server cannot distinguish an asynchronous invocation from a synchronous one—either way, the server simply sees that a client has invoked an operation on an object.

Asynchronous method dispatch (AMD) is the server-side equivalent of AMI. For synchronous dispatch (the default), the server-side run time up-calls into the application code in the server in response to an operation invocation. While the operation is executing (or sleeping, for example, because it is waiting for data), a thread of execution is tied up in the server; that thread is released only when the operation completes.

With asynchronous method dispatch, the server-side application code is informed of the arrival of an operation invocation. However, instead of being forced to process the request immediately, the server-side application can choose to delay processing of the request and, in doing so, releases the execution thread for the request. The server-side application code is now free to do whatever it likes. Eventually, once the results of the operation are available, the server-side application code makes an API call to inform the server-side Ice run time that a request that was dispatched previously is now complete; at that point, the results of the operation are returned to the client.

Asynchronous method dispatch is useful if, for example, a server offers operations that block clients for an extended period of time, or to complete an operation (so that the results of the operation are returned to the client) but to keep the execution thread of the operation beyond the duration of the operation invocation, to perform cleanup or write updates to persistent storage.

Synchronous and asynchronous method dispatch are transparent to the client, that is, the client cannot tell whether a server chose to process

a request synchronously or asynchronously.

Run-Time Exceptions

Any operation invocation can raise a run-time exception. Run-time exceptions are pre-defined by the Ice run time and cover common error conditions, such as connection failure, connection timeout, or resource allocation failure. Run-time exceptions are presented to the application as proper C++, Java, or C# exceptions and so integrate neatly with the native exception handling capabilities of these languages.

User Exceptions

User exceptions are used to indicate application-specific error conditions to clients. User exceptions can carry an arbitrary amount of complex data and can be arranged into inheritance hierarchies, which makes it easy for clients to handle categories of errors generically, by catching an exception that is further up the inheritance hierarchy. Like run-time exceptions, user exceptions map to native exceptions.

Properties

Much of the Ice run time is configurable via properties. Properties are name–value pairs, such as `Ice.Default.Protocol=tcp`. Properties are typically stored in text files and parsed by the Ice run time to configure various options, such as the thread pool size, the level of tracing, and various other configuration parameters.

6.1.2 Slice (Specification Language for Ice)

Each Ice object has an interface with a number of operations. Interfaces, operations, and the types of data that are exchanged between client and server are defined using the Slice language. Slice allows you to define the client-server contract in a way that is independent of a specific programming language, such as C++, Java, or C#. The Slice definitions are compiled by a compiler into an API for a specific programming language, that is, the part of the API that is specific to the interfaces and types you have defined consists of generated code.

6.1.3 Language Mappings

The rules that govern how each Slice construct is translated into a specific programming language are known as language mappings. For example, for the C++ mapping, a Slice sequence appears as an STL vector, whereas, for the Java mapping, a Slice sequence appears as a Java array. All Slice constructs are translated in the most straightforward way to the idioms of the chosen language. So, most of the time, in order to determine what the API for a specific Slice construct looks like, only the Slice definition and knowledge of the language mapping rules is needed. The rules are simple and regular enough to make it unnecessary to read the generated code to figure out how to use the generated API.

Currently, Ice provides language mappings for C++, Java, C#, Visual Basic .NET, Python, and, for the client side, PHP and Ruby; however, we have extensively used only the C++ and Java mappings.

6.1.4 The Ice Protocol

Ice provides an RPC protocol that can use either TCP/IP or UDP as an underlying transport.[4] The Ice protocol defines:

- a number of message types, such as request and reply message types,
- a protocol state machine that determines in what sequence different message types are exchanged by client and server, together with the associated connection establishment and tear-down semantics for TCP/IP,
- encoding rules that determine how each type of data is represented on the wire,
- a header for each message type that contains details such as the message type, the message size, and the protocol and encoding version in use.

The Ice protocol is suitable for building highly-efficient event forwarding mechanisms because it permits forwarding of a message without knowledge of the details of the information inside a message. This

[4] Ice also allows you to use SSL as a transport, so all communication between client and server are encrypted.

means that messaging switches need not do any unmarshaling and remarshaling of messagesthey can forward a message by simply treating it as an opaque buffer of bytes.

The Ice protocol also supports bidirectional operation: if a server wants to send a message to a callback object provided by the client, the callback can be made over the connection that was originally created by the client. This feature is especially important when the client is behind a firewall that permits outgoing connections, but not incoming connections.

6.1.5 Architectural Benefits of Ice

The Ice architecture provides a number of benefits to application developers:

- Object-oriented semantics

  Ice fully preserves the object-oriented paradigm “across the wire”. All operation invocations use late binding, so the implementation of an operation is chosen depending on the actual run-time (not static) type of an object.

- Support for synchronous and asynchronous messaging

  Ice provides both synchronous and asynchronous operation invocation and dispatch, as well as publish/subscribe messaging via IceStorm. This allows you to choose a communication model according to the needs of your application instead of having to shoe-horn the application to fit a single model.

- Support for multiple interfaces

  With facets, objects can provide multiple, unrelated interfaces while retaining a single object identity across these interfaces. This provides great flexibility, allowing for example to introduce new features in an application while remaining compatible with older, already deployed clients.

- Machine, OS and language independence

  Clients and servers need not be aware of the current machine architecture: issues such as byte ordering and padding are hidden from application code. Moreover, the Ice APIs are fully portable, so the same source code compiles and runs under both Windows and

Unix/Linux—clients and servers can even be developed in different programming languages (currently C++, Java, C#, and, for the client side, PHP).

- Implementation independence

  Clients are unaware of how servers implement their objects. This means that the implementation of a server can be changed after clients are defined, for example, to use a different persistence mechanism or even a different programming language. The Slice definition used by both client and server establishes the interface contract between them and is the only thing they need to agree on.

- Threading support

  The Ice run time is fully threaded and APIs are thread-safe. No effort (beyond synchronizing access to shared data) is required on part of the application developer to develop high-performance, multithreaded clients and servers.

- Transport independence

  Ice currently offers both TCP/IP and UDP as transport protocols. Neither client nor server code are aware of the underlying transport. (The desired transport can be chosen by a configuration parameter.)

- Location and server transparency

  The Ice run time takes care of locating objects and managing the underlying transport mechanism, such as opening and closing connections. Interactions between client and server appear connectionless. Via IceGrid, you can arrange for servers to be started on demand if they are not running at the time a client invokes an operation. Servers can be migrated to different physical addresses without breaking proxies held by clients, and clients are completely unaware how object implementations are distributed over server processes.

- Source code availability

  The source code for Ice is available. While it is not necessary to have access to the source code to use the platform, it allows you to see how things are implemented or port the code to a new operating system. Overall, Ice provides a state-of-the art development and deployment environment for distributed computing that

is more complete than any other platform we are aware of.

6.1.6 A Comparison with CORBA

Obviously, Ice uses many ideas that can be found in CORBA and earlier distributed computing platforms, such as DCE [32]. In some areas, Ice is remarkably close to CORBA whereas, in others, the differences are profound and have far reaching architectural implications.

Differences in the Object Model

The Ice object model, even though superficially the same, differs in a number of important points from the CORBA object model.

Type System An Ice object, like a CORBA object, has exactly one most derived main interface. However, an Ice object can provide other interfaces as facets. It is important to notice that all facets of an Ice object share the same object identity, that is, the client sees a single object with multiple interfaces instead of several objects, each with a different interface. Facets provide great architectural flexibility. In particular, they offer an approach to the versioning problem: it is easy to extend functionality in a server without breaking existing, already deployed clients by simply adding a new facet to an already existing object.

Proxy Semantics Ice proxies (the equivalent of CORBA object references) are not opaque. Clients can always create a proxy without support from any other system component, as long as they know the type and identity of the object. (For indirect binding, it is not necessary to be aware of the transport address of the object.)

Allowing clients to create proxies on demand has a number of advantages:

- Clients can create proxies without the need to consult an external look-up service, such as the CORBA naming service. In effect, the object identity and the object's name are considered to be one and the same. This eliminates the problems that can arise from having the contents of the naming service go out of sync with reality, and reduces the number of system components that must be functional for clients and servers to work correctly.

- Clients can easily bootstrap themselves by creating proxies to the initial objects they need. This eliminates the need for a separate bootstrap service.
- There is no need for different encodings of stringified proxies. A single, uniform representation is sufficient, and that representation is readable to humans. This avoids the complexities introduced by CORBA's three different object reference encodings (IOR, corbaloc, and corbaname).

Experience over many years with CORBA has shown that, pragmatically, opacity of object references is problematic: not only does it require more complex APIs and run-time support, it also gets in the way of building realistic systems. For that reason, mechanisms such as corbaloc and corbaname were added, as well as the (ill-defined) `is_equivalent` and hash operations for reference comparison. All of these mechanisms compromise the opacity of object references, but other parts of the CORBA platform still try to maintain the illusion of opaque references. As a result, the developer gets the worst of both worlds: references are neither fully opaque nor fully transparent—the resulting confusion and complexity are considerable.

Object Identity The Ice object model assumes that object identities are universally unique (but without imposing this requirement on the application developer). The main advantage of universally unique object identities is that they permit you to migrate servers and to combine the objects in multiple separate servers into a single server without concerns about name collisions: if each Ice object has a unique identity, it is impossible for that identity to clash with the identity of another object in a different domain.

The Ice object model also uses strong object identity: it is possible to determine whether two proxies denote the same object as a local, client-side operation. (With CORBA, you must invoke operations on the remote objects to get reliable identity comparison.) Local identity comparison is far more efficient and crucial for some application domains, such as a distributed transaction service.

Differences in Platform Support

CORBA, depending on which specification you choose to read, provides many of the services provided by Ice. For example, CORBA supports asynchronous method invocation and, with the component model, a form of multiple interfaces. However, the problem is that it is typically impossible to find these features in a single implementation, since too many CORBA specifications are either optional or not widely implemented.

Other features of Ice that do not have direct CORBA equivalents:

- Asynchronous Method Dispatch (AMD)

  The CORBA APIs do not provide any mechanism to suspend processing of an operation in the server, freeing the thread of control, and resuming processing of the operation later.

- Security

  While there are many pages of specifications relating to security, most of them remain unimplemented to date. In particular, CORBA to date offers no practical solution that allows CORBA to coexist with firewalls.

- Protocol Features

  The Ice protocol offers bidirectional support, which is a fundamental requirement for allowing callbacks through firewalls.[5] In addition, Ice allows you to use UDP as well as TCP, so event distribution on reliable (local) networks can be made extremely efficient and light-weight. CORBA provides no support for UDP as a transport. Another important feature of the Ice protocol is that all messages and data are fully encapsulated on the wire. This allows Ice to implement services such as IceStorm extremely efficiently because, to forward data, no unmarshaling and remarshaling is necessary. Encapsulation is also important for the deployment of protocol bridges, for services such as persistent data storage, because the bridge does not need to be configured with type-specific information.

- Language Mappings

  CORBA does not specify a language mapping for many “new” programming languages, such as C#, Visual Basic or PHP.

[5] At one point, CORBA specified a bidirectional protocol, but the specification was technically flawed and never actually implemented.

Differences in Complexity

CORBA is known as a platform that is large and complex. This is mostly a result of the way CORBA has been standardized: decisions have been reached by consensus and majority vote. In practice, this means that, when a new technology is being standardized, the only way to reach agreement is to accommodate the pet features of all interested parties. The result are specifications that are large, complex, and burdened with redundant or useless features. In turn, all this complexity leads to implementations that are large and inefficient. The complexity of the specifications is reflected in the complexity of the CORBA APIs: applications are thus frequently plagued with latent bugs that do not show up until after deployment.

CORBA's object model adds further to CORBA's complexity. For example, opaque object references force the specification of a naming service because clients must have some way to access object references. In turn, this requires the developer to learn yet another API, and to configure and deploy yet another service when, as with the Ice object model, no naming service is necessary in the first place.

One of the most infamous areas of complexity in CORBA is the C++ mapping. The CORBA C++ API is rather arcane; in particular, the memory management issues of this mapping are very well known. For example, objects that are received as input parameters to a CORBA server-side method call may or may not need to be `deleted` inside the method, depending on a number of conditions which the developer always has to consider. Yet, the code required to implement this C++ mapping is neither particularly small nor efficient, leading to binaries that are larger and require more memory at run time than they should.

In contrast to CORBA, Ice is first and foremost a simple platform. The designers of Ice took great care to pick a feature set that is both sufficient and minimal: you can do everything you want, and you can do it with the smallest and simplest possible API. This simplicity makes it easy to learn and understand the platform, and leads to shorter development time. At the same time, however, Ice does not compromise on features: with Ice, you can achieve everything you can achieve with CORBA and do so with less effort, less code, and less complexity. This has been probably the most compelling advantage of Ice over any other middleware platform: things are simple—so simple, in fact, that it is

possible to start developing distributed applications after only a few days exposure to Ice.

6.1.7 Ice Services

The Ice core provides a sophisticated client/server platform for distributed application development. However, realistic applications usually require more than just a remoting capability: typically, you also need a way to start servers on demand, distribute proxies to clients, distribute asynchronous events, configure your application, and so on.

Ice already provides a number of services that cover these and other features. The services are implemented as Ice servers to which your application acts as a client. None of the services use Ice-internal features that are hidden from application developers—however, having these services available as part of the platform allows to focus on application development instead of having to build a lot of repetitive infrastructure first. Moreover, building and debugging such services is not a trivial effort, so it is useful to use what is already available, instead of reinventing your own wheel.

The following is a survey of the provided ICE services which were used in the autonomous vehicle project.

IceGrid

IceGrid is an implementation of an Ice location service that resolves the symbolic information in an indirect proxy to a protocol–address pair for indirect binding. A location service is only the beginning of IceGrid's capabilities:

- IceGrid allows you to register servers for automatic start-up: instead of requiring a server to be running at the time a client issues a request, IceGrid starts servers on demand, when the first client request arrives.
- IceGrid provides tools that make it easy to configure complex applications containing several servers.
- IceGrid provides a simple query service that allows clients to obtain proxies for objects they are interested in.

IceBox

IceBox is a simple application server that can orchestrate the starting and stopping of a number of application components. Application components can be deployed as a dynamic library instead of as a process. This reduces overall system load, for example, by allowing you to run several application components in a single Java virtual machine instead of having multiple processes, each with its own virtual machine.

IceStorm

IceStorm is a publish–subscribe service that decouples clients and servers. Fundamentally, IceStorm acts as a distribution switch for events. Publishers send events to the service, which, in turn, passes the events to subscribers. In this way, a single event published by a publisher can be sent to multiple subscribers. Events are categorized by topic, and subscribers specify the topics they are interested in. Only events that match a subscriber's topic are sent to that subscriber. The service permits selection of a number of quality-of-service criteria to allow applications to choose the appropriate trade-off between reliability and performance. IceStorm is particularly useful if you have a need to distribute information to large numbers of application components.[6] IceStorm decouples the publishers of information from subscribers and takes care of the redistribution of the published events.

[6] A typical example is a stock ticker application with a large number of subscribers.

Figure 6.1: The autonomous vehicle

6.2 Vehicle architecture

The following section will briefly describe the environment that has been created for the autonomous vehicle project [33].

6.2.1 Hardware

The car used in this project, visible in Figure 6.2.1, is an automatic transmission 1990 Range Rover. This car has been augmented with several mechanical and electronic equipment to make drive-by-wire possible.

Onboard sensors The vehicle is equipped with a number of different sensors that provide knowledge about its surroundings:

GPS A *NovAtel OEM4G-2*[7] GPS device, whose antenna is visible in Figure 6.2(a), provides the system with absolute, but relatively slow (10 Hz) and inaccurate (20 m RMS), positioning.

IMU The box visible in Figure 6.2(b) contains the *X-Sens MT9-B*[8] inertial measurement unit, which returns at 100 Hz raw 3-dimensional

[7] See `http://www.novatel.com/products/oem4g2.htm` for device information.
[8] Documentation is available on `http://www.xsens.com`.

(a) GPS antenna

(b) IMU

(c) Short range cameras

(d) Long range cameras

(e) LIDAR scanner

Figure 6.2: The autonomous vehicle: onboard sensors

linear acceleration, angular (gyro) velocity, and Earth magnetic field readings.

Short-range vision Two *AVT Marlin F-131B*[9] FireWire cameras are mounted on a pan&tilt stand at the center of the rooftop. Their short baseline makes them useful to recover good range data for close objects (covering 2–10 m from the focal point). Protection from atmospheric elements is given by the transparent plastic sphere visible in Figure 6.2(c).

Long-range vision Two similar (but fixed) cameras have been placed at the front edges of the vehicle roof, as can be seen in Figure 6.2(d), giving them a much wider baseline (1.7 m) and thus making them capable of perceiving object depth up to 50 m.

LIDAR Figure 6.2(e) shows the *SICK LMS-211*[10] LIDAR device mounted on the roof, facing slightly downwards. Together with the forward movement of the vehicle, this is used to provide a mesh of points describing the road surface.

Motors and power electronics Three AC electric motors have been added, acting respectively on the steering wheel and the brake and throttle pedals.[11] Each of these three motors is connected, on the power electronics board visible in Figure 6.3(d), to a *Danaher Motion SR200*[12] AC servo drive unit, which feeds the power signals to the motor and internally handles the position control loop.

The reference position for these drive units, in turn, comes from a step-direction input. Since a personal computer would not be able to generate these signals at the required speed, an additional position controller has been added, which translates commands, received from an RS485 serial port with the MODBUS protocol, into high-speed step and direction signals. Thanks to this controller's extensibility, it has been possible to augment the firmware for this device, to allow the reference point to be moved while the controller was still generating signals (the original firmware only allowed the reference to be changed while the sys-

[9] Detailed documentation available on `http://www.alliedvisiontec.com`.

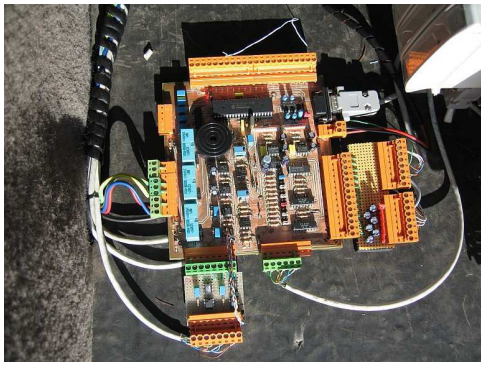
[10] Technical documentation available from `http://www.sick.com`.

[11] More advanced, but intrusive modifications were avoided because Italian law requires to have a human driver and complete and unimpeded driving capabilities, if the car is to be run along standard traffic.

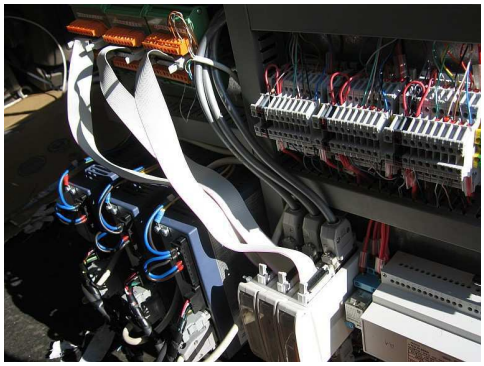
[12] See `http://www.danahermotion.com` for technical documents.

(a) Visible, from left to right: the safety device (orange), the power electronics board (mounted upright), the control PC, the two stacked vision PCs, and the UPS power supply.

(b) UPS batteries detail



(c) Safety device detail



(d) Power electronics detail

Figure 6.3: Trunk electronics.

tem was idle—this introduced unacceptable response lag in the system).

The electric power for the whole system comes from a *SOCOMEC MODULYS*[13] large UPS device, which is capable of generating 220 V AC power and providing up to 1000 W of sustained power output. All of this energy is stored in four dedicated 24 V batteries, shown in Figure 6.3(b), which together can provide up to 2 kWh energy. Harnessing the power of the vehicle engine to charge these batteries proved impractical, since the modifications to the vehicle would have been significant, so the batteries are recharged by connecting the vehicle to the mains power.

Figure 6.3(a) shows the electronics housed in the trunk of the car.

Low-level control A custom board, shown in Figure 6.3(c), has been developed [34] to provide all the low-level interfaces and special processing that were necessary to allow complete software control of the autonomous vehicle. This device features a PIC microcontroller, which appears as an additional slave on the RS485 bus, and is capable of:

- Handling several high-power output lines

  For example, once the autonomous system is engaged, the key and ignition signals are controlled by this device, providing automatic start and stop of the vehicle engine.
- Performing online, continuous system safety check

  The embedded microcontroller uses some digital IO lines to check the status of all the onboard PCs, so that if something happens and breaks the system, an emergency shut-down procedure will be executed (more on this in Section 6.5).
- Reading high-speed signals from ABS encoders

  To provide odometry and dead-reckoning information, the vehicle's stock ABS encoders have been used. A dedicated electronic circuit was designed to convert the analog signal coming from the sensors into a digital signal, which is monitored by the device. The resulting revolution count, one for each wheel, is then periodically polled by the high-level software via the serial interface.

High-level control The vehicle software environment has been organized around three 64 bit server-class PCs which have been mounted in

[13] Documentation available on `http://www.socomecgroup.com`.

Figure 6.4: The passenger-side controls.

a custom rack on the back of the car. Due to the heavy computational requirements of machine vision, two have been exclusively reserved to image processing (each one connecting to a pair of FireWire cameras), while the remaining one handles all the remaining coordination, control, and planning tasks. All of them are networked together with a Gigabit Ethernet switch.

The control PC interfaces with the LIDAR using a dedicated onboard high-speed (1000 kbps) serial card; this card also provides a low-speed RS485 line going to the power electronics and safety device. It is also connected (see Figure 6.4) to a USB driving wheel and pedals, arranged on the passenger side of the vehicle (which has been used to subjectively test drive-by-wire performance), and to a 17" LCD display mounted on the dashboard that can be used to monitor the system's behavior.

The computers are running a customized version of Gentoo Linux, a standard Linux 2.6.18 kernel (with no special real-time additions), and the Ice middleware: each has its own IceBox service daemon (to spawn and monitor server processes), while the control PC also hosts the IceGrid daemon (which keeps track of what services are available where, and replies to location queries) and the IceStorm daemon (so that events are propagated to all the entities who asked to be notified).

6.2.2 Ice extensions

The middleware structure used by ICE is not adequate for directly implementing the various control mechanisms and building blocks for a whole vehicle infrastructure. A further layer of elements has been introduced to simplify interfacing with the middleware and give a better structure to the whole system.

Cells

Borrowing a term from the natural sciences, we defined the system as a collection of individual *cells*. A cell is an abstraction of the system's "building block", meaning that, in this context, a cell has these characteristics:

- Has autonomous lifespan

  Cells are viewed by the system as a single entity that can be started or stopped individually. From the ICE point of view, they are considered as servers.
- Requires inputs or provides outputs

  For their correct operation, most cells need to work in cooperation with adjacent ones. For this reason, the *boundary* of each cell defines a number of connections between adjacent cells, detailing the required inputs and the provided outputs for the current cell. These are what Ice calls services.
- Adds functionality to the system

  Each cell has some specific duties they oversee, but may not directly need the rest of the system to function correctly (apart from the inputs and outputs), that is, cells internally have the implementation for the duties they are appointed to. The whole control system is a result of the interconnections between active cells.

Sensor interfacing and decoding, trajectory planner, output actuators are all examples of cells.

Tasks

Cells abstract the core part of the algorithmic implementation. However, a number of practical implementation issues must be still covered before the cell can perform work and fully integrate with the other ones. A *task* is a special cell implementation that adds a number of very useful software tools to the cell developer:

- Multithreading

  While cells define only the implementation of their interface methods, tasks go one step further providing an internal execution thread. This can be used to offload some computation from the service call points, or provide proactive behavior, perhaps in response to external stimuli.
- Logging

  An additional layer has been implemented so that all Ice server calls are logged. Each task stores its own timestamped list of calls, along with its serialized representation (thus containing all input and return values), which is periodically flushed to disk for later retrieval or analysis.
- Communication

  Tasks automatically register their services to the network coordinator and the Pilot.
- Alarms and exception handling

  Tasks are monitored in their execution. When an exception escapes from the implementation of one of the task's methods (either because it was deliberately raised, or because it was not expected or handled) the task generic code traps this unexpected condition and sends an alert to the Pilot.

Decision system

A special task has been created to act as the decision system of the whole vehicle. As will be later explained, the *Pilot* is responsible for generating a target reference for the vehicle to reach. However, the Pilot task is unique among its peers, since it is also ultimately responsible for handling emergency conditions, so all unexpected events are relayed to this cell for processing.

Slice types

A set of predefined types have been implemented in Slice language to provide common grounds for data exchanging between servers, for items such as byte, integer, double and string vectors, and for time values. In the same language, the interface for a few key servers has been declared.

Enhancements to Ice services

A number of customizations have been made to the stock Ice services and core code. For example, the serialization code has been augmented with an "online logging" capability: all of the Ice remote method invocations are intercepted in the client library, timestamped and sent to the relevant IceBox daemon. This data is then possibly logged to disk for later retrieval, or immediately sent via the network to the GUI for real-time data flow inspection. This is also heavily customizable, as depending on the current configuration, filters can be added to limit bandwidth, so that the logging/reporting is limited to a specific remote servant, method call, or operational result. Thanks to Ice's implementation, it is also easy to discard the I/O parameters and store a more compact representation with only the method called and the timestamp. This function is proving itself invaluable in debugging multi-cell problems.

The IceBox daemon has also been extended. Ice's stock service is used to start and stop services on demand; for maximum vehicle safety, we added a runtime periodic check for each server which is started by IceBox: each task in the system periodically has to invoke an `alive()` operation on the IceBox service, otherwise, the error is reported to the Pilot cell.

6.3 Cell implementation

The complete vehicle control system is shown in Figure 6.5. Each of the boxes in the figure is a cell, and most of them are tasks too (i.e. have their own processing). A detailed description of the implementation of all these cells will be provided in the following sections.

It is interesting to note that there are at least three different control loops (depicted using different shades of gray in the figure). Since the vehicle speeds involved in this project are very low, the system can be considered quasi-static and thus require only "soft real-time" semantics for these feedbacks. However, priorities have been modified in the Linux implementation so that three groups of tasks are obtained:

- High priority

  The innermost, highest priority loop involves the IMU (2) and odometry (3) sources, which are used by the position integrator (4) to provide fast pose updates to the vehicle controller (7), which in turn adjusts the vehicle controls. This whole loop takes less than 40 ms (25 Hz) to update, and it was found to be more than adequate for the vehicle frequency response.

- Normal priority

  A more complex control loop considers the availability of GPS position updates (1) or horizon estimation values (9) for the position integrator. The latter estimates the vehicle bank angles from the images acquired by the long-range stereo camera pair. These informations are updated several times per second.
  At around the same rate, the tire model (5) provides maximum forward and lateral forces to the path planner (6), which may replan the route if the security conditions are not satisfied anymore.

- Low priority

  Finally, all the other cells are scheduled. Most of the remaining ones deal with the problem of updating the map of the surroundings (11), given the current readings from the onboard sensors. Both long- and short-range cameras feed their video streams to two obstacle detectors (8), while the laser ranging sensor sends linear depth scans to the road profile integrator (10).

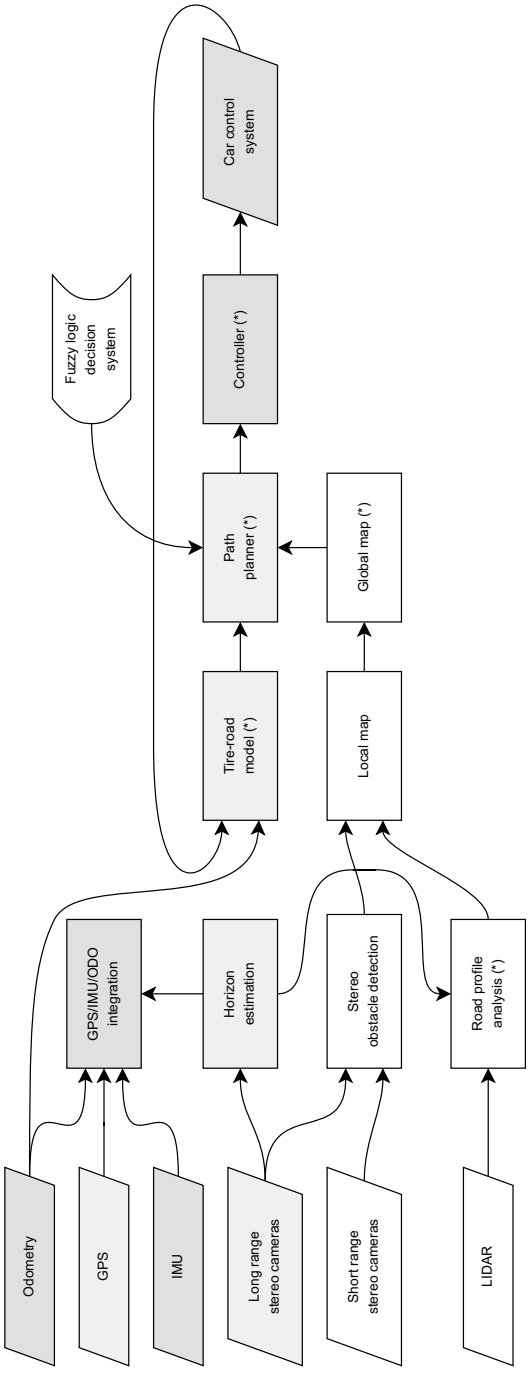


Figure 6.5: Control system structure. Darker shades of gray represent different control loops, while slanted boxes are physical system I/Os, such as sensors or actuators. To simplify the diagram, the cells that use the current computed position, velocity or attitude information are marked with (*).

6.3.1 Sensor acquisitions

1. **GPS**
   The GPS device, connected to one of the control PC's serial ports, periodically outputs NMEA standard messages. The GPS cell parses this data and forwards it by calling the `updateGPS()` function in the position integrator (4).
2. **IMU**
   Another serial port receives from the IMU sensor, at 100 Hz, the raw readings of all its internal sensors. This data is transformed according to a calibration matrix and forwarded to the position integrator (4) by calling the `updateIMU()` function.
3. **Odometry**
   By querying the safety device via the RS485 link, this cell periodically retrieves the angle each wheel moved from the last update; since the source uses the ABS wheel-locking sensors, the measure has quite good accuracy (there are 100 steps per revolution). This information is sent to the position integrator (4) and tire-road model (5) by calling the `updateOdometry()` function.

6.3.2 Control systems

4. **Position and attitude**
   The position integrator cell is responsible for reconstructing the absolute vehicle position and attitude. Currently it implements a custom solution, that merges both algorithms presented in Section 3.1 while introducing useful changes.

   The current angular position is estimated by the Gerdes filter; these values are then used to compute the Qi-Moore rotation matrix (which in the original model was derived directly from the integration of the angular accelerations). The resulting model thus improves accuracy and noise rejection. Furthermore, roll and pitch angles are also corrected by the horizon computed by the video cameras, while yaw angle drift is minimized by using the Earth magnetic field measured through the 3D magnetometer, which is much more reliable than the GPS measurements when the vehicle is steady or moving slowly. Current position, when wheel slippage is low, is also tracked by using odometry data obtained from the vehicle's ABS sensors.

5. **Maximum tire force**

   The implemented tire-road model is based on a simplified version of the Pacejika Magic Formula, approximated by three linear segments. Wheel slip is calculated comparing the current velocity vector with the actual measurement of the odometer sensor for each wheel, taking into account the geometry of the vehicle.

   Vehicle attitude and accelerations are used to calculate the current lateral, longitudinal, and normal components of the road/tire interaction force. Using these computed values, a Kalman filter has been implemented to estimate the most relevant Paceijka coefficients, which are then used to compute the current maximum tire forces. Finally, this information is sent to the vehicle controller (7) and the Pilot (12).

6. **Path planner**

   All the algorithms mentioned in Section 3.4 ultimately resulted in high computational requests; for this reason, a simpler but still effective strategy has been implemented instead.

   The current obstacle map surrounding the vehicle, and up to a certain distance, is retrieved from the global map. The obstacle boundaries are then enlarged with an ellipse whose size, aspect ratio, and major axis direction is obtained as a function of the current vehicle speed and direction. A series of splines starting from the car and ending on different points in a circle in the rough direction of the goal is then generated, and the algorithm chooses the one that, while avoiding obstacles, either is the longest or gets closest to the goal, depending on a high-level decision from the Pilot (12). From that curve, the maximum driving speed is then computed, based on the distance from the obstacles and the radius of the instantaneous osculating circle.

7. **Vehicle controller**

   The implemented algorithms (presented in Section 3.5) calculate instantaneous steer and speed values appropriate to keep the vehicle on the requested path. However, while the steer signal can be directly applied, the actual vehicle controls for the speed are indirect, being the throttle and brake pedals. This requires another controller layer.

   The speed regulator consists of two PI controllers, one for the

brake and one for the throttle. This architecture has been chosen because its output dynamics are the closest to the actual commands given by a human driver. Also, thanks to the inherent lowpass filter response of the PI controllers, this helps to reduce jerky actuator positioning and thus provides smooth, bumpless vehicle movements, safeguarding the delicate onboard electronics.

Since the car has an automatic transmission, if both the throttle and the brake are not applied, the car moves ahead at a slow pace of about 2 m/s. When the requested speed is over this threshold, the PI controller on the throttle is enough to reach the required equilibrium point. However, in the other case, the speed has to be controlled by actions on the brake pedal.

The PI controllers have been implemented with the usual integral saturation prevention methods, and parameters have been tuned to provide the quickest stable response compatible with the actuators.

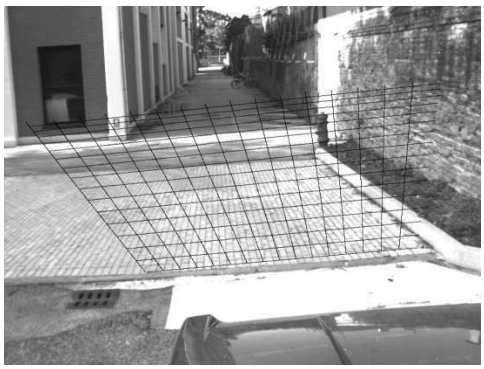
6.3.3 World sensing

8. **Stereo vision**

   The autonomous vehicle is equipped with two stereo camera pairs. The two long-range cameras are mounted at a fixed, wide baseline (160 cm), at the front edges of the roof (see Figure 6.2(d)), while the two short-range cameras have a 40 cm baseline and are mounted on a pan&tilt stand placed at the roof center.

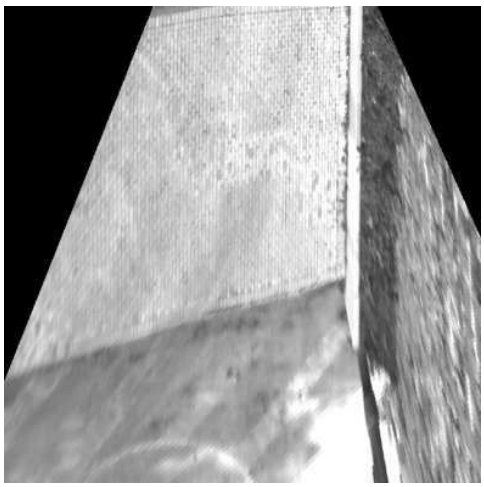
   Each of these pairs is connected to a different, dedicated PC, on which a stereo vision algorithm has been implemented. Thanks to the FireWire bus, both cameras are inherently synchronized so the two images can be compared without timing errors. The stereo cell implements the algorithms shown in Section 4.1. Figure 8 shows an actual example of the IPM algorithm running on the long-range cameras.

   This standard algorithm has been enhanced to make it more robust to rough terrain. This has been achieved by generating multiple “image-local” planes, at various grades, and testing for each combination if and where the resulting IPM matched. As an optimization, only planes that could result in passable terrain are tested (since the rest is classified as obstacle anyways).

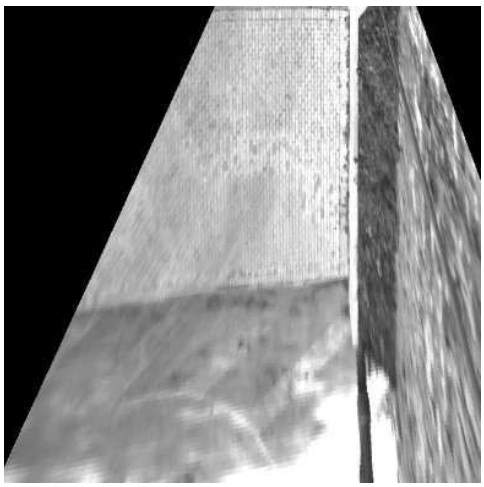


(a) Left image with grid

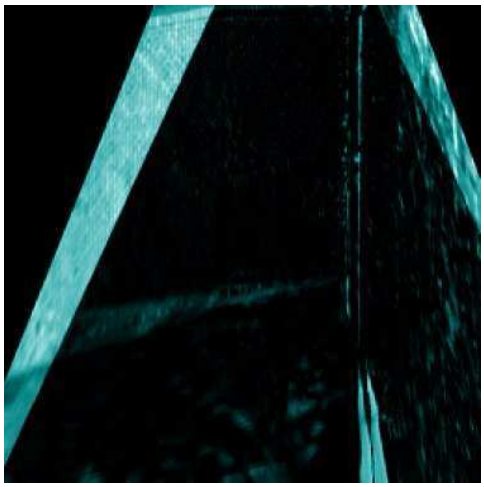
(b) Right image with grid



(c) Left IPM image



(d) Right IPM image



(e) Resulting disparity map. The building edge appears clearly at the bottom of the image.

Figure 6.6: A sample image obtained using the Stereo Inverse Perspective Mapping algorithm

Figure 6.7: Example of the horizon estimation procedure.

The short-range cameras have the added flexibility that they can be directed toward an "interesting" area around the vehicle. This has been controlled by providing the decision system (12) with actions to, for example, slightly turn the cameras in the current steering direction: this gives better localization and obstacle mapping, especially while performing tight turns.

The resulting obstacle map is then sent to the mapping cell (11) for integration.

9. **Horizon detection**

   The horizon detection algorithm, described in Section 3.2, has been implemented in a cell on the PC connected to the long-range stereo camera pair. An example of the output of the horizon detection algorithm is shown in Figure 6.7.

10. **LIDAR**

    This cell is responsible for acquiring data from the LIDAR sensor, and internally creating local obstacle maps. The implemented algorithm works by adding the 3D points obtained by each scan in a cloud map. This is processed to find out an actual terrain surface, and obstacles are obtained by grouping outliers. These maps are then sent to the mapping cell (11).

6.3.4 Reasoning

11. **Local and Global Map**

    This cell receives local obstacle maps from all the active sensors on the vehicle, and as explained in Section 4.3 combines them by calculating a geo-referenced composite map.

    To avoid filling the system memory with unnecessary information, a multiresolution map storage approach is used. The world map is divided in squares roughly 50 m on the side, and only the nine squares around the car are kept with full obstacle detail. The squares that lie farther from the car are reduced to a simpler description (a graph showing connectivity between the square edges), while the obstacle information is stored to a database on disk. A fixed size cache further limits memory usage moving the most unused data to disk when full.

12. **Decision system**

    The algoritms introduced in Section 4.4 have been implemented in a cell in the vehicle's main PC. Conditions and rules are parsed and instantiated from script files at runtime; it is possible to change the vehicle behavior quite easily, since the scripts are written like the following example:

```
Context CarEngineOn { CarActive, 0.4 }
{
  Rule betterSlowDown {
    !RoadClear | CloseToObstacle -> SlowDown, 0.2 }
  Rule goAhead {
    RoadClear -> NextPoint, 0.2 }
}
Context CarStopped { !CarMoving, 0.4 }
{
  Rule pathCompleted {
    NoMorePoints -> ShtudownEngine, 0.2 }
}
Rule PowerOnCar {
  !CarActive && !pathCompleted -> TurnEngineOn, 0.1 }
```

    In Section 6.5, the Pilot's role in the overall system safety will be discussed. One of the additional tasks is to react quickly at the occurrence of an error or exception in the system. This is handled by serializing the exception at the process level and sending this information via normal Ice calls to the Pilot. This, in turn, updates some internal conditions that will cause (via the Ruler) the activation of rules in the fuzzy logic system.

6.4 GUI

The vehicle can be monitored from a graphic interface, shown in Figure 6.8. This application has been written in Java for maximum portability (using the Java Ice wrapper libraries). The interface is based upon the NetBeans structure, which already provides windows and layout code, so that a user can drag and resize each screen object almost freely.

All of the cells have their own widget showing their parameters in the most easy way, along with an “engineering” view that shows either the current numeric values, or a trend graph.

This interface allows also to query the Ice runtime and display the current system status. As described in Section 6.2.2, all Ice messages are recorded by the runtime system, and when the GUI is connected, the current message flow can be inspected to provide debugging insights as well as more advanced status information. This data can also be replayed later by loading a saved file, for filtering or step-by-step debugging.

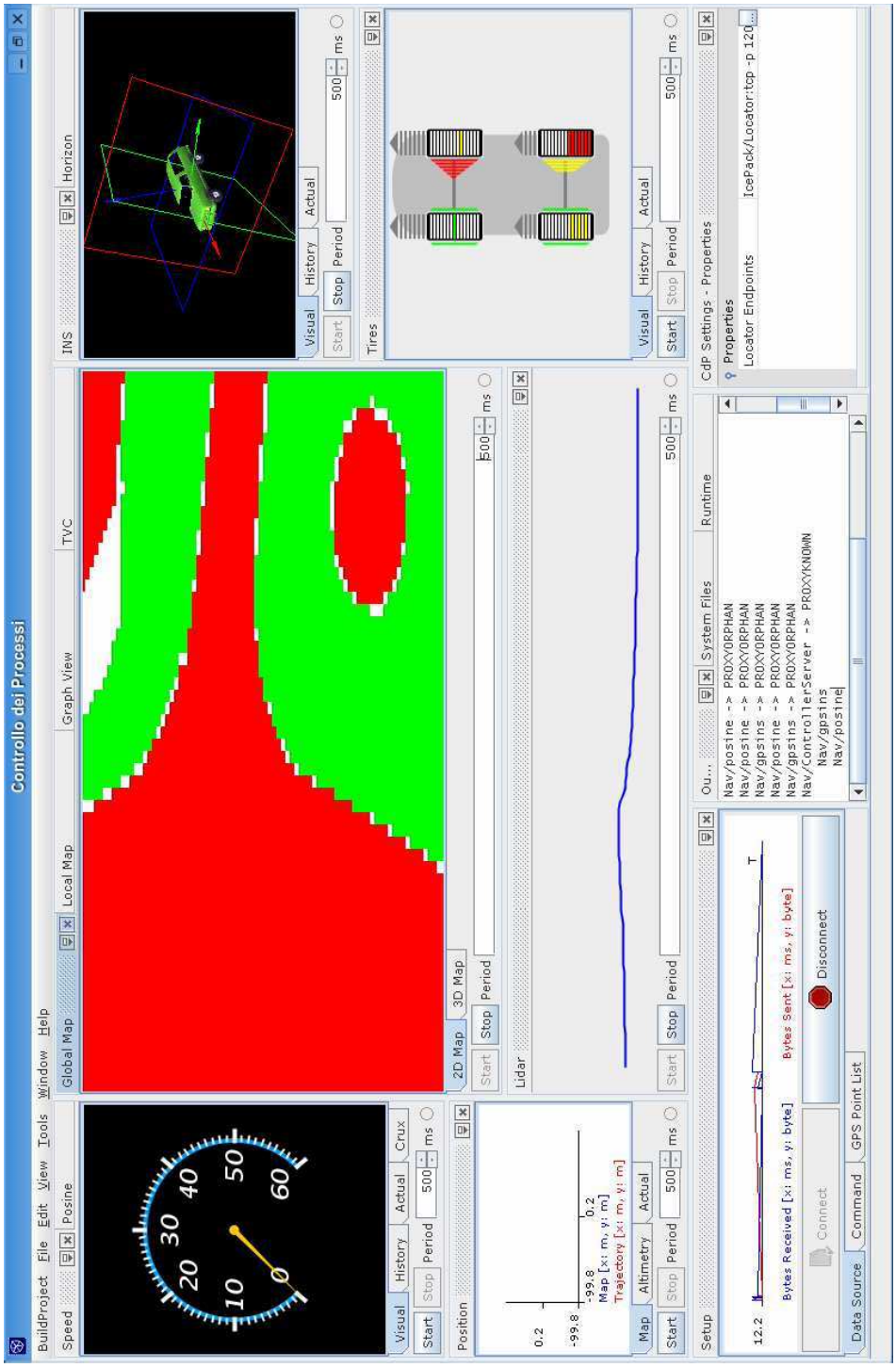


Figure 6.8: The vehicle control GUI, showing the status of multiple architectural elements (speed, local map, current attitude, LIDAR scan).

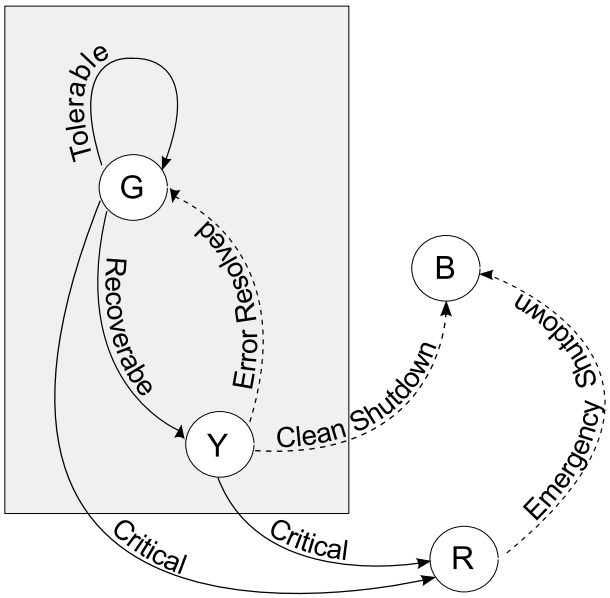


Figure 6.9: Vehicle fault classifications

6.5 Safety system

The autonomous vehicle is a complex device, and failures are almost unavoidable. For this reason, a complete system requires the development of strategies that allow to quickly identify and react to non-standard conditions. A complex, multiple-level safety system has been developed for this autonomous vehicle. Error conditions are classified on the basis of their severity, and the Pilot cell (12) cooperates with the low-level control board (previously shown in Figure 6.3(c)) in solving the issues.

6.5.1 Fault categorization

In a way resembling the classification presented in [35], it is possible to describe the system status by dividing it in four categories, according to the malfunction the system is observing:

Green is the normal system state; every subsystem is operational and no failures have been observed. The complete system resources can be used to reach the current objective.

Yellow is reached when one of the non-critical subsystems has indicated a recoverable failure; the system goes temporarily to a paused

state and tries to recover the lost functionality: if the fault is removed successfully, the system returns to Green state. However, if it can't be fixed, the system gracefully shuts down and moves to Black state.

Red is reached when a critical subsystem failed, and the overall system safety cannot be guaranteed: an emergency action is taken, and the system is immediately shut down and put in Black state.

Black means the system is not active anymore and has shut down, and it can be recovered only by manual intervention.

Using the categories outlined above, faults can be divided in three groups:

Tolerable faults do not change system status. These faults typically are in some way "expected" in advance, and thus can be handled gracefully in the user code.

Recoverable faults hinder the functionality of the system, but have still been planned for and have a recovery action associated with them. The system, however, is required to enter the Yellow state to try to recover the lost functionality.

Critical faults have a direct impact on the core control system and thus could bypass the normal error handling mechanisms, or render them ineffective. These kinds of errors are of course non-recoverable and bring the system to the Red state.

Figure 6.9 shows the transition diagram resulting from this description. It is useful to note that the transitions involving the Green and Yellow states can be managed automatically by the system. Instead, the ones taking to the Red system state (i.e., critical errors) do require a dedicated external device to assist in emergency response. Additionally, this device must be *fail-safe*: should there be other errors or malfunctions in this device, whatever the cause, the system response shall be safe. To satisfy this stringent requirement the device itself must be very simple; for example, the sheer number of failure points in a computer system makes it too complex to meet the fail-safe requirement.

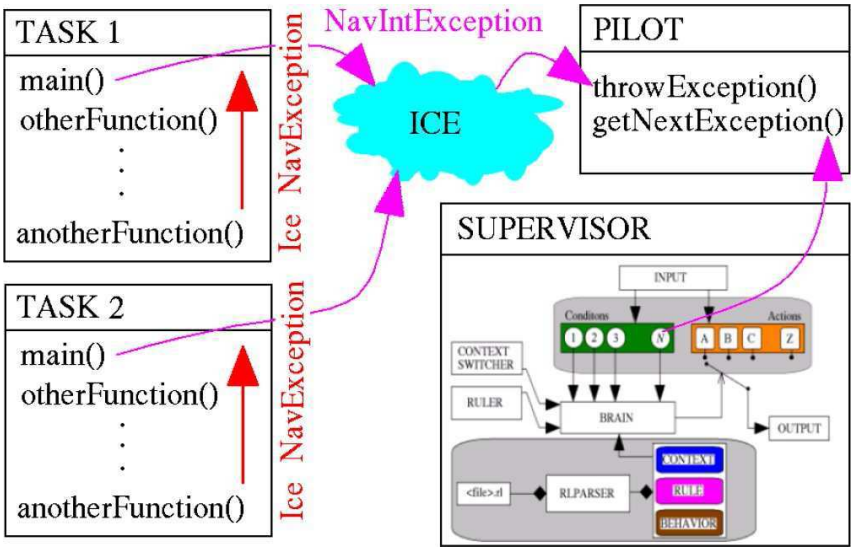


Figure 6.10: General exception handling. Uncaught exceptions are serialized and sent via Ice to the Pilot for review.

6.5.2 Tolerable errors

By far, the most probable errors the system must be able to deal with are the tolerable ones, that is, software faults in the cells. These faults are often caused by an unexpected condition or a programming error; some of these can even be anticipated and will be handled inside the process (for example, by retrying a command over a noisy serial line). Apart from—possibly—a slight performance degradation, there is no other effect on the system status.

6.5.3 Recoverable errors

Sometimes it may happen that an external sensor starts behaving erratically, or its communication channel goes out of sync. A total reset could then be required to restore full functionality; if the Pilot determines that this is required, it sends a special command to the power electronics board that effectively removes and then restores power to the device.

On the software side, some unexpected exception might not be caught by the cell-specific code. To overcome this problem, an extra security level has been embedded in the generic Ice cell code (see Figure 6.10). If an exception escapes from the implementation code for a cell (or a task),

it is propagated all the way up to the main program stack, where it is caught, serialized, and sent via normal Ice means to the error handler in the master server. In this way the Pilot is able to react and take the appropriate corrective action, depending on the erratic cell's duties.

If the error is more serious and cannot be handled by the programming code, the operating system might decide to abruptly terminate the process hosting the cell. This is, however, immediately recognized by the IceBox host monitor (which is its parent process), who will try to restart the offending executable. Any other cells that were using the restarted cell's services will transiently experience a "Service unavailable"-type error, so that they are also informed of the malfunction.

It may even happen that the communication between network hosts is lost. The server, then, cannot know if the hosts on the network are working or not. To overcome this problem, an external form of handshaking is required. For this reason, each host monitor routinely sends a heartbeat signal (via the PC's parallel port) to the low-level control board. This data is processed by the microcontroller and is periodically checked by the Pilot, so that if some host hangs for any reason, in a very short time the system notices the failure. The server may then ask the board to power cycle the failed host.

Another safety constraint is checked by the Pilot cell: it keeps double-checking the controls that are given to the car, evaluating the dynamic physical constraints to verify that these controls keep the vehicle inside its safety margins (e.g., to avoid vehicle roll over if too much steering at high speed is requested, or to reduce speed on uneven terrain). If the system is approaching a boundary, the Pilot tells the car's controller to reduce the vehicle speed and plan a more careful route.

All of these faults share the same trait: they temporarily reduce the functionality of the system. To avoid security risks, the Pilot goes into "standby" mode, stopping the car if it is moving. It then starts a recovery procedure: if the recovery goes well, the car resumes operations; otherwise, after a set number of retries, the system gives up, shuts down and waits for manual intervention.

6.5.4 Critical errors

Finally, there are failures that can't be managed by the system itself because the failure disables vital elements of the car's onboard control system.

One of these cases is that of the master server itself not responding. In this condition the the low-level control device must act autonomously and bring the car to a halt from potentially any condition, without higher-level intervention. This required a slight customization to the firmware used by the motor drives, so that if a special digital input line is raised, then the drive suspends whatever action it was doing, and starts an user-defined movement. In our case, the throttle is quickly released and the brake is slowly applied, while the steering is kept constant. While these actions might not seem the most appropriate at first, it is important to remark that these are to be used in real emergency situation, when the car is potentially in motion and a major system problem happens.

The worst case scenario is that of the drives or motor systems failing themselves. In this case no onboard electronics can act on the car control devices, and the only solution is to cut power to the engine. The microcontroller thus, upon receiving a failure status back from the drives, immediately shuts the engine off, bringing the car to a halt.

We have mentioned previously that the low-level electronics itself must be fail-safe. This has been assured in the firmware side by using a watchdog, so that if a software error is triggered, and the microcontroller stops responding to the master server, the board is immediately reset. Also, when the board is reset—or loses power—the engine relay's excitation is removed and the car stops immediately. This way, even in case of very complex or deeply located failures, a minimal safety response is ensured.

6.6 Conclusions

A software architecture for controlling autonomous vehicles has been proposed. By using a middleware solution, developers and system integrators can focus on the actual tasks instead of "reinventing the wheel" each time. This also allows to write cross-platform code more easily, and the fact that the overall vehicle status system, written in Java, has no troubles working in Linux or Windows and actively collaborating with

C++ code on remote machines is testament to Ice's power and flexibility.

Building from the presented structure, more advanced algorithms can be tested on the field, eventually helping in reaching the goal of autonomous and safe vehicle navigation.

Chapter 7

A Real-Time Data Acquisition System for Racing Vehicles

7.1 Introduction

In an automotive environment, like in many complex systems, current generation technology allows an unprecedented availability of data of very different kinds. Engine performance, vehicle safety, current system status are all routinely measured through a number of onboard sensors. Under the hood, the engine control unit (ECU) and other subsystem controllers make use of these values to constantly monitor performance and calculate corrections using proven algorithms.

In a racing context this behaviour is taken to the extreme, since the emphasis is on the highest possible performance. Onboard sensors monitor practically every possible engine parameter, the current attitude and general vehicle status; even chassis deformations are monitored to detect possible ringing effects.

The impressive quantity of sensors results in a sheer number of different interfaces required: some of the sensors simply change their physical properties (such as strain gauges, used for mechanical stress measurement, whose resistance depends on the material deformation), some only use digital signals to flag non-standard conditions (like security thermostats), others are so advanced they send digital messages over a CAN bus[1] (like the ECU), or a serial port (like the GPS).

[1]A Controller Area Network bus) is a multicast serial type bus standard used to connect several electronic control units. Developed within the automotive industry, it is

All this data is collected in real-time, and at very high frequencies, for the above purposes; however, historically most of it has been discarded immediately because of the storage size that would be required for a meaningful data log. Data logging equipment used to be very specialized and used only when required. However, the recent advances in both storage technology (with the advent of cheap, huge Flash memory devices) and computing power permit ever-increasing complexity in the data storage and analysis algorithms.

The aim for this project was to develop a custom data acquisition board capable of recording time correlated data from several sources, to aid the development and testing of different engine profiles. However, this has to be considered a base platform on which development of more complex onboard algorithms can take place, and that would ultimately provide real-time feedback to the ECU, to achieve ever-increasing performance levels.

7.1.1 Features

It is necessary to develop a system capable of synchronously acquiring data in real-time from analog voltage, digital, CAN and GPS sources, time-stamp and store it on the onboard memory for later retrieval. Furthermore, it should be small in size, use little power, and be easy to use and integrate with a standard Ethernet computer network, which is the *de facto* standard for data exchange in the racing pit.

Moreover, because of the noisy vehicle environment, it would be favorable to provide some sort of pre-processing of the incoming data, per channel, with a running average or another customized FIR filter. To this aim, analysis algorithms processing the data flow in real-time should be designed and integrated on board.

Another peculiar constraint is that, in addition to specifying the logging rate for periodic channels, non-periodic inputs (such as CAN or GPS messages) should be stored only when they actually occur. This has subtle practical implications, since most data storage file formats are heavily optimized considering only periodic data sources, which in some conditions do not provide enough flexibility.[2]

widely used also in the embedded world because of its noise immunity characteristics. For more info, see `http://en.wikipedia.org/wiki/Controller_Area_Network`.

[2]Consider, for example the case of a lap-split signal (activated each time the vehicle

Some other features that have been considered for the data acquisition system:

- data logging should start and end at configurable times, possibly depending on the current values of the configured inputs; some "pre-trigger" data should also be stored, so that it would be possible to see data before the actual start event happens;
- the storage unit is the *session*, which is further subdivided in *laps* by the lap split event. If no input is associated with this event, it defaults to a fixed time duration;
- the data retrieved from the device should be compatible with the most common engineering software packages;
- additionally, outputs may be configured to send digital signals or CAN packets with current channel data when certain events occur;
- all the settings for the device should be stored in human-editable text files, and it should be easy to upload these configurations to the actual device.

The designed system, whose details are described in the remainder of the Chapter, meets these specifications, providing:

- 2 CAN 2.0A/B ports with hardware selectable bus termination
- 1 Ethernet 10/100 Mbps network port
- 1 RS-232 serial port (with no hardware flow control); RS-232, TTL (0–5 V) or LVTTL (0–3.3 V) voltage levels
- 1 RS-485/RS-422 serial port, half/full duplex, up to 1 Mbps, programmable transmit/receive phase
- 14 differential analog inputs, with two reference voltages (one common to inputs 0–7 and one to inputs 8–13). Characteristics:
  - Input voltage: 0–5 V over the appropriate reference
  - ADC resolution: 16 bit ($\pm 76\ \mu$V)
  - Sampling frequency: 1 kHz on all channels
  - Sampling synchronization error: less than 50 μs

crosses the finish line). The user of these systems is then forced to choose either a very fast polling rate (so that the exact time of the event is recorded) or a very slow one (to conserve memory).

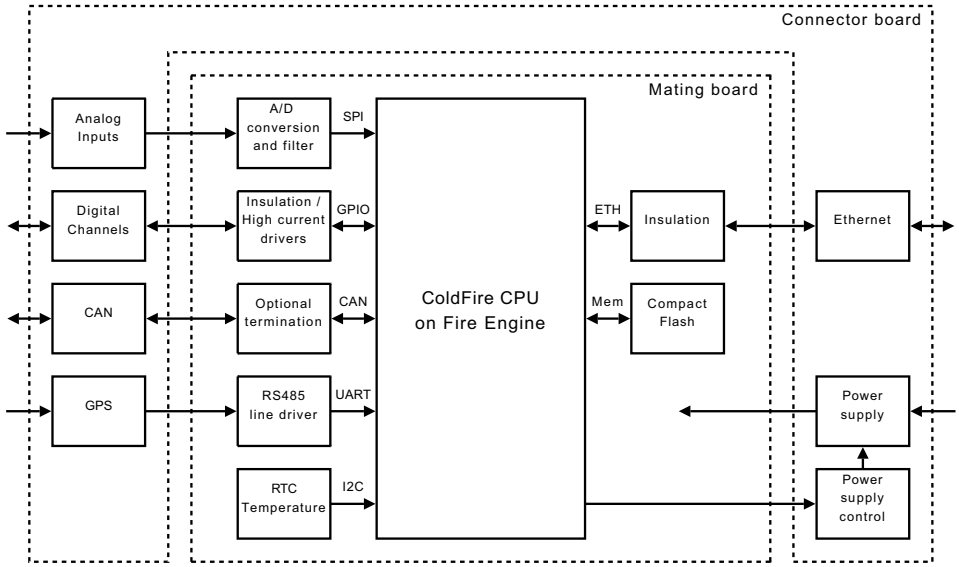


Figure 7.1: HW block diagram of the developed device

  - Hardware selectable analog 1st order low-pass filter, 117 Hz cutoff frequency
- 6 digital I/O lines with fast capture/PWM capability
- Internal temperature sensor with 0.1 C accuracy

7.1.2 Project development

There are not so many microcontrollers or CPUs capable of supporting all the different peripherals required by the project; finding one with enough processing power and 2 onboard high-speed CAN buses was especially difficult, since devices with CAN buses are usually used in vehicle environments as communication nodes and do not require high computational power. Also, the company requirement for a short development time meant it was not feasible to design the whole device from the ground up. It was thus decided to acquire a COTS[3] CPU board, while keeping in mind the small size requirement.

We eventually opted for the *Freescale ColdFire*[4] *MCF5485* CPU, a 200 MHz core with MMU, FPU, Cache, and DDR SDRAM controller,

[3]Consumer Off-The-Shelf, meaning "readily available in retail channels".

[4]See `http://en.wikipedia.org/wiki/Freescale_ColdFire` for an introduction to the family and more relevant links.

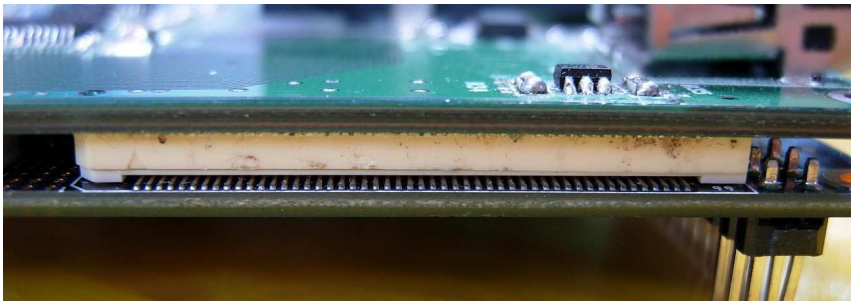


Figure 7.2: A macro detail of the mated Hirose high-density connectors.

along with Ethernet, CAN, SPI, UART, I^2C and static memory controller peripherals. The CPU board, called *Fire Engine*, was developed by Logic Product Development but marketed by Freescale itself, even in volume shipping. The board implements all the necessary core functions (voltage regulation, clocking, RAM, NOR flash for boot and code, and also a slot for CompactFlash devices) on a somewhat compact SOM-ETX footprint, which is roughly 9.5×11.5 cm. All the peripherals exported by the core are routed to 4 high-density connectors on the back of the board (higlighted in Figure 7.2).

A mating board has been designed and implemented, to add an additional CompactFlash module and all the analog processing and conversion required.

7.2 Electronics

The device, when mounted in its container box, was designed to be rugged and resist to high temperatures. For this reason, a sturdy box has been custom-made by carving an aluminum block. Two circular holes have been etched on one of the small sides, so that two military connectors could carry out the required I/O lines. The cover for this box has been again etched from a block of aluminium, with the internal side having a few protruding vertical columns, intended to dissipate heat from the active elements (mostly the CPU and a CPLD device on the Fire Engine).

This mechanical design resulted in two PCBs, mounted together as a wide “L”, the wider one providing support for the Fire Engine (see Figure 7.3). The block diagram for the developed device is shown in Fig-

Figure 7.3: The assembled device, showing the Fire Engine CPU board and the board with the Deutsch connectors and the power supply.

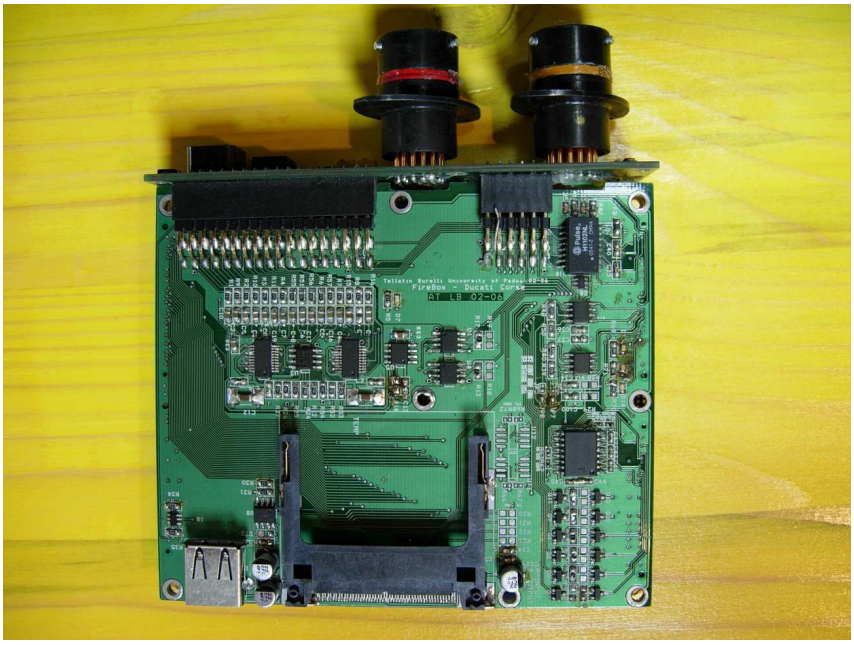


Figure 7.4: The analog electronics and CompactFlash slot on the host board. The black mating elements with the connector board are visible on top.

ure 7.1, while complete schematics for both boards are available in Appendix A.

7.2.1 Connector board

The smaller PCB board, mounted vertically, has been designed to route all I/O signals from the two round, military-grade *Deutsch AS* connectors to two long thru-hole connectors that match the ones on the host board.

The 5 V power supply has also been implemented on this board, so that EMI noise coming from the unfiltered power line is reduced as much as possible. The power supply is directly connected to the vehicle battery; to avoid excessive draining, the converter is powered up when the "ignition" contact is closed. However, once the power is applied to the board, a pull-up resistor keeps the device powered until a software-decided shutdown. This prevents loss of power or ringing on the ignition signal to interfere with the device.

Analog signal ground and power supply are also decoupled from the digital ones (they connected together only in the power supply), to reduce crosstalk and digital interference in the analog conversions.

7.2.2 Host board

The host board, visible in Figure 7.4, features on its back side the 4 high-density *Hirose* connectors that connect the Fire Engine to the outside world. These connectors are also structural in that, along with a few screw mounting holes, they physically hold the Fire Engine together with the host board. On this PCB all the electronics designed for this project is implemented:

- The analog channels are first passed through a low-pass filter, to reduce aliasing noise. They are then connected to 2 A/D converters, which are controlled by the CPU via the SPI interface. Each input is protected with reverse polarity diodes, to avoid damaging the board.
- GPIO lines used as digital inputs are insulated using MOS transistors; the same is done to provide high-current output drivers.

- One of the serial interfaces is converted to differential RS485 line levels, for remote connection to a GPS; standard RS232 levels have been tested to be too sensitive to the environment's EMI noise. Another serial port is RS232, used for local debugging.
- The two CAN buses have their own line driver/insulation, and an optional termination resistor.
- An external Ethernet line transformer has been added, as required by the Fire Engine.
- Another CompactFlash connector is provided. This is needed because the ColdFire CPU in the Fire Engine is placed right under the original CF card slot, and when a card was inserted, it was impossible to provide the required heat dissipation.

7.3 Software overview

Due to the requirements of running on a ColdFire platform, and being easily interfaced to a complex field network, Linux was the first choice as for the operating system. An appropriate hard real-time subsystem had to be added, however, to allow time-critical tasks (data logging) to be executed. At the time, the more commonplace real-time extensions to the Linux kernel were not yet ported to the ColdFire family of processors; however, a very restricted subset of the features provided by these extensions was required. For this reason, it was decided to implement a custom RT subsystem.

The custom software solution is divided in two cooperating parts, a kernel-space module that includes all the real-time data acquisition, handling and hardware-related functions, and an user-space application that takes care of storage and initial configuration.

Data is acquired in the kernel module, inside a real-time interrupt, then analyzed and filtered in a lower priority task. Finally, the data that needs to be stored is piped via a real-time FIFO (that appears to the user space as a device file) to the application, for writing to disk or sending via the network. The user space application also parses the configuration files at boot and sets the kernel module appropriately.

7.3.1 Naming and conventions

The following subsections define the terms that will be used in this discrussion. We begin with the term *channel*, which can be used in a number of different contexts, as outlined below:

hardware channels are the physical I/O lines that are implemented on the board.

input channels are the currently configured logical data sources, which do not always match exactly with the physical channels. For example, CAN input channels extract data from a specified message received from one of the CAN hardware network interfaces.

logging channels configure which channels are stored to disk while the logging system is active, and at which rate.

output channels define which data is sent to the physical output channels (CAN or digital outputs).

Listing 7.1: Input channel structure

```
typedef struct {
        ChannelType     type;
        ChanModStatus   changed;
        DataType        datatype;
        int             value;
        union {
                AnalogChData    analog;
                DigitalChData   digital;
                GPSChData       gps;
                CANChData       can;
                ConstChData     constant;
                EventChData     event;
                InternalChData  internal;
        };
} InputChannel;
```

Channel types

Internally, channels can have many different types, each with its own qualifying attributes:

analog channels are sampled via the onboard ADCs. Their qualifying attribute is the hardware analog input number.

digital channels are logic signals that are connected to the digital I/Os. Their qualifying attributes are the hardware digital IO number and a minimum pulse duration (for hysteresis purposes).

GPS channels are acquired from the GPS device via the serial link. Their qualifying attributes are the NMEA message and field numbers.

CAN channels are extracted from messages received via the CAN interface. A single message can contain more than one channel, therefore, hardware input number, message ID, and offset (in bytes) inside the message must be specified.

constant channels are used for custom events. Their qualifying attribute is the constant value they have.

event channels are boolean channels whose value depends on the current state of the respective event. Their qualifying attribute is the associated event ID.

Listing 7.2: Output channel definition

```
typedef struct {
        uint8_t          input_channel;
        ChannelType      type;
        union {
                DigitalChData   digital;
                CANChData       can;
        };
} OutputChannel;
```

A number of different data types are used: each channel can be mapped to an 8 bit, 16 bit or 32 bit field. In addition, for comparison purposes, it is necessary to know if the data is stored in 2's complement (signed) or as unsigned values.

Each input channel, as seen in Listing 7.1, is parameterized by its channel type, its data type, and all the extra qualifiers needed for its channel type.

Output channels are less complex, since they depend on a specific input channel for the data type description, and only digital and CAN outputs are supported. As seen in Listing 7.2, they only require the input channel number, output channel type, and relevant extra qualifiers for their channel.

Events

An *event* is a particular condition that, when verified, results in a reaction of the system. Session start and lap split are examples of these events. They are primarily described by means of a `Trigger` structure. An event may be specified as level-sensitive (fired each time its associated trigger is true or false) or edge-sensitive (fired only once when the trigger condition changes from false to true, or the other way around).

The many allowed conditions are listed in Listing 7.3, and fall in the following categories:

periodic triggers are triggered at regular time intervals, irrespective of any channel value or event state.

event triggers use other triggers as event sources. They can be programmed to perform basic logic operations (AND, OR, XOR) on these values, and thus can be used to create more complex actions.

Listing 7.3: Trigger definitions

```
typedef enum {
    compDisabled,            // disabled, never happens
    compPeriodic,            // periodic trigger, period val1
    compEvent,               // use event# val1 as trigger
    compEventAnd,            // trigger on both event# val1 and val2
    compEventOr,             // trigger on any of event# val1 or val2
    compEventXor,            // trigger on either event# val1 or val2
    compChanged,             // x(t) != x(t-1)
    compUpdated,             // x(t) has been updated
    compSignGreaterThan,     // signed, x(t)   >  val1
    compSignLessThan,        // signed, x(t)   <  val1
    compSignRange,           // signed, val1 < x(t) <  val2
    compUnsignChanged,       // x(t) != x(t-1), val2=tolerance
    compUnsignGreaterThan,   // unsigned, x(t) >  val1
    compUnsignLessThan,      // unsigned, x(t) <  val1
    compUnsignRange,         // unsigned, val1 < x(t) <  val2
    compGreaterThanChan_SS,  //   sgnd x(t) >   sgnd ch# val1
    compGreaterThanChan_SU,  //   sgnd x(t) > unsgnd ch# val1
    compGreaterThanChan_US,  // unsgnd x(t) >   sgnd ch# val1
    compGreaterThanChan_UU,  // unsgnd x(t) > unsgnd ch# val1
    compLessThanChan_SS,     //   sgnd x(t) <   sgnd ch# val1
    compLessThanChan_SU,     //   sgnd x(t) < unsgnd ch# val1
    compLessThanChan_US,     // unsgnd x(t) <   sgnd ch# val1
    compLessThanChan_UU,     // unsgnd x(t) < unsgnd ch# val1
    NUM_COMPARE_TYPES
} PACKED ComparisonType;
```

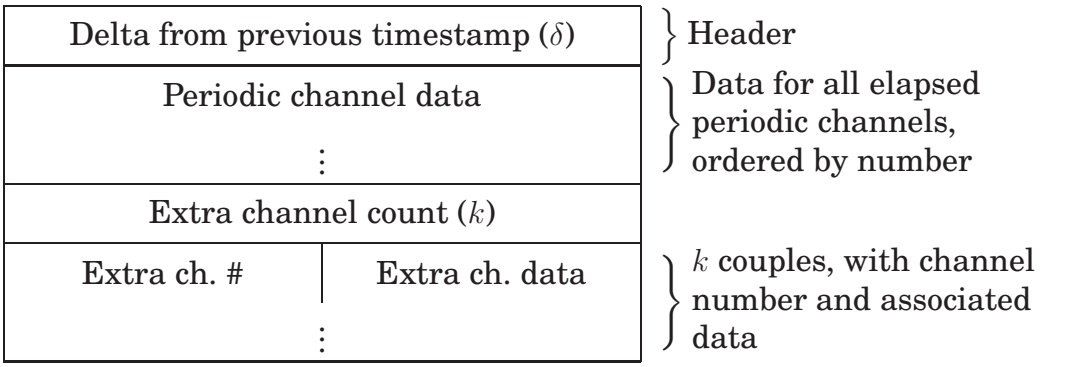
changed triggers are sensitive to a change in the value of the associated channel.

updated triggers are sensitive to when a single channel has been updated (not necessarily with a new value).

comparison triggers use integer functions to compare the associated channel. It is possible to select the compare function (less than, greater than, in range), and use a second channel instead of a fixed value for comparison.

To implement these different conditions in the fastest way, a function lookup table has been used. This results in an explosion in the number of actual comparison types, since all the comparison attributes (including the signedness of the values used), must be factored in the type enumeration.

Figure 7.5: Data storage layout (a single packet).



7.3.2 Data storage format

Logged data is stored in a compact format that requires knowledge of the associated channel logging configuration to be read. This format, shown in Figure 7.5, does not have headers explaining the contents of each data point, but instead exploits the implicit time ordering one would expect items to be stored by the logger. Should there be non-periodic channels, such as events or incoming messages, they are marked in a recognizable way. Each time step for which the logger wants to output data, the following algorithm is used:

- Store the time delta δ, measured in ticks, from the previous stored block's timestamp.[5].
- If there are input channels that are configured for periodic logging, and the period matches with the current time T, store only the actual channel data, in ascending order of channel number.
- Store the number of non-periodic channels k that are to be logged for this time interval. If there are none, store 0 and end the packet.
- For each non-periodic channel that has to be logged, store its channel # and subsequently its current value.

This algorithm has an interesting property. In the case where all channels are periodic, the resulting channel sequence is also periodic.

[5]As an optimization, if there are periodic channels at maximum system frequency, then this can be omitted as it will necessarily be always '1'.

Its period can be readily calculated as the least common multiple of the channel periods (or conversely, the greatest common divisor of the channel frequencies). This is extremely useful in the case of data corruption: by using this property one can predict when a new periodic cycle will start, and thus recover the stream from that point on. These instants are called *synchronization points*.

For example, suppose we have three logging channels, channel 1 storing 16 bit data at 500 Hz, channel 2 storing 8 bit data at 200 Hz, and channel 3 storing 32 bit data but being activated at random times (in this example, at $T = 7$ ms and $T = 15$ ms). For these conditions, the synchronization period is 10 ms, and the actual output would be as shown in figure Figure 7.6.

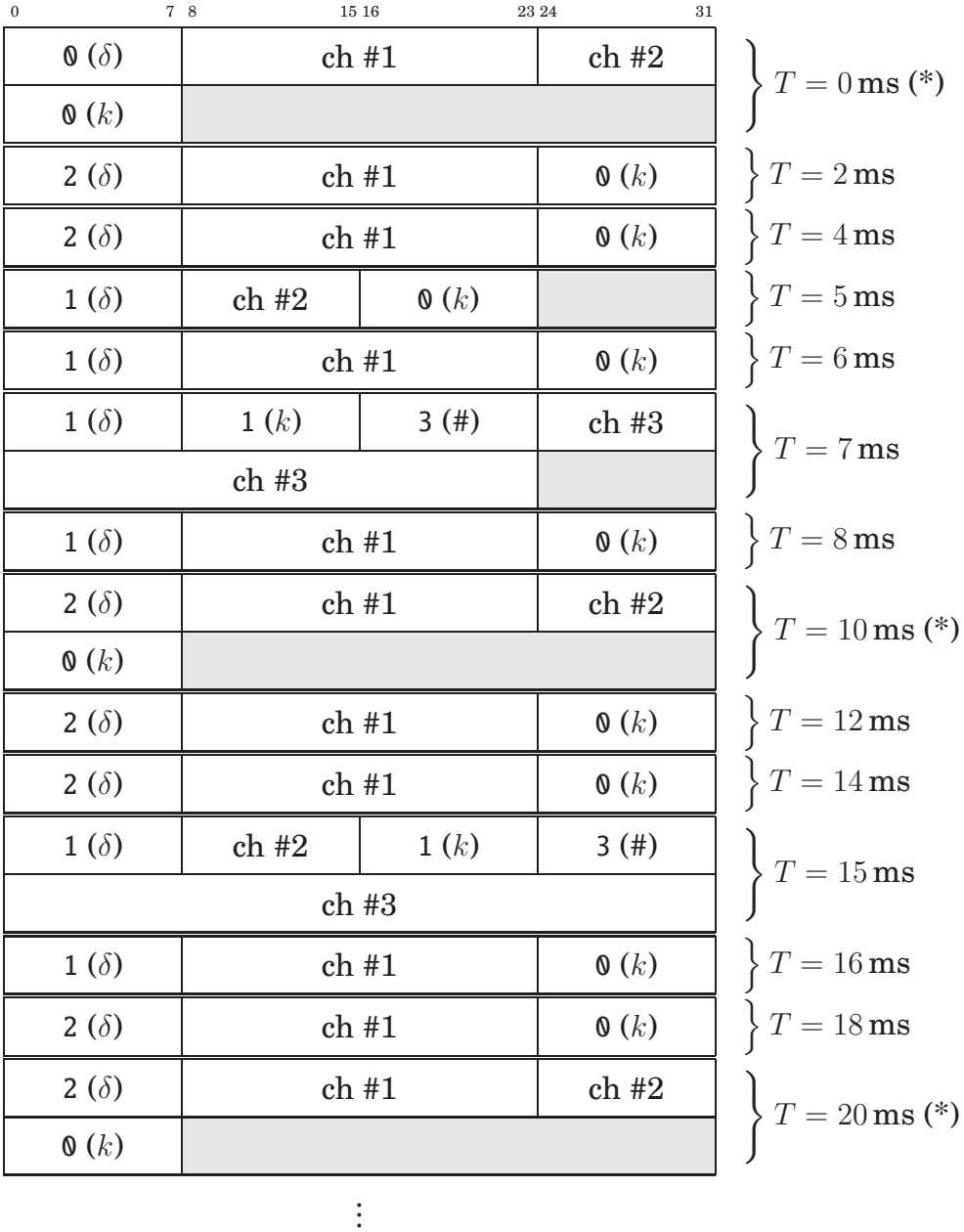
The recording starts at time $T = 0$ with all the periodic channels; no periodic channel logging occurs for $T = 1$ so the next data is stored only for $T = 2$ and $T = 4$, where channel 1 is logged. At $T = 5$ channel 2 is expected, so δ is now 1. Note that at $T = 7$ ms there were no expected periodic channels, but channel 3 had to be stored (e.g. a message has arrived from the CAN bus), so k immediately followed δ. At $T = 15$, instead, both periodic and non-periodic channels had to be stored.

From this example you can also see how the channel period is given by the synchronization points. If the recording started at $T = 10$ instead of $T = 0$, nothing would change apart from the initial δ.

The overhead imposed by this data structure, even if in this specific example looks important, is in fact limited. Usually, the bulk of the recorded data are channels which are logged periodically at the highest possible frequency, while non-periodic channels have comparatively long mean periods.[6]

[6]Think about the lap split signal example given before.

Figure 7.6: Data storage example. The logging settings that produced this output are discussed in the text. Packets marked with (*) are also synchronization points.



7.4 Software organization

At power up, the CPU starts executing code from the 2 MB NOR flash that is used for booting. A basic bootloader[7] stored there takes care of loading and starting the Linux kernel from the same memory. The kernel has been programmed to use the second NOR flash chip (16 MB in size) by splitting it in two JFFS2 partitions: the bigger one (15 MB) is read-only and holds the root file system with the application code, and the rest stores the current setup data and basic system logfiles. The recorded data is instead stored in the 2 GB onboard CompactFlash card.

A simple, SysV-like boot process sets up a small ram-disk for `/var` and `/tmp`, and loads the required network daemons:

dhcpcd waits for a network connection, and auto-configures the IP address via DHCP if such a server is present.

betaftpd and **smbd** handle Samba (Windows networking) and FTP services, that allow easy access to the configuration and data folders.

boas provides an HTTP Web interface with CGI support, which is used as the device's status and debug interface.

rsyncd is used to sync the root partition, for firmware update purposes.

syslogd provides storage for the kernel and system log.

Finally, the real-time kernel module and the corresponding control application are loaded and run.

7.4.1 The kernel module

The code in the kernel module is divided between several files and subdirectories:

`base/` contains the general logic for the kernel module,

`sampler/` implements the hard real-time logic for acquiring analog and digital inputs,

[7] The default Freescale bootloader is dBUG, but we opted for the LogicLoader software from LogicPD. All these bootloaders are freely available from their producers.

can/ implements the CAN device driver to acquire messages from the bus,

gps/ implements the GPS driver, parsing NMEA sequences coming from the serial line,

dsp/ handles the optional digital signal processing of the received data.

Each sampling tick, the following actions are handled by the code inside the module:

- Regardless of the current logging settings, all high speed, real-time hardware input channels (analog, digital and CAN) are sampled for their current value. Low speed data from the GPS is also acquired and processed shortly after.
- Digital signal processing performs all configured activity (moving average, FIR or custom digital filters) on the current values of all the input channels.
- For each of the configured inputs, their value is compared to the associated hardware channel, and flags are updated accordingly.
- All of the configured triggers' conditions are evaluated, updating their "active/idle" flag.
- Based on the current value of some triggers, the systems performs special actions, such as start/stop logging, etc.
- The logging subsystem scans all active channels, logs each one according to their configured schedule, and continuously stores the resulting data in a ring buffer.
- Finally, configured output channels are evaluated, and the appropriate action is taken if their associated event is active.

7.4.2 The user-space application

The user space application loads the current settings from the on-board Flash memory. These settings are stored in hand-editable text files, as follows:

inputs.conf describes currently configured input channels and their attributes (visible in Listing 7.1).

events.conf declares conditions for the pre-defined actions (start/stop session, lap, etc) as well as custom ones obtained by logic composition. See Listing 7.3 for the complete list.

logs.conf defines logging frequencies (or matching events) for the configured input channels.

outputs.conf configures output channels, specifying contents, activation condition, and their attributes (visible in Listing 7.2).

These files are used by both the real-time application and the PC-side data converter; additionally, the latter also uses **resample.conf**, which describes how to resample the non-periodic channels when the output format supports only periodic streams.[8] (This feature can also be effective to downsample high-frequency, periodic channels so that the resulting data is smaller and easier to process.)

After setting the parameters in the kernel module, the application locks in a `read()` function call, waiting for data and commands from the kernel module. Once a new session is started by activating its trigger, the logging system starts sending data down the pipe, and the user space application stores this data, along with a copy of the above configuration files, in a session-numbered directory.

The actual recorded data is stored in a number of files, one per lap, and is compressed with the Open Source library `zlib` to optimize the storage space.[9]

[8] WinTAX, one of the most famous visualization and engineering software used in racing environments, has this constraint on its data files.

[9] Most redundancy is eliminated already at the lowest level of compression, reducing file size by at least 30 %.

7.5 Real-time implementation

The following section will detail some of the real-time implementation issues encountered while developing this system, and how these have been solved.

7.5.1 Interrupts on the ColdFire

The 5485 has a very flexible interrupt controller, much like the ones in the the M68000 family, where a 3-bit encoded interrupt priority level is sent from the interrupt controller to the core, providing 7 levels of interrupt requests. Level 7 represents the highest priority interrupt level, while level 1 is the lowest priority (0 being absence of interrupts). The processor samples for active interrupt requests once per instruction by comparing the encoded priority level against a 3-bit interrupt mask value (I) contained in the machine's status register (SR). If the priority level is greater than the SR[I] field at the sample point, the processor suspends normal instruction execution and initiates interrupt exception processing. Level 7 interrupts are treated as non-maskable within the processor, while levels 1-6 may thus be masked depending on the value of the SR[I] field.

During the interrupt exception processing, the CPU enters supervisor mode, and then fetches an 8-bit vector from the interrupt controller. The fetched data provides an index into the exception vector table that contains 256 addresses, each pointing to the beginning of a specific exception service routine. Most of these are reserved for processor and MMU events and have fixed priorities; however, 63 are associated with internal peripherals, and each of these has a unique interrupt control register to define the software-assigned levels and priorities within the level.

The peripheral set includes two "slice timers", used to provide short-term periodic interrupts. Each timer consists of a 32-bit counter, with no prescaler, that counts count down from a prescribed value. When they reach zero and expire/interrupt, they are automatically reloaded, starting a new cycle.

7.5.2 Interrupt scheduling

By programming the interrupt controller appropriately, it is possible to tweak interrupt priorities so that real time tasks have fixed latencies. The standard implementation of Linux on the ColdFire platform does not change the default values of the peripheral interrupt priorities, as all interrupts are usually considered maskable and treated equally by the standard Linux kernel; in particular, this applies to the slice timer 0, which was used to provide the usual 100 Hz system tick.

Using the same strategy for our periodic timer, initially we tried associating a stock Linux ISR to the second slice timer. By monitoring the output waveforms with a scope, however, as expected the jitter results were very poor (especially when the system was not idle). This was clearly due to the fact that while the Linux kernel was executing in a non-preemptible zone, the sampling interrupt had to wait until completion of the critical section before being serviced.

The ideal solution was to raise the priority of the second slice timer up to that of a non-maskable interrupt. However, that means the CPU may handle the interrupt even while "all" interrupts have been masked by the Linux kernel. It is thus imperative not to interfere with the kernel operations, while in this ISR: if any part of the kernel is called from here, there is a chance for deadlock or data corruption. Extra care must be taken, since there are subtle and non-obvious interactions, as will be described in the following paragraphs.

With this change, the resulting jitter has been almost completely cleared; the small remaining jitter on the sampling clock is mostly due to the underlying base clock and the inevitable PLL jitter.

This real-time code needs to be as quick as possible to minimize the impact on the system performance. Also, as stated above, being called outside of the kernel context makes it impossible to use any of the functions provided by the Linux kernel. For these reasons, this ISR only takes care of channel sampling, which is the most sensitive real-time task; data is then placed in a buffer to be later retrieved and analyzed by a lower priority, soft real-time task, which runs synchronized with the Linux kernel. This has been implemented by generating (via the Coldfire interrupt controller) a simulated interrupt request for an unused interrupt source, and assigning higher priority than all other interrupts (but still not NMI). Using standard Linux kernel methods, the bulk process-

ing routine is associated with this interrupt; as such, it will be handled just after the NMI has completed (if interrupts were enabled), or at the end of the current critical section (when the interrupts are re-enabled).

The original system tick interrupt source has been disabled to reduce the interrupt contention on the CPU. This interrupt is instead scheduled by the real time ISR, exploiting the fact that the low-speed tick is an integer fraction of the sampling frequency. This way, it is also guaranteed that the Linux kernel scheduling will be done immediately after the real-time and soft real-time data processing occurs, so that the user-space application will have priority over the other Linux tasks.

Linux ISR handler

The interrupt handling functions of the Linux kernel are designed to allow multiple devices to share a single IRQ. To make this possible, each interrupt is handled by the same Linux-provided ISR stub, which checks the internal interrupt tables and calls the associated list of ISRs. The stub also does other minor things, like incrementing the interrupt count table that is visible in `/proc/interrupts`.

This behaviour has to be avoided as the stub was not designed to be reentrant. For this reason, a custom form of the `request_irq()` function, called `request_hard_irq()`, was implemented. This function substitutes the Linux low level handler with its own very small one, which just calls the requested ISR and then executes the "return from interrupt" assembly instruction. This double jump is still needed to allow a standard C function (with its compiler-added preamble/epilog code) to be called without modifications in interrupt context.

MMU

Linux in its unmodified form uses the MMU for security and memory management purposes, and the MMU inside the Coldfire chip is fully supported.

Accesses from the CPU are mapped from virtual addresses to physical ones using a hardware page table of fixed size, which has only room to store the most used entries. When an address is requested which is outside the currently programmed page table entries, a page fault exception is thrown and the standard Linux page fault handler is called.

This handler searches a much bigger page pool (on standard machines, for example, handles also the case where a page has been swapped out to slower storage) to find out the physical position of this page, overwrites the least recently used entry in the page table with this information, and resumes the application.

Unfortunately, all of this is very sensitive to reentrancy. At the time the real-time ISR is called, the Linux kernel might be busy altering page table structures, and so a walk of these tables could have unpredictable results. For this reason, both the code and the data which is used by the processor inside the real time ISR has been placed in a fixed-address SRAM area, which is directly accessed by the ColdFire CPU bypassing the MMU and its possible issues.

7.5.3 Interprocess communication

The current data samples, taken by the real-time task, are stored in a double buffer which is later fetched from the soft real-time task. Ideally there would never be a reason for more than one buffer, but delays introduced by the Linux kernel's critical sections may cause the bulk processing task to not terminate before another real-time ISR is executed. Overflow of this buffer is also monitored for completeness, but this has never happened.

The real-time FIFO that handles communication between the real-time task and user space (actually, the kernel-space implementation of the `read()` system call), is more interesting.

This *buffer cache* is used at runtime to continuously store data, even while the user-space application is not serviced for some time, but conveniently doubles as a "pre-trigger buffer": once the system is configured, data is always appended to the buffer even if logging is not yet active, so that after an initial ramp the whole cache is always full (or nearly full). This makes it easy to store data even before the actual "start data logging" event triggers. However, due to the peculiar format used in data storage, the data stream can't be started from any of these measured samples, but only where the timestamp is an exact multiple of the cyclic interval. Data must thus be discarded up to a synchronization point, in bigger blocks than it was previously stored.

This FIFO is implemented by a single linked list of `DataBlockNodes`, each holding a fixed amount of data, statically allocated at module load

time. Each of the data blocks stored by the real-time task begins with a `BufHeader`, which records the real-time timestamp, the length in bytes of the whole data block, the current system flags and a simple checksum value. Both the kernel and the user-space program check consistency in these fields for maximum security.

Adding data blocks to the buffer cache

During the execution of the real-time task, when the logger submits a data block for addition, it is split in `DataBlockNodes` and stored by the following algorithm:

1. If the system is in the pre-trigger mode and the read block is stale (older than the maximum required age), or the queue is full, the block is discarded advancing the read pointer to the next block header, freeing a number of nodes.

2. If the system is running, and the queue is full, this means that some of the data that had to be stored is going to be overwritten. The read pointer is advanced to the next block header that is also a synchronization point.[10]

3. At this stage the write pointer has room for at least one node. The current one is filled with data and marked as full.

4. If the data that had to be stored didn't fit in this node, repeat from 1 with the remaining data.

The rationale behind step 2 above is that, in this case, it is important for the lower levels to detect the "skip", and also make useful use of the following information (since an undetected jump in the data stream would confuse the decoding state machine). You can use buffer headers to compare subsequent timestamps and detect a skip. But once the skip is found, since the information in the queue is already in the storage format, the periodic data stream is lost—the only way to obtain useful information is by starting again from a synchronization block. Furthermore, since the queue was full, discarding a large block of data reduces the chances for many little consecutive skips, something that would happen if the buffer was slowly consumed.

[10] Recall the discussion in Section 7.3.2.

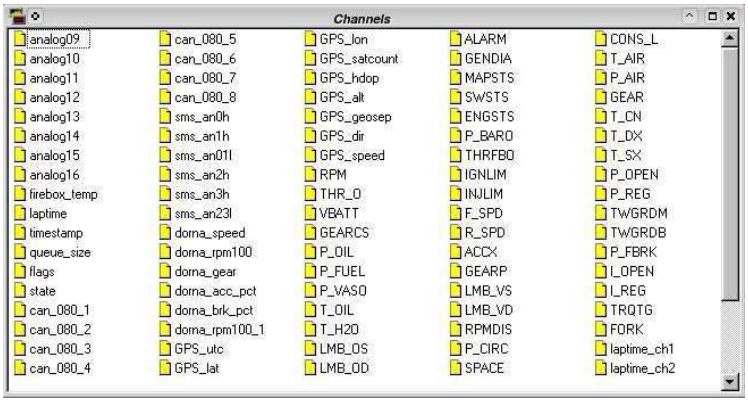


Figure 7.7: Channels acquired during a test at the Mugello circuit.

Reading from the buffer cache

If the system is idle, or the cache is empty, the reader process suspends in a wait queue, to be awaken by the soft real-time task when a block has been written in the FIFO.

While reading buffers from the cache, however, the real-time tasks could be activated and write data to the FIFO again (and thus could move the read pointer, overwriting the same data that was going to be read). The read pointer must always refer to the beginning of a header, because otherwise the update algorithm above would not fetch correct information. But it is not feasible to disable interrupts for the whole length of the `read()` function call, because that could cause data blocks to be skipped at soft real-time level. Thus, the interrupts are disabled while one data block is copied, and allowed to run again at the end of each block.

The check for correct timestamp sequencing is done in the user space program: if there is a gap in the received stream's timestamps, the program forcibly closes the current file and starts a new one, forcing reader state machines to restart from a synchronization point.

7.6 Conclusions

The device has been tested twice during practice sessions of the Ducati Corse team at the Mugello International Circuit. After being mounted on the Ducati GP7 bike, and having passed basic "survival" tests, we

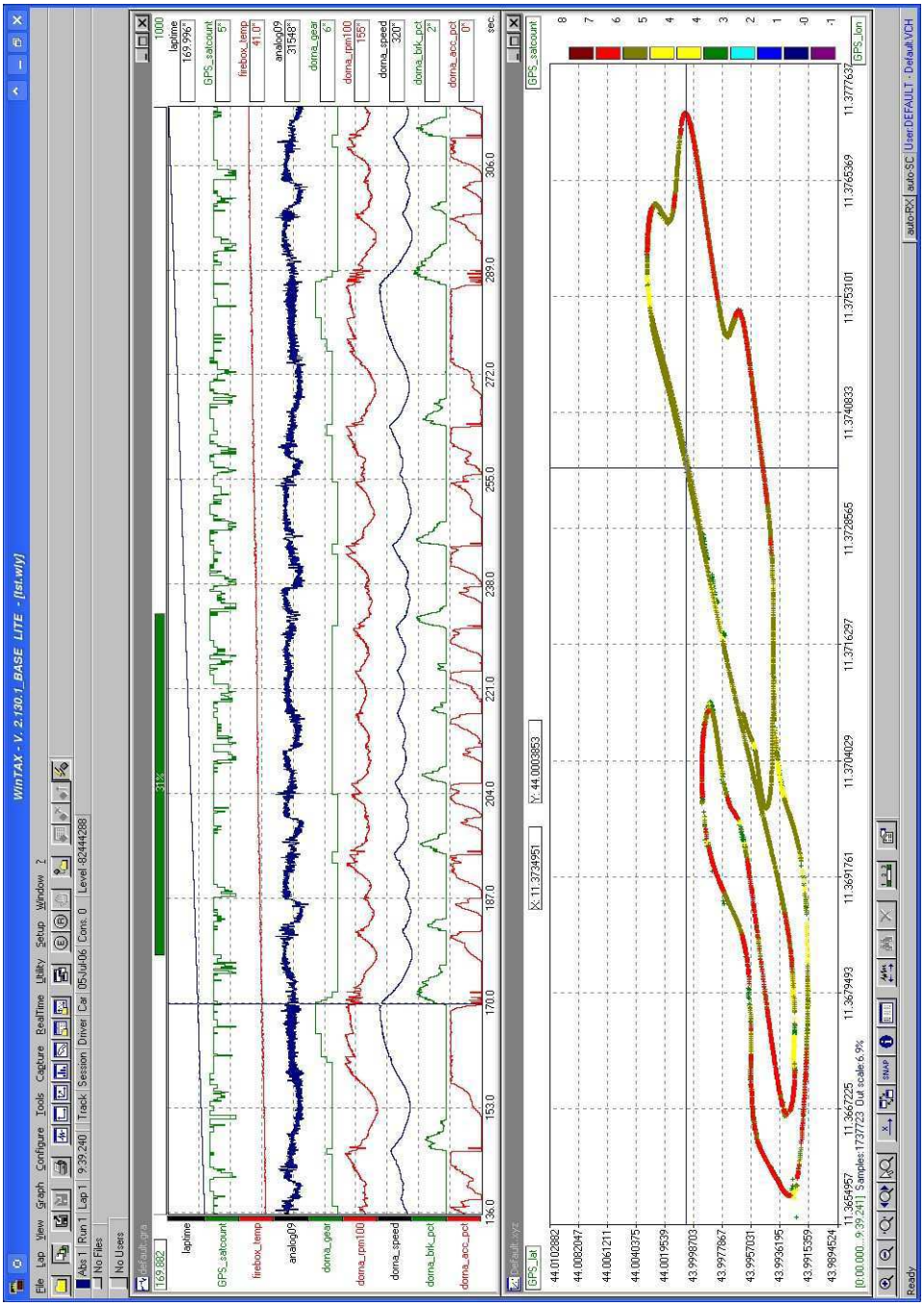


Figure 7.8: The WinTAX software displaying lap data recorded at the Mugello circuit. The upper half of the window shows some of the time-correlated signals recorded; the peak speed point (highlighted, close to 170") is also located on the GPS point display in the bottom half of the window.

have succesfully recorded several sessions' worth of data, making use of all the hardware channels. The analog recording performance was better than the current data logger used by the team, and the capability of recording non-periodic data sparingly was a nice plus.

Figure 7.5.3 shows an actual snapshot of WinTAX, the software used by the team for their engineering analysis, displaying a number of channels acquired by the presented device. The lower half of the window shows a GPS point cloud that highlights the shape of the famous Mugello circuit. At the highlighted instant (close to 170 ") the bike was at its maximum speed for the lap, at the end of the runway.

We also tested CAN communication with the ECU, by sending processed data (the current GPS coordinates, translated to metric units) to the ECU via a special CAN packets. The device acknowledged, and the data could be successfully read via the ECU's control software.

As originally stated, this platform thus enables much more complex algorithms to be tested on the field: CPU usage barely reached 20 % during all of our tests, and the ColdFire platform is ready for heavy throughput, DSP-like algorithms, since it has dedicated DSP instructions such as MAC (Multiply and ACcumulate) and a floating-point unit.

One of the biggest open issues is having real-time, accurate bike attitude information. New algorithms, merging GPS and inertial sensors, could increase the overall knowledge of the ECU enabling unprecedented levels of performance to be obtained.

Chapter 8

Conclusions

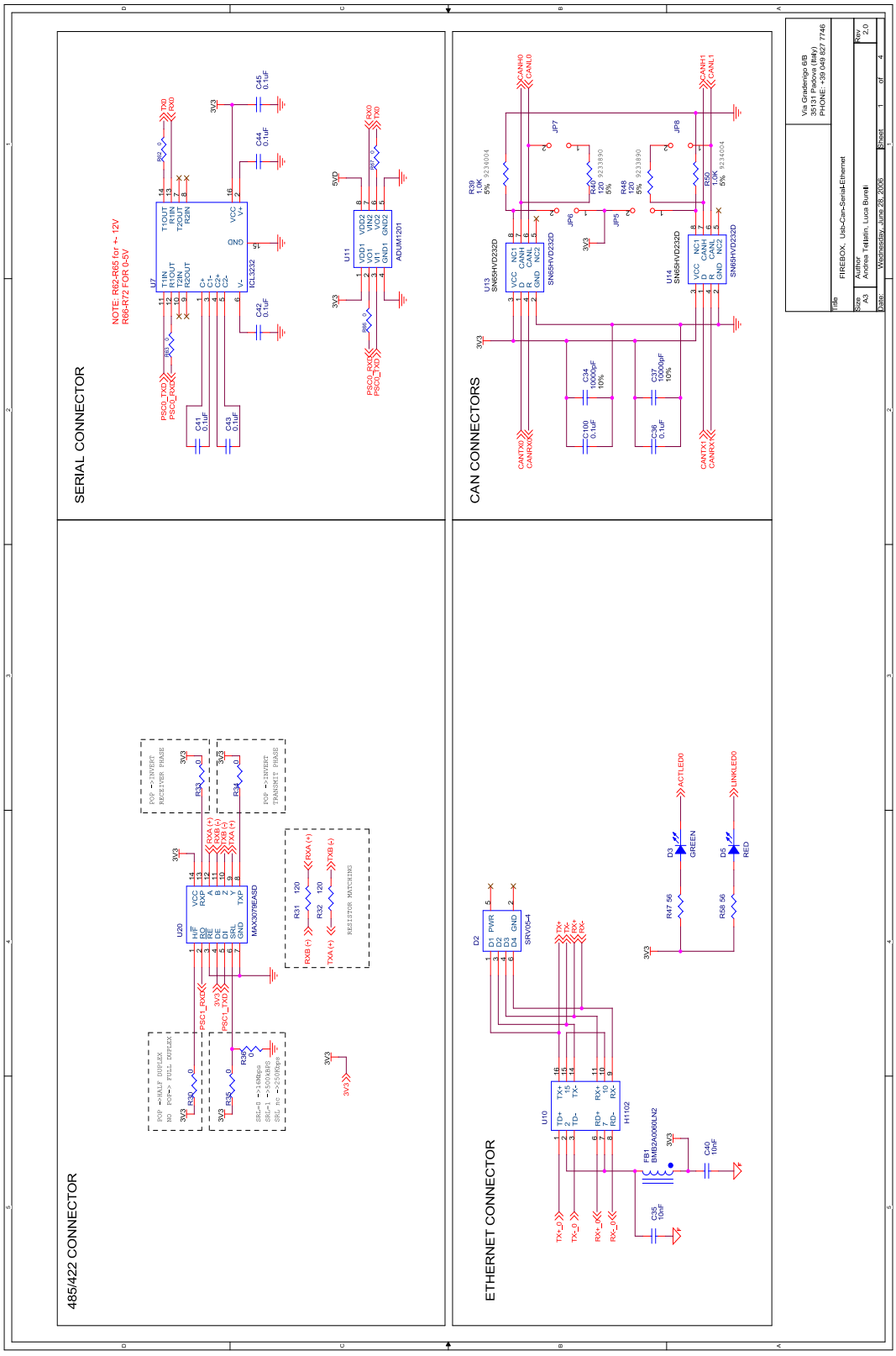
This work presented a combination of different aspects of the software and the architectures used for developing such systems, without the ambition of being exhaustive but with the aim of giving the flavor of the complexity and the interaction of the underlying structures, methodologies, and procedures. Architectural and methodological issues in the designs have been presented together with a deepening insight into the technological and development aspects of the implementation of the algorithms and design of custom, application specific boards.

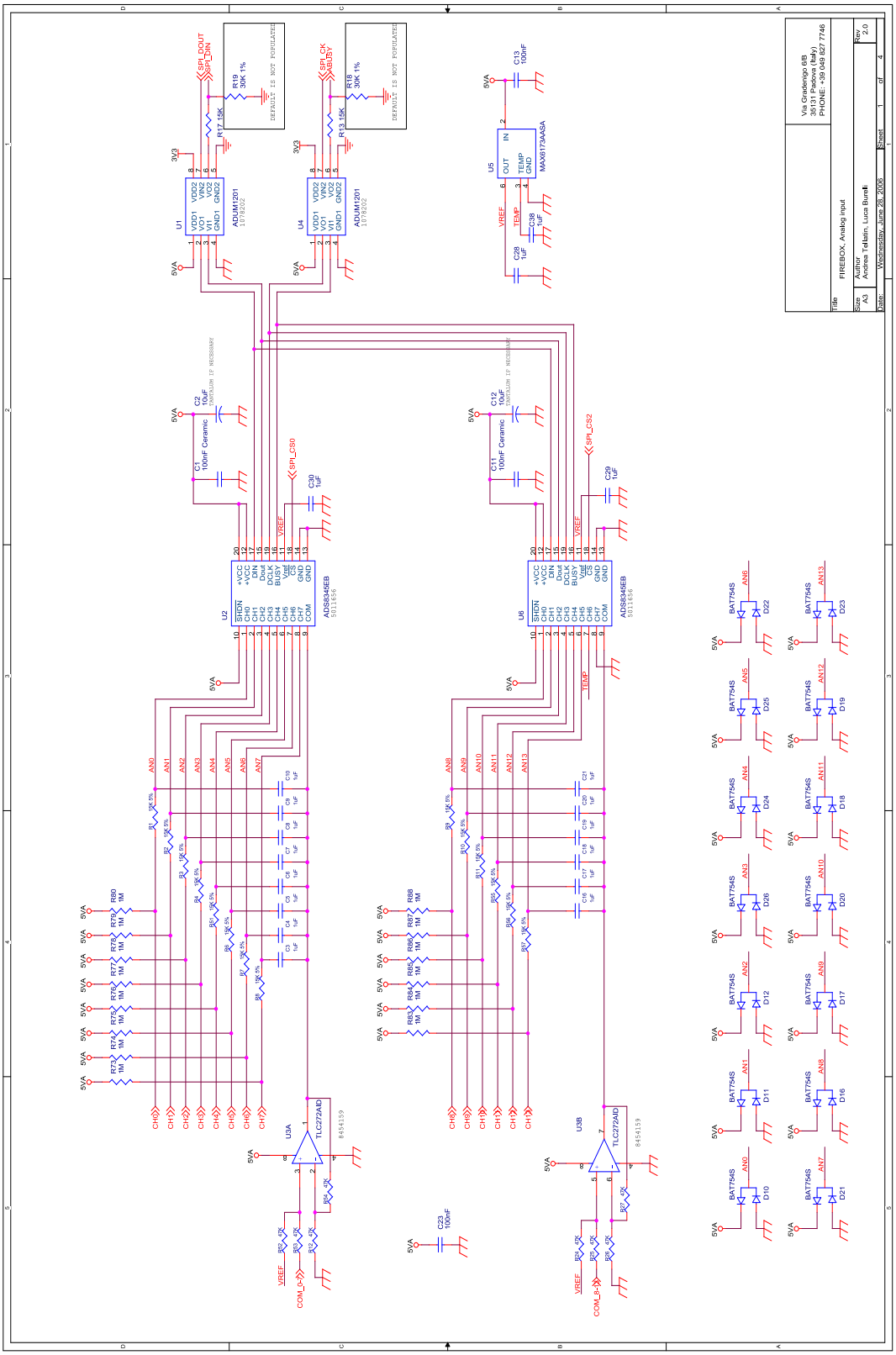
The four vehicle lifecycle phases introduced in the introduction to Chapter 5 (design, development, simulation and testing) already have available commercial software architectures. Many different middleware solutions provide very useful, specialized abstractions which reduce development time and costs, while allowing greater flexibility, and ultimately resulting in higher overall design robustness.

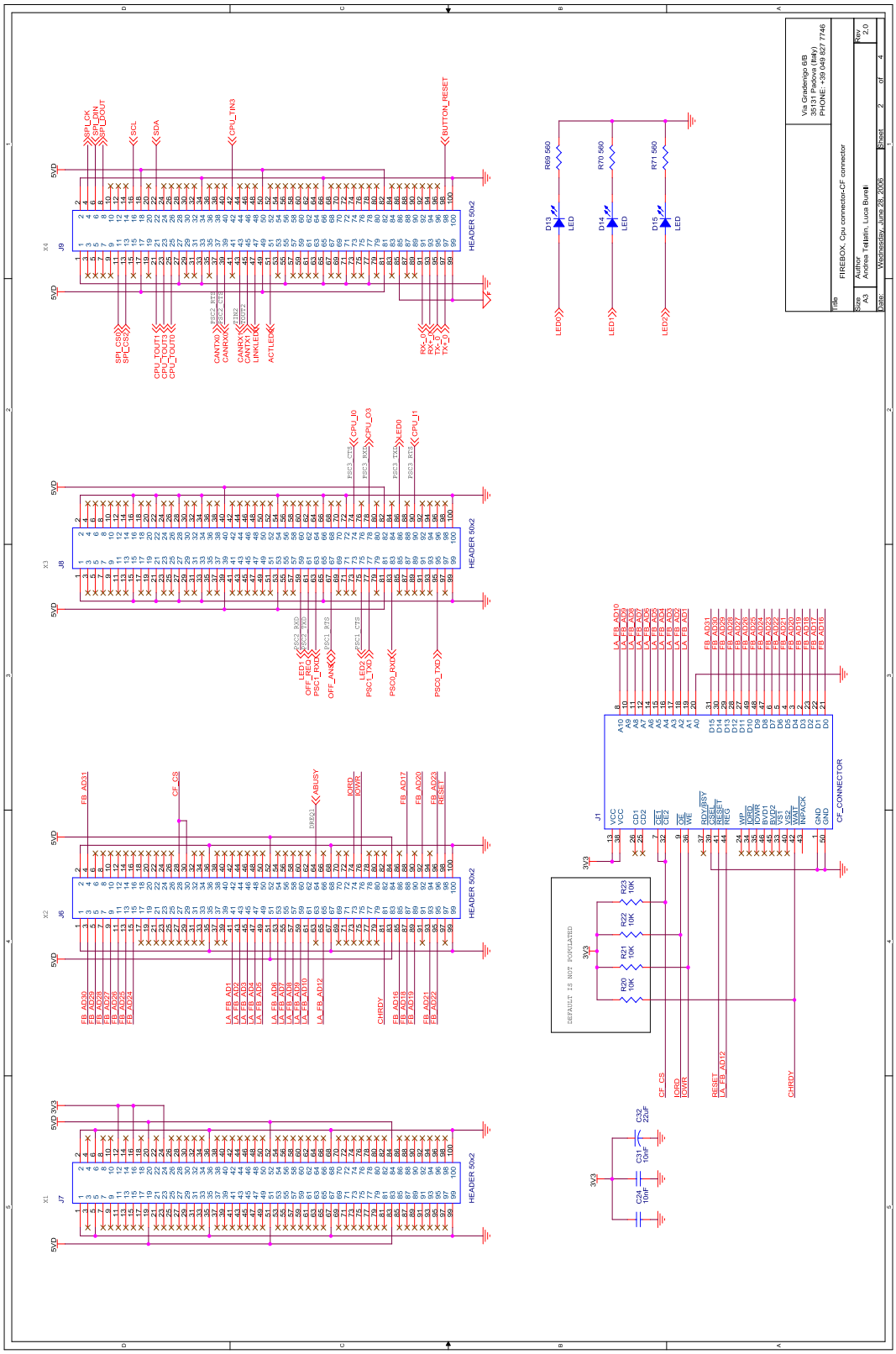
Little has been done, however, in the way of creating a single, common platform for these tasks, so that by integrating these steps together, deeper insight in the workings of automotive electronics could be achieved, enabling ever increasing performance returns.

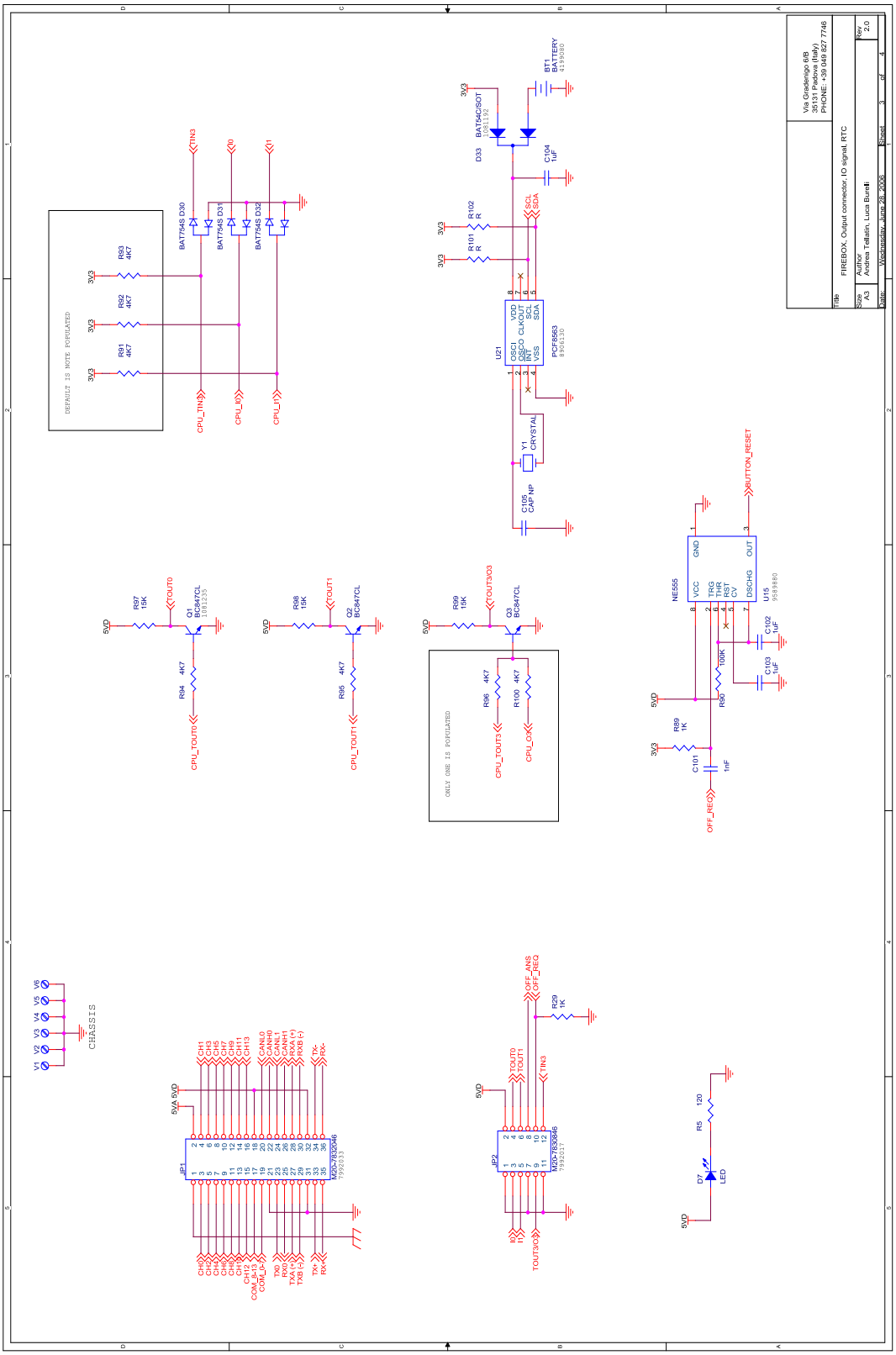
Appendix A

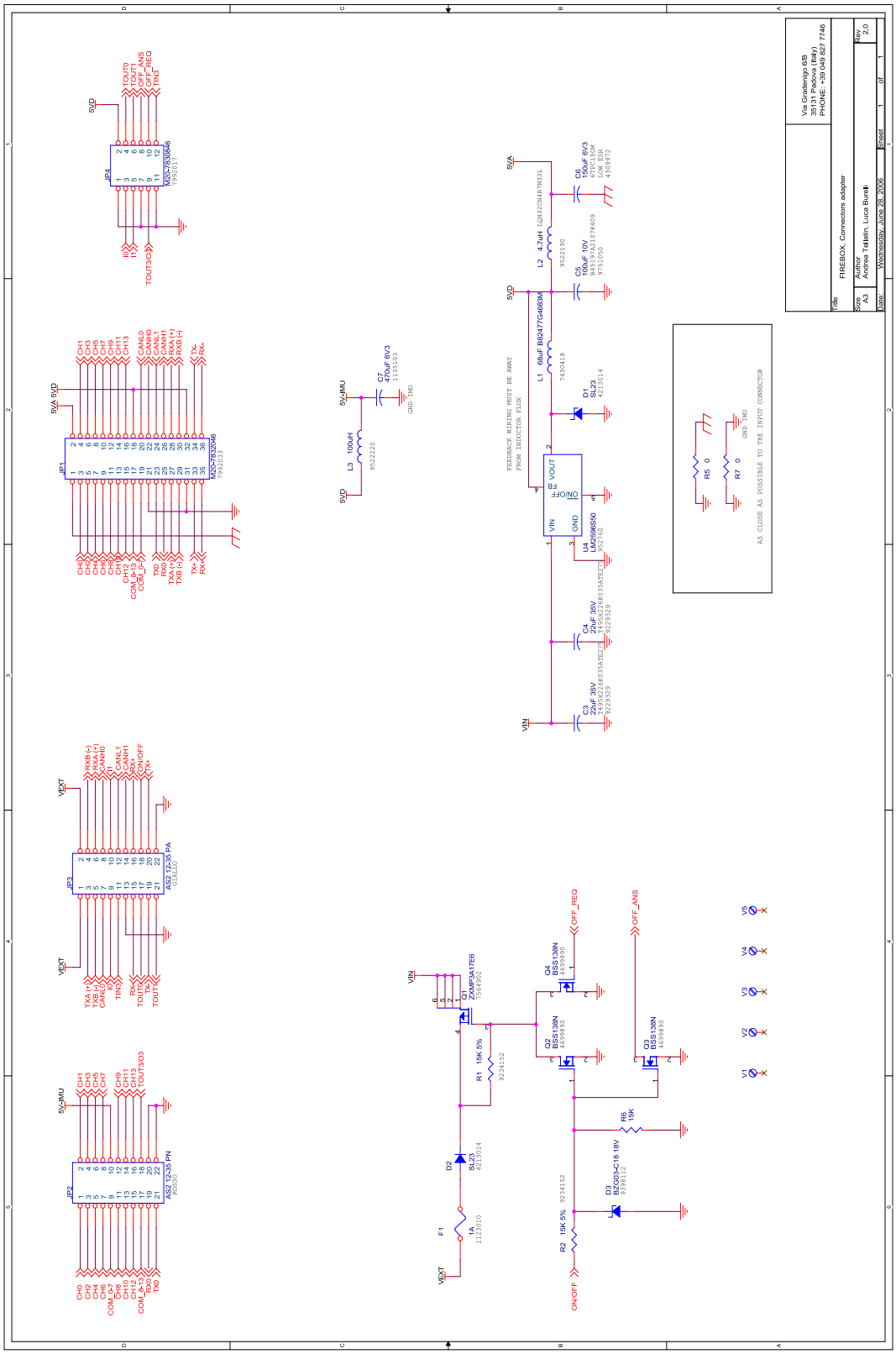
Data Acquisition System Schematics


SERIAL CONNECTOR
CAN CONNECTORS
485/422 CONNECTOR
ETHERNET CONNECTOR
NOTE: R60-R65 for +- 12V
R66-R72 FOR 0-5V
U7
ICL3232
U11
ADUM1201
U13
SN65HVD232D
U14
SN65HVD232D
U20
MAX3078EASD
U10
H1102
D2
SRV05-4
D3
GREEN
D5
RED
FB1
BMB2A0600LN2
RESISTOR MATCHING
Via Gradenigo 6/B
35131 Padova (Italy)
PHONE: +39 049 827 7746
Title: FIREBOX, Usb-Can-Serial-Ethernet
Size: A3
Author: Andrea Tatalin, Luca Bursi
Rev: 2.0
Date: Wednesday, June 28, 2006
Sheet: 1 of 4


FIREBOX, Analog input
Author
Andrea Tellatin, Luca Buratti
Wednesday, June 28, 2006
Via Gradenigo 6/B
35131 Padova (Italy)
PHONE: +39 049 827 7746
ADUM1201
ADS8346EB
TLC272AID
MAX6173AASA
BAT754S
TEMP
VREF
SPI_CS0
SPI_CS2
SPI_DOUT
SPI_DIN
SPI_CK
SPI_BUSY


HEADER 50x2
CF_CONNECTOR
DEFAULT IS NOT POPULATED
Via Gradenigo 6/B
35131 Padova (Italy)
PHONE: +39 049 827 7746
Title: FREBOX, Cpu connector-CF connector
Author: Andrea Tatalin, Luca Bursi
Wednesday, June 28, 2006


Via Giradengo 6/B
28010 Cavaglio d'Agogna (NV)
PHONE: +39 049 8277746
Title
FIREBOX, Output connector, IO signal, RTC
Size
A3
Author
Andrea Tellatin, Luca Bunelli
Rev
2.0
Date:
Wednesday, June 28, 2006
Sheet
3
of
4
DEFAULT IS NOT POPULATED
ONLY ONE IS POPULATED
CHASSIS
BAT754S D30
BAT754S D31
BAT754S D32
BAT54C/SOT
D33
BATTERY
BT1
R91 4K7
R92 4K7
R93 4K7
R101 R
R102 R
C104 1uF
U21
PCF8563
Y1
CRYSTAL
C105 CAP NP
Q1 BC847CL
Q2 BC847CL
Q3 BC847CL
R94 4K7
R95 4K7
R96 4K7
R100 4K7
R97 15K
R98 15K
R99 15K
NE555
U15
R89 1K
R90 100K
C101 1nF
C102 1uF
C103 1uF
R5 120
D7
LED
JP1
JP2
M20-7832046
M20-7932017
BUTTON_RESET
OFF_REQ
OFF_ANS
R29 1K
CPU_TIN3
CPU_I0
CPU_I1
CPU_TOUT0
CPU_TOUT1
CPU_TOUT3
CPU_O3
I0
I1
TOUT3/O3
5VD
3V3
5VA 5VD



Bibliography

[1] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[2] W. F. Milliken and D. L. Milliken, *Race Car Vehicle Dynamics*. Warrendale, PA, USA: Society of Automotive Engineers, Inc., 1995.

[3] J. D. P. Z. Berman, "Advances in motorcycle design and control," *IEEE Control Systems Magazine - special issue*, vol. 26, no. 5, pp. 510–517, 1998.

[4] Z. Berman and J. D. Powell, "The role of dead reckoning and inertial sensors in future general aviation navigation," in *IEEE Position Location and Navigation Symposium*, 1998, pp. 510–517.

[5] R. Da, G. Dedes, and K. Shubert, "Design and analysis of a high-accuracy airborne gps/ins system," in *Int. Technical Meeting of the Satellite Division of the Institute of Navigation (ION GPS-96)*, 1996, pp. 955–964.

[6] D. Gebre-Egziabher, R. C. Hayward, and J. D. Powell, "A low-cost gps/inertial attitude heading reference system (ahrs) for general aviation application," in *IEEE Position Location and Navigation Symposium*, 1998, pp. 518–525.

[7] D. Gebre-Egziabher, J. D. Powell, and P. Enge, "Design and performance analysis of a low-cost aided dead reckoning navigation system," *Gyroscopy and Navigation*, 2001.

[8] NIMA, "World geodetic system 1984, its definition and relationships with local geodetic systems," Department of Defense, Tech. Rep. TR8350.2, Third Edition, 1997.

[9] J. Ryu and J. C. Gerdes, "Intergrating inertial sensors with gps for vehicle dynamics control," *Journal of Dynamic Systems, Measurement and Control*, June 2004.

[10] H. Qi and J. B. Moore, "Direct kalman filtering approach for gps/ins integration," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 38, no. 2, April 2002.

[11] F. Nori and R. Frezza, "Accurate reconstruction of the path followed by a motorcycle from the on-board camera images," in *IEEE Intelligent Vehicles Symposium*, 2003.

[12] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 8, no. 6, pp. 679–698, 1986.

[13] R. O. Duda and P. E. Hart, "Use of the hough transform to detect lines and curves in pictures," Artificial Intelligence center, Tech. Rep. 36, 1971.

[14] D. H. Ballard, "Generalizing the hough transform to detect arbitrary shapes," in *Readings in computer vision*. Morgan Kaufmann, 1987, pp. 714–725.

[15] H. B. Pacejka, *Tyre and Vehicle Dynamics*. Oxford: Butterworth-Heinemann, 2002.

[16] C.-S. Liu and H. Peng, "Road friction coefficient estimation for vehicle path prediction," *Vehicle System Dynamics*, vol. 25, no. 413-425, 1996.

[17] J. S. B. Mitchell and C. H. Papadimitriou, "The weighted region problem: finding shortest paths through a weighted planar subdivision," *Journal of the ACM*, vol. 38, no. 18-73, January 1991.

[18] H. Haddad, M. Khatib, S. Lacroix, and R. Chatila, "Reactive navigation in outdoor environments using potential fields," in *IEEE Conference on Robotics and Automation*, vol. 2, no. 1232-1237, 1998.

[19] Y. K. Hwang and N. Ahuja, "A potential field approach to path planning," *IEEE Transactions on Robotics and Automation*, vol. 1, no. 23-32, 1992.

[20] H. Freeman, "Computer processing of line-drawing images," *Computing Surveys*, vol. 6, no. 57-97, March 1974.

[21] D. Ferguson and A. Stentz, "The delayed d* algorithm for efficient path replanning," in *IEEE Conference on Robotics and Automation*, no. 2045-2050, 2005.

[22] R. Frezza, G. Picci, and S. Soatto, "A lagrangian formulation of nonholonomic path following," in *IEEE Conference on Vision and Control*, 1998.

[23] M. Pasquotti, *Tecniche di Model Predictive Control per la guida di un'auto*. DEI-University of Padova, 2004.

[24] M. Bertozzi, A. Broggi, and A. Fascioli, "Stereo inverse perspective mapping: Theory and applications," *Image and Vision Computing Journal*, vol. 16, no. 585-590, 1998.

[25] L. Henriksen and E. Krotkov, "Natural terrain hazard detection with a laser rangefinder," in *Proceedings of IEEE International Conference on Robotics and Automation*, 1997.

[26] J. Roberts and P. Corke., "Obstacle detection for a mining vehicle using a 2-d laser," in *Proceedings of Australian Conference on Robotics and Automation*, 2000.

[27] "Fuzzy logic toolbox," *Matlab Reference guides*, 2002.

[28] A. D'Angelo, E. Menegatti, and E. Pagello, "How a cooperative behaviour can emerge from a robot team," *Proceedings of DARS'04*, 2004.

[29] M. S. Branicky, "Analysis of continuous switching system: Theory and examples," *Proceedings of the American Control Conference*, 1994.

[30] P. A. Bernstein, "Middleware. an architecture for distributed system services," DEC Cambridge Research Lab, Tech. Rep., 1993.

[31] A. Campbell, G. Coulson, and M. Kounavis, "Managing complexity: Middleware explained." *IT Professional*, pp. 22–28, September 1999.

[32] The Open Group, "Dce 1.1: Remote procedure call," The Open Group, MA, Tech. Rep., 1997.

[33] L. Burelli, "An integrated control system architecture for autonomous vehicles," in *IASTED Intelligent Systems and Control (ISC)*, 2007.

[34] A. Tellatin, *Sistema hardware e software di supervisione di un veicolo autonomo*. DEI-University of Padova, 2005.

[35] L. Luo, "Fault manifestation model for predicting anomalous system behavior," in *International Conference on Dependable Systems and Networks (DSN)*, Bethesda, MD, 2002.

Acknowledgements

Compiling a list of people that deserve being mentioned here is arguably the most overwhelming task I faced during this Thesis. I could go on and on writing about the people that helped me in one way or the other during all these years. **Thanks, guys**, for making this choice so difficult!

Above all, I would like to thank my advisor, prof. Ruggero Frezza, and his "twin" in spirit, prof. Alessandro Beghi, for having given me so many research projects which were also a lot of fun to work on. They put unquestioned faith in me and all the other lab students, and have given me unique opportunities to broaden my horizon with their knowledge and help. On this topic, Angelo Cenedese rightly deserves an honorable mention, for all the insights he tirelessly (at all times, really) shared with me while I was busy writing my articles or this Thesis.

However, I could not have done half of what I did without the help of my lab-mates. My biggest thanks go to my pal Andrea; a lot of things that we accomplished together would have been simply out of my reach without him. We shared lots of long hours in the lab, but had also lots of fun. I would like to also thank all the students who enrolled in the "Controllo dei Processi 2005" course for having done their fair share (and often more than that) to participate in the autonomous vehicle project.

Thanks also go to all my fellow Ph.D. students in the office at the 3rd floor, and especially to Maura, Paolo and Ruggero: we began this adventure together, and we are finishing together (well, except for Paolo... I guess there must always be one ahead of the pack!). I am sorry because we didn't share as much as I would have liked to: you guys are really a pleasure to stay with, and I hope we will keep in touch in the future. I will always remember my California trip in summer 2006; 7 people there from the same office... so many adventures!

Back to being serious for a minute because I want to say thanks to all the people I had the pleasure to work with during these years, starting with San Mateo: Mark Spence, Bob and all the guys at SES, for being so helpful and supportive. I would also like to thank the guys in Ducati Corse: Marco Amorosa, Roberto Canè, Luca Gasbarro, Riccardo Sancassani and Filippo Tosi, for all their support in our work, including the

astounding opportunity to go and work with them at the Mugello circuit.

Finally, among the guys that are working with me now in M31 there are some of the best friends I have made at the University (Al, Andrea, Arrigo, Enrico, Fabio), and I am proud to be both their friend and colleague. But my last thanks are for those that silently, and perhaps unknowingly, helped me continuosly during all these years: my friends in Friuli (Alex, Daniele, Davide, Federica, Tiziana. . .) and Veneto (Alberto, Ale, Andrea, Francesca, Stefano, Vincenzo. . .). My tirelessly supportive parents, to whom this Thesis is dedicated. Oh, and in case I forgot, you that are reading this.

My heartfelt thanks go to all of you, wonderful people, for having shared with me this intense period of my life.