Definition of a Cost–Effective, Fault–Tolerant Control Architecture: Application to the Design of a Steer–by–Wire System

by

Mathieu Pierre Bouvier

Diplôme de Gradué en Ingénierie, ENSTA, Paris, 2000

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(Department of Mechanical Engineering)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

December 11, 2002

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Mechanical Engineering

The University Of British Columbia
Vancouver, Canada

Date December 11, 2002

Abstract

The objective of this thesis is the definition of a cost–effective fault–tolerant architecture for use in safety–critical embedded control systems. Typical practical examples of such systems are the "By–Wire" systems (e.g. Steer–by–Wire, Throttle–by–Wire, etc) which will likely be applied on–board cars and pleasure boats in the not–too–distant future.

The novel architecture presented in this thesis performs error detection and error treatment at the sensor, actuator and Electronic Control Unit levels. It is based on the use of triple modular redundancy. A number of software utilities are defined, which interact with an object–oriented model of the physical system and provide redundancy management, multi–level error detection and dynamic software reconfiguration. A stateline architecture is also presented, which allows the system to dynamically isolate faulty nodes from the network and to perform the necessary hardware reconfiguration when a faulty ECU is detected. The methods developed concentrate upon the use of dynamic reconfiguration so as to ensure optimal use of the available resources and provide safe system operation in the presence of faulty components. The software architecture is coded in the ANS Forth programming language.

This architecture has been implemented on a laboratory prototype system which represents a marine Steer–by–Wire application. Details of the actual implementation and of the design of the prototype are provided.

Contents

List of Tables

List of Figures

List of Symbols and Acronyms

ω_{max}	Maximum allowable shaft rotational velocity [RPM]
$\Omega(s)$	Rotational velocity of the Drive Unit's DC motor shaft [rad.s^{-1}]
ω_n	Closed–loop natural frequency [rad.s^{-1}]
τ_{vel}	First order transfer function time constant [s]
θ_a	Analog angle (either Drive or Helm Unit) [V]
$\hat{\theta_a}$	Estimated analog angle (either Drive or Helm Unit) [V]
θ_d	Digital angle (either Drive or Helm Unit) [11–bit]
$\hat{\theta_d}$	Estimated digital angle (either Drive or Helm Unit) [11–bit]
θ_D	Drive Unit angle [rad]
$\theta_{D,a}$	Analog Drive Unit angle [V]
$\theta_{D,d}$	Digital Drive Unit angle [11–bit]
θ_H	Helm Unit angle [rad]
$\theta_{H,a}$	Analog Helm Unit angle [V]
$\theta_{H,d}$	Digital Helm Unit angle [11–bit]
θ_{nA}	Nominal angle [rad]
$\theta_{nA,d}$	Digital nominal angle [11–bit]
ζ	Closed–loop damping ratio
a	Lead/lag filter parameter in continuous time

A	Lead/lag filter parameter in discrete time
ABS	Antilock Braking System
ADC	Analog to Digital Converter
b	Lead/lag filter parameter in continuous time
B	Lead/lag filter parameter in discrete time
CAN	Controller Area Network
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DAC	Digital to Analog Converter
ECU	Electronic Control Unit
e_d	Digital error [11–bit]
EMI	Electro–Magnetic Interferences
ES	ECU state vector
ES_t	ECU state vector in the time domain
ESP	Electronic Stability Program
ES_v	ECU state vector in the value domain
$ES_{v,e}$	Estimated ECU state vector in the value domain
f_{enc}	Optical encoder decoding frequency [Hz]
GPS	Global Positioning System
$H(s)$	Global position control Laplace domain transfer function
$H_{ll}(s)$	Lead/lag filter Laplace domain transfer function
i_a	DC motor armature current $[A]$
J	Rotor inertia $[kg.m^2]$
K_A	Amplifier gain $[A.V^{-1}]$
$K_{A/D,\omega}$	Tach acquisition ADC gain (either Drive or Helm Unit) $[V^{-1}]$
$K_{A/D,\theta}$	Pot acquisition ADC gain (either Drive or Helm Unit) $[V^{-1}]$
$K_{A/D,D,\theta}$	Drive Unit pot acquisition ADC gain $[V^{-1}]$

$K_{A/D,H,\theta}$	Helm Unit pot acquisition ADC gain $[V^{-1}]$
$K_{D/A,D}$	Drive Unit DAC gain $[V]$
K_{ll}	Lead/lag filter gain
k_{loop}	Loop gain (adjustable drive amplifier velocity control)
K_{nA}	Nominal angle gain
$K_{pot,D}$	Drive Unit potentiometer gain $[V.rad^{-1}]$
$K_{pot,H}$	Helm Unit potentiometer gain $[V.rad^{-1}]$
k_{ref}	Reference gain (adjustable drive amplifier velocity control)
K_T	Torque constant $[Nm.A^{-1}]$
k_{tach}	Tachometer gain (adjustable drive amplifier velocity control)
$K_{tach,D}$	Physical Drive Unit tachometer gain $[V.(rad.s^{-1})^{-1}]$
$K_{tach,H}$	Physical Helm Unit tachometer gain $[V.(rad.s^{-1})^{-1}]$
K_{vel}	First–order transfer function gain
l_{enc}	Number of lines on the optical encoder's surface
LS	Link state vector
M_c	Agreed–on correct measurement
M_e	Estimated correct measurement
MS	Measurement state vector
PCMCIA	Personal Computer Memory Card International Association
PFA	Product Family Architecture
PWM	Pulse Width Modulated
R^*	Arbitrary coefficient $[\Omega]$
R_1	Resistance used for treatment of analog signals $[\Omega]$
R_2	Resistance used for treatment of analog signals $[\Omega]$
R_3	Resistance used for treatment of analog signals $[\Omega]$
R_4	Resistance used for treatment of analog signals $[\Omega]$
$RH5$	Adjustable internal Drive Unit servo amplifier resistance $[\Omega]$

$RH7$	Adjustable internal Drive Unit servo amplifier resistance [Ω]
$RH10$	Adjustable internal Drive Unit servo amplifier resistance [Ω]
RTOS	Real Time Operating System
SAE	Society of Automotive Engineers
SS	Sensor state vector
T	Torque [$N.m$]
T_D	Drive Unit torque [N.m]
TDMA	Time Division Multiple Access
T_H	Helm Unit torque [N.m]
T_{in}	Torque input at the Helm by the driver [N.m]
T_{load}	Load torque [N.m]
TMR	Triple Modular Redundancy
T_s	Sampling time [s]
TTA	Time Triggered Architecture
TTP	Time Triggered Protocol
v	Boat speed [$m.s^{-1}$]
$V_{D,a}$	Analog command signal to the Drive Unit actuator [V]
$V_{D,a,LT}$	Analog command signal to the Drive Unit actuator, supplied by the laptop [V]
$V_{D,a,TDS}$	Analog command signal to the Drive Unit actuator, supplied by TDS_2 [V]
$V_{D,d}$	Digital command signal to the Drive Unit actuator (output from the position controller) [12–bit on laptop, 9–bit on TDS]
V_{in}	Input voltage [V]
V_{out}	Output voltage [V]
V_{shift}	Shifting voltage [V]
V_{switch}	Digital switching signal; either low (0V) or high (5V)

XBW	X–By–Wire
XT	Execution Token

Acknowledgements

I would like to thank Dr Ian Yellowley, my supervisor, for providing me with invaluable advice and support during the completion of this work. His expertise and accessibility have made my experience here at UBC an extremely enriching and pleasant one.

I would also like to thank Kevin Oldknow, who has been my lab partner for the last two years, and who has always been willing to help whenever I needed it. His presence has contributed to make my work environment exciting and enjoyable.

I would also like to recognize the unconditional support of my parents, who have helped me morally and financially, and have always been of great advice despite the distance. Finally I would like to give a special thank you to Christine.

Chapter 1

Introduction

1.1 Tomorrow's personal vehicle

The future development of personal vehicles is dependent on progress made in the design of basic electronic networks and control systems. Many electronic systems are readily available today; from simple power window to ABS and traction control on cars, from autopilots to bow thrusters on boats.

The current trend in the automobile industry is to take this approach a step further, and replace vital (safety–critical) mechanical or hydraulic systems with electronic systems. These are commonly called "By–Wire" systems, refering to the "Fly–by–Wire" systems found on commercial airplanes since the late eighties, and include Brake–by–Wire, Steer–by–Wire and Throttle–by–Wire.

All of these By–Wire systems involve the measurement of driver input (position, force), and the use of an intelligent networked control system to provide the correct response through the actuators involved.

It should also be observed that in many cases the use of hydraulic actuators for braking and steering will become less and less attractive compared to electrical drive elements (controllability, weight and efficiency are the main issues).

The steering system, and in particular marine examples of steering systems, are used in this thesis to demonstrate the difficulties associated with the design of By–Wire systems, and to examine possible solutions. Clearly though the system level problems are similar in all cases.

1.1.1 Advantages of "By–Wire" systems

Many improvements are expected from the use of By–Wire systems. This section attempts to provide an exhaustive list of these potential improvements.

Performance improvement

One of the main advantages of By–Wire systems is the opportunity for performance improvement. Using software it is considerably easier to implement more flexible control strategies, which also allow for better "tunability" and performance of the control systems.

As an example, consider the classical approach to car steering that combines a mechanical linkage and hydraulic power steering. The effort required to maneuver a car using this system is inversely proportional to the vehicle's speed, and therefore turning the steering wheel for parking always requires more torque than high–speed direction adjustments. Improvements can be made to the hydraulic system but they are expensive and complex. With a Steer–by–Wire system, it is easy to incorporate the vehicle's speed into the control loop, and thereby adjust the tactile force feedback to guarantee effortless use to the user.

Many studies have aimed at developing improved strategies for car steering systems. Most have been for Electric Power Steering (EPS)[28, 47]. EPS is not Steer–by–Wire, since the mechanical linkage between the steering wheel and the front axle (i.e. the steering column) is not removed; it is simply electrically assisted steering. It represents however "the first step to Steer–by–Wire"[38] and is commercially available today.

Fuel efficiency

The traditional power steering on both boats and cars uses a hydraulic pump, which assists the driver in the steering task. However, with such systems, there

is permanent oil rotation, regardless of steering conditions. This results in increased fuel consumption. Peter and Gerhard[38] have calculated this to consume 0.3 to 0.4 liters per 100 km for a medium sized car. With an EPS system, and *a fortiori* Steer–by–Wire, the steering system requires energy only when an actual steering action is commanded, which according to [38] amounts for relative fuel savings of around 85%, the best values being obtained while driving on highways, where there is minimal steering action involved.

Throttle–by–Wire systems also give rise to significant decrease in fuel consumption, by making optimized throttle opening possible in real–time.

Integration in the vehicle–wide electronic network

Having electronic elements connected through a network allows for sharing of information, and thereby represents a way of increasing the overall vehicle performance. For example, a traction control system optimizes the grip of each tire by acting jointly on the braking and powertrain systems. It simultaneously modulates braking on the considered wheel, and also delays the transmission of torque to this wheel.

On a pleasure boat, an autopilot or GPS[1] can be much more easily coupled to a Steer–by–Wire system than to a classical mechanical or hydraulic steering system. The implementation of multiple helm stations is also greatly simplified.

It should be mentioned as well that a well designed By–Wire system will provide monitoring functionalities, hence giving easy access to self and/or coupled diagnosis *via* the vehicle network. This will ease and speed up the whole diagnosis/repair cycle, and therefore represents a possibility for cost reduction. Cost, though, is a major concern in the design of such systems, as detailed in the following section.

[1] Global Positioning System

Passive safety and design flexibility

All traditional design constraints due to the steering linkage itself are removed, giving the potential for innovative and safer designs.

For example, in a car the pedal set and steering wheel/steering column arrangement can be replaced by a "joystick" type user interface, thereby allowing complete freedom in the design of the car interior. The absence of steering column in particular makes significant passive safety improvements possible. In a pleasure boat it is possible to replace the steering wheel with a handheld "steering unit", thereby providing remote steering functionalities.

1.1.2 Challenges

Despite the envisioned improvements detailed in 1.1.1, the design of By–Wire systems is a difficult task. The challenge here is to guarantee fault–tolerance of such safety–critical systems, that is to guarantee that they perform safely even if one or more of their components fail. The latter property is essential for obvious reasons of personal safety, company liability and for public acceptance. A steering system failure will put the driver's life at risk in most cases. However, the technological solutions adopted to achieve fault–tolerance in existing By–Wire applications (typically Fly–by–Wire) rely on high levels of redundancy, and are therefore not directly applicable to the development of products in markets as cost–sensitive as the pleasure boat or automobile markets, where the costs of By–Wire systems should be comparable to that of the conventional systems that they replace.

1.2 Thesis objective

The objective of this thesis is to define a cost–effective control architecture for fault–tolerant, safety–critical embedded systems. This architecture should make

an optimal use of multi–level redundancy, and allow the system to tolerate faults at all levels of its organization.

1.3 Thesis outline

This thesis is organized as follows. In Chapter 2, a detailed description of pleasure boat steering systems and components is provided. This description includes traditional systems and future Steer–by–Wire systems, which are described from a hardware, controls and software point of view.

Chapter 3 addresses the subject of fault–tolerance. It first specifies the concepts involved and possible fault sources in typical By–Wire systems. It then provides a complete literature review of research efforts which have been conducted to design fault–tolerant safety–critical systems. A particular emphasis is put on the communication protocols used in such networked systems, and on the various available approaches to error detection and error treatment. Finally, the architecture which was developed as part of this thesis is detailed.

Chapter 4 describes the implementation of a laboratory marine Steer–by–Wire system, which was designed and built to illustrate the concepts described in both previous chapters.

Finally Chapter 5 lists a series of conclusions from the thesis and recommendations for future work.

Chapter 2

Boat steering systems and components

2.1 Introduction

As mentioned in Chapter 1, the application used as an example in this thesis is the steering system of a typical pleasure boat. The purpose of marine steering systems is to transmit the input from the helmsman at the helm station to the steering actuators, which move the rudder or engine[1] thereby steering the vessel. Traditionally this is achieved by using either a mechanical or hydraulic connection. Many additional systems are readily available to assist the helmsman in the steering task, such as power–steering and autopilot systems. However it is foreseen that the future development of marine steering systems will lie in the generalized use of Steer–by–Wire, where the entire steering action is electronically controlled, and all mechanical and hydraulic links are removed.

This chapter provides a detailed review of the existing traditional marine steering and power–steering systems, and of their integration with electronic components such as autopilots and GPS. It then attempts to describe the organization of a marine Steer–by–Wire system, by detailing the control strategies

[1] On inboard engine boats the steering is provided by the movement of rudders or of the propellers, while on outboard engine boats it is provided by the movement of the engine itself, to which the propeller is attached. In subsequent discussions, the steering "component" will be refered to as rudder.

that can be used, as well as providing a complete description of the components of such systems. Finally an approach to software development using object–oriented technology is introduced and justified.

2.2 Description of existing systems

This section examines the various types of steering systems employed in pleasure boats, their main characteristics and integration with the other vessel systems.

2.2.1 Steering systems

Mechanical steering

The most simple marine steering systems available are based on a direct mechanical connection between the helm and rudder. In such systems, the operator turns the steering wheel which itself turns a geared helm. The rotary movement of the helm is used to impose a linear motion on a push–pull flexible cable, which links the helm station to the steering unit. The motion of this cable in turn induces a movement of the rudder, thereby steering the boat. Two types of arrangements can be used to transform the rotary motion of the helm into linear motion of the cable: either rack and pinion (where the push–pull cable is attached to a rack gear, itself moved by a pinion) or rotary (where the cable wraps around a pulley)[50].

Fig 2.1 shows a rotary mechanical steering system. A slightly modified system is also shown, in which steering is provided by the motion of twin push–pull cables.

Such systems are particularly well adapted to smaller low–cost pleasure boats. They are durable, reliable and require little maintenance. The number of lock–to–lock turns of the steering wheel, which determines the effort required to turn the boat, is dependent on the helm gear used. When selecting

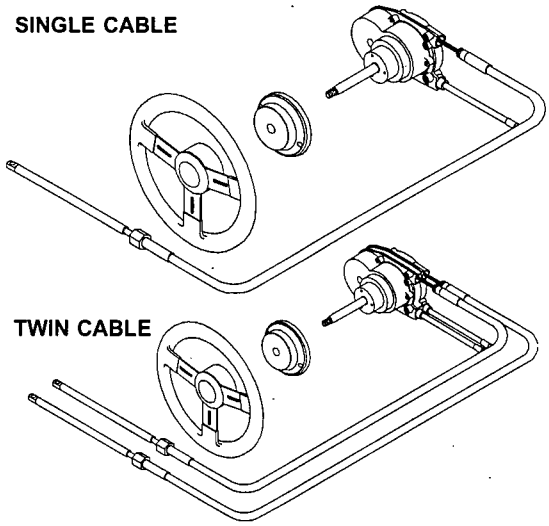

Figure 2.1: *Typical mechanical steering systems: single and twin cables (figure from [50])*

a mechanical steering system, the customer has thus to consider a trade–off between comfort and handling levels.

Hydraulic steering

The other commonly used type of marine steering systems is based on a hydraulic connection between the helm and rudder. In such systems, the helm and drive units are connected by hoses, into which a fluid (oil) is pumped by rotary movements of the helm. On the drive side the pumped fluid induces the movement of a cylinder, which causes the rudder to move.

A typical example of such a system is shown in Fig 2.2.

Hydraulic steering systems provide smoother and more comfortable operation. They are usually used in bigger pleasure boats, especially when combined with power–steering systems, as described in 2.2.2. The use of hydraulic fluid implies more frequent maintenance than mechanical systems.

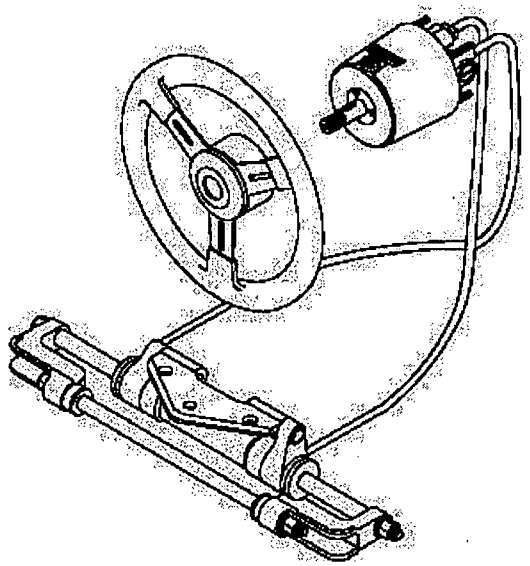

Figure 2.2: *Typical hydraulic steering system (figure from [50])*

It should be noted that both hydraulic and mechanical steering systems can be adapted to specific boat configurations, to steer twin rudders or outboard engines for instance. Generally speaking, hydraulic and mechanical steering systems constitute the basic subsystem on which more complex systems are developed, such as presented in 2.2.2 and 2.2.3.

Electrical steering

Purely electrical marine steering systems for pleasure boat applications (Steer–by–Wire) are not commercially available. However a very simple such system was developed, commercialized and sold for a short time in 1998[3]. This system has since been removed from the market, and the lack of available literature describing it makes it hard to discuss its features in detail. It is based on a single digital position closed–loop controller, which acquires current helm and rudder positions using a potentiometer at each of these units. The actuation

itself is provided by a DC motor, commanded by the controller, which drives a lead screw. The rotational displacement of this lead screw is converted into linear motion of a nut, which moves the rudder. The performance of the system was evaluated in boating tests in [3].

This system pioneered the use of marine Steer–by–Wire, but it is unfortunately overly simple. In particular it does not implement any fault–tolerance features of the controller. Furthermore it cannot be customized by the user and is not meant to be integrated into a boat–wide steering system which could include autopilot and GPS functions for instance. Therefore its level of functionality is very similar to that of a classic manual (non power–assisted) hydraulic or mechanical steering system.

Finally systems are available, which combine electrical and mechanical steering. Such systems make use of a modified, electrically operated helm. In effect, the linkage between the helm and the rudder is identical to that of a mechanical steering system in all points, but the user input to the system is provided through a remote hand–held control unit. This input is then processed by an open–loop controller, which actuates the electrically operated helm. It is also possible to switch the controller off and operate the helm manually. Such a system is only suitable for low speed boating, and is particularly interesting for situations where remote steering is required.

2.2.2 Power steering

In many cases the basic steering systems described in 2.2.1 are coupled with systems which assist the driver in the steering task. This allows the system to reach higher comfort levels, and is especially useful when used on larger yachts.

In a power steering system, the assist is provided by a hydraulic pump, which is driven by the engine or by an electric motor. This type of system is very similar to the hydraulic power steering systems found in cars. The primary

steering system is either mechanical or hydraulic. The steering cylinder, whose movement causes the rudder to move in both types of marine steering systems, is fitted with a servo cylinder and a power steering valve. When the cylinder moves, the valve opens and causes hydraulic fluid to be pumped into the system, thereby providing the desired assist. In case of failure of the hydraulic pump, the system automatically goes into back–up manual steering mode.

Fig 2.3 describes a typical hydraulic power steering system. In the example system shown, the hydraulic power steering system is used to drive twin rudders. Furthermore, the system shown is able to accomodate multiple helms.

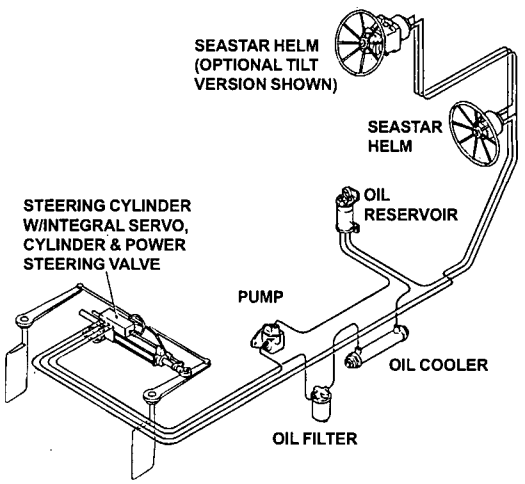

Figure 2.3: *Hydraulic power steering system (figure from [50])*

Note that more advanced power steering systems are also available. Such systems are based on a hydraulic steering system and use a pressure sensor to measure the torque applied at the helm, and a digital open–loop controller to command the valve opening of a hydraulic pump driven by the engine or by an electric motor. The improved performance of such systems compared to classical power steering resides in the possibility for the user to interactively tune the controller, so as to change the assist level.

2.2.3 Integration of boat systems

In this section boat systems which interact with the steering systems are described. The main such system under review is the autopilot. An autopilot is a system which allows the boat to follow a trajectory imposed in advance by the user. The way in which the boat is steered is similar to a hydraulic power steering system: an engine driven hydraulic pump is used to move a fluid and thereby induce movement of a steering cylinder. However, in an autopilot system, the pump is actuated following the instructions from a digital controller. It is generally possible to use the autopilot in three modes of operation:

- Compass mode: in this mode the user specifies the desired heading direction in degrees. A compass sensor and a position sensor at the rudder are used to provide feedback information to the controller, which can then apply a closed–loop control scheme and ensure that the rudder is moved according to the user reference heading.

- Jog mode: in this mode open–loop control is used. The user uses jog buttons to modify the direction in a "step–by–step" manner. The use of an autopilot in jog mode is very similar to that of the combined electrical and mechanical steering system described at the end of 2.2.1. Therefore it can be used for simple "light steering", or to correct the direction set for in compass mode, for instance for obstacle avoidance.

- GPS mode: this mode is similar to compass mode, except that the reference direction input to the closed–loop controller is updated in real–time according to a user specified path. A linear interpolation scheme determines the direction to follow at each instant according to the current absolute position as measured by a GPS, and to the desired path.

A typical autopilot system is shown in Fig 2.4.

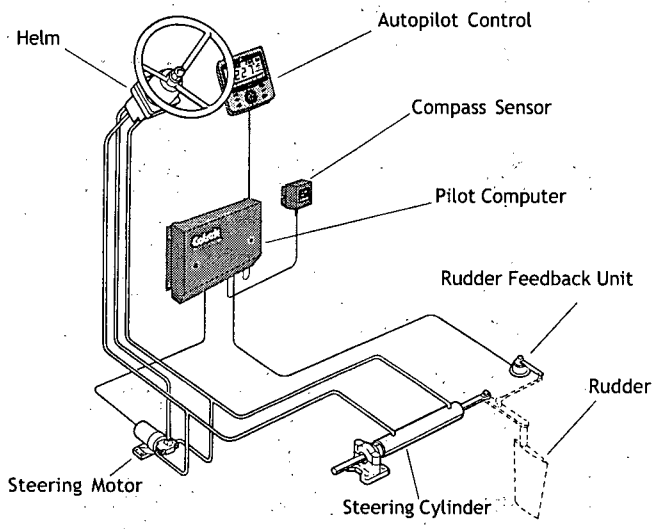

Figure 2.4: *Autopilot system and integration in the steering system (figure from [7])*

2.3 Steer–by–Wire

As discussed in Chapter 1, the major development expected in boat steering lies in the generalization of Steer–by–Wire systems. This section defines and describes such systems. It also justifies the use of object–oriented technology for the development of their software.

2.3.1 System description

By definition, a Steer–by–Wire system is a steering system in which the hydraulic or mechanical connection between the helm and rudder is replaced by an electronic control arrangement. Sensors are used to measure the helm input provided by the user (e.g. position, applied torque, etc), and to measure data at the rudder (e.g. rudder angle, applied load, etc). An electronic control platform is then used to process this information and send a command to actuators

(typically DC motor based drives) which perform the steering action.

Unlike any of the previously described existing marine steering systems, a Steer–by–Wire system can be entirely tuned to accomodate the user's preferences and changing boating conditions, since the whole system is based on "intelligent" controllers. As described in 1.1.1 the system functionalities can be expanded to virtually any level, because the application is performed in software.

In many cases it may be desirable to implement a tactile feedback feature where a torque controller commands an actuator which applies a torque to the helm, so as to provide the helmsman with a feel of the torque applied at the rudder. This is an important feature with respect to safety and comfort. It should be possible for the user to tune this controller and thus adjust the level of assist provided by the steering system. This level of assist as well as the ratio of displacements at the helm and at the drive actuator could also vary automatically with the speed of the vessel, in order to provide greater comfort at low speeds and during docking maneuvers, as well as better handling characteristics at higher speeds.

In future, more advanced systems, it is expected that various parameters such as vessel speed, pitch, yaw and roll will be monitored and integrated into a global real–time boat model, which the steering controllers will use to ensure that the boat stays within its safe operating conditions and to enhance performance and comfort.

A marine Steer–by–Wire system is schematically represented in Fig 2.5. The primary controller used for steering the boat is the drive controller, while the helm controller determines the feedback torque to be applied at the helm. Position and torque sensors are obviously required for these control actions, and rotational velocity sensors may also be useful in certain cases[2]. In the example shown, movement of the rudder is provided by a lead–screw driven by a DC motor.

[2]See 2.3.3

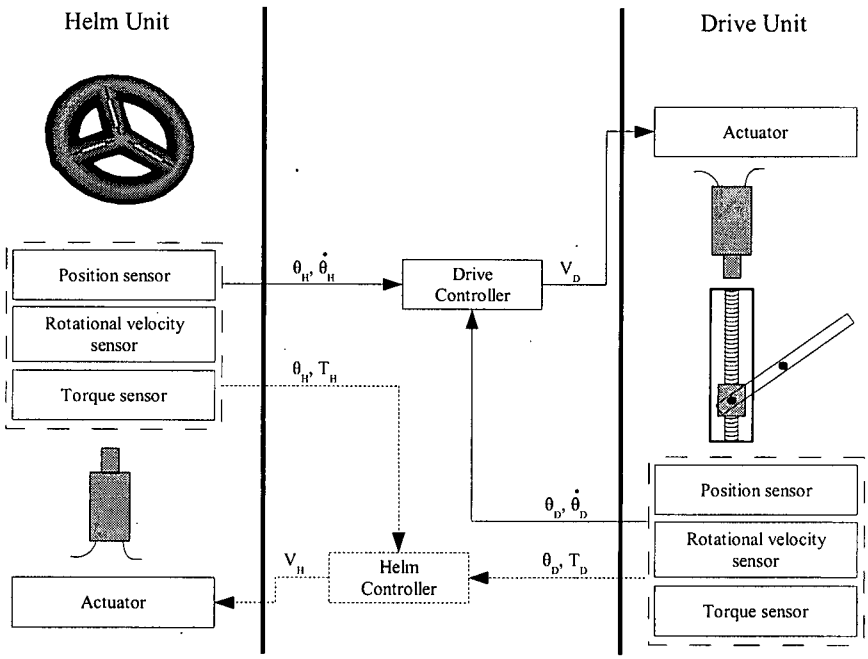

Figure 2.5: *Steer–by–Wire diagram*

2.3.2 Control model

This section describes marine Steer–by–Wire systems from a controls point of view. External inputs applied to the system are:

- the torque applied by the driver at the helm,
- the load torque (disturbance) applied by the environment at the drive. This torque is dependent on the boating conditions, e.g. speed of the vessel, rate of turn, etc.

Fig 2.6 portrays a typical Steer–by–Wire control diagram.

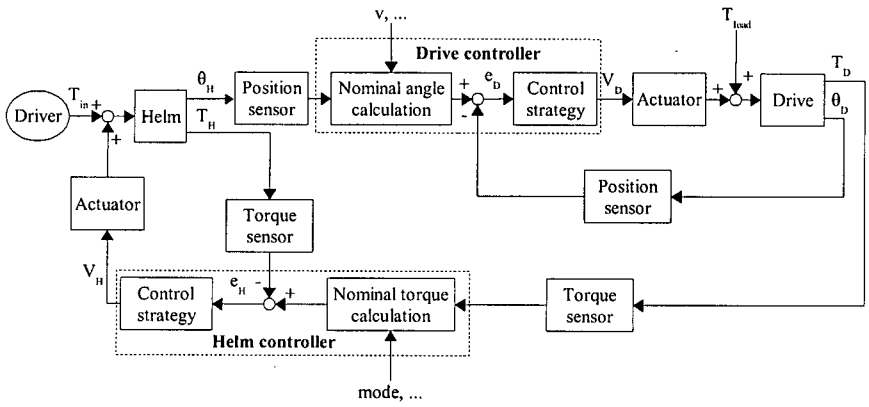

Figure 2.6: *Control diagram of a typical Steer–by–Wire system*

The symbols used in Fig 2.6 are defined in Table 2.1. In order to give as general a description as possible, digital to analog and analog to digital conversions are not shown in Fig 2.6. However, in most implementations the controllers will be implemented in digital form[3].

[3]In all subsequent sections of this thesis, analog and digital values will be indicated respectively by "a" and "d" subscripts (e.g. $V_{H,d}$: digital output of the helm controller and $V_{H,a}$: analog output from the helm DAC)

Symbol	Unit	Description
θ_H	rad	Helm angle
θ_D	rad	Drive angle
e_H		Helm (torque) error (analog or digital)
e_D		Drive (position) error (analog or digital)
T_H	N.m	Helm torque
T_D	N.m	Drive torque
T_{in}	N.m	Driver input torque
T_{load}	N.m	Load torque
v	$m.s^{-1}$	Boat speed
V_H		Helm actuating command (analog or digital)
V_D		Drive actuating command (analog or digital)

Table 2.1: Definitions of symbols used in Fig 2.6

As shown in Fig 2.6, the helm and drive controllers can be internally divided into two blocks:

- The **nominal angle calculation** and **nominal torque calculation** blocks are used by the drive and helm controllers respectively to determine the reference value input to the actuators. In its simplest form, the nominal angle calculation consists of multiplying the helm angle θ_H by a constant gain, which accounts for the ratio between the number of lock–to–lock turns at the drive actuator and at the helm. However, in more advanced implementations it is possible to make this gain dependent on external parameters such as the speed of the vessel. This allows modification of the relationship between the helm and drive angles, and thus adaptation of the handling characteristics of the boat to the boating conditions, resulting in an improvement of performance and comfort.

A similar approach can be used at the helm controller to calculate the nominal (reference) torque depending on user settings and on external parameters.

- The **control strategy** blocks apply the control algorithms, which take the computed errors as inputs, and output commands to the actuators. The degree of complexity of these algorithms is dependent on the desired level of performance of the steering system. In particular additional parameters such as the drive actuator and helm rotational velocities could be used in advanced control schemes. Furthermore, the *a priori* unknown load torque T_{load} could be estimated in real–time using a boat model which would take parameters such as the vessel speed into account. Such developments are outside the scope of this thesis.

2.3.3 System components

From the previous sections it is possible to identify the physical components used in the building of a Steer–by–Wire system. This section provides a detailed description of these components.

Sensors

Sensors are devices, which output an electrical signal that is related to a given physical parameter. One needs to separate *analog* sensors from their *digital* counterparts. An analog sensor outputs an analog signal (typically a voltage), which must be converted to digital form before it can be processed by an ECU[4]. On the other hand, the output of a digital sensor is carried by a number of parallel lines, which can only take on 0 or 1 logical values (i.e. typically 0V or +5V). The output logical values change in time, therefore they are usually counted by custom hardware and stored inside registers, which can then be

[4]Electronic Control Unit

accessed by the system's ECU's. [11] provides an extensive review of sensing devices for controls applications.

For a marine Steer–by–Wire system position sensing is required at both the helm and drive units for closed–loop position control. As detailed in the preceeding sections, additional measurements of motor torque and rotational velocity are useful for more sophisticated control strategies, and in cases where tactile feedback is implemented. Furthermore, rotational velocity measurements, associated with a software integration scheme, can be used to provide back–up position measurements.

For this type of position sensing purposes, a potentiometer is the most obvious candidate sensor. A potentiometer is a modified resistor, comprising three terminals. Two of these terminals are connected to the ends of the resistive element, while the third terminal (the "wiper") slides along the resistive element, making electrical contact at a single point. A potentiometer can be used as position sensor because the resistance between either fixed terminal and the wiper varies linearly with the displacement measured relative to a reference position. Therefore if a constant voltage is applied across the whole resistive element, the measured voltage across one of the fixed terminals and the wiper is a linear function of the position. Both linear and rotary potentiometers are available. A digital alternative to a potentiometer is an encoder (either rotary or linear). Incremental encoders output pulse signals on one or more lines. Counting the number of pulses received in a predetermined amount of time allows one to obtain the current position compared to a reference position[5]. It should be noted that the maximum displacement rate is limited by the counting (decoding) frequency. For an incremental optical rotary encoder in particular, the decoding frequency f_{dec} [Hz] is related to the maximum allowable shaft rotational velocity ω_{max} [RPM] in the following manner:

[5]Note that inversely, measuring the time that each pulse lasts allows one to obtain the velocity

$$\omega_{max} = \frac{15 \cdot f_{enc}}{l_{enc}} \tag{2.1}$$

where l_{enc} is the number of lines on the encoder's surface. Therefore for a typical maximum rotational velocity of 500RPM which can be encountered at the drive's lead screw, and a typical l_{enc} of 1000, it is required to decode the encoder's output at a minimum frequency of 34kHz. Absolute encoders on the other hand output a direct binary representation of the absolute position, and can be input to an ECU using a parallel port. The precision of the measurement of such devices is dependent on the number of parallel lines used.

The most common rotational velocity sensors are tachometers. A tachometer outputs a voltage directly proportional to the rotational velocity of the shaft to which it is attached. As shown above, shaft encoders can also be used as rotational velocity sensors.

Torque can be measured using strain gages and piezoelectric sensors, or by measuring the armature current from the DC motor which provides the unknown torque (or counters it in a servo control application)[11]. In this last case, the torque T $[N.m]$ is related to the armature current i_a $[A]$ in the following way:

$$T = K_T \cdot i_a \tag{2.2}$$

Note that it is possible to combine the sensing element itself with a dedicated microprocessor and memory at the sensor level e.g. for data conditioning purposes. Such systems are called "intelligent" sensing units, and can also contain a communication controller, which allows the sensing unit to send its data on a network.

Actuators

As described in the previous sections, actuators are required in a marine Steer–by–Wire system to provide movement of the rudder and tactile feedback at the

helm. Emphasis will be put on the primary system actuator, i.e. the drive actuator. A typical choice for such a component is a DC motor coupled to a lead screw/nut combination, which provides the required orientation of the rudder.

A "DC motor consists of an armature with conductors which rotates in a fixed magnetic field"[54]. The fixed magnetic field is created by field windings, wrapped around the stator's poles. Armature windings are wrapped around the armature (or rotor), which is attached to the motor shaft, inside the magnetic field. By applying a current through the armature windings, Laplace forces are created, which cause the rotor to rotate. The rotation is possible because the directions of the current at diametrically opposite points of the armature windings are opposite, thereby creating a torque, and inducing the rotation of the armature. Hence it appears that it is necessary to invert the direction of the armature current in a given armature winding when this winding crosses the neutral axis[6] if a constant rotational direction is desired. In a classical DC motor, this is achieved by using a pair of brushes, which maintain a sliding contact with the split ring and through which the armature voltage is applied[11].

Brushless DC motors on the other hand avoid the use of brushes and achieve commutation by using electronic switching of the stator winding current. In a brushless DC motor, the rotor is a permanent magnet, and therefore the magnetic field created is rotating. The stator is divided into a number of segments, and "commutation is accomplished by energizing the diametrically opposite segments sequentially, at time instants determined by the rotor position"[11]. In such systems as opposed to traditional DC motors for which the torque and rotational velocity are related to the armature current (at a fixed field current), these two parameters are dependent on the pulse rate which is used to command the commutations. As implied above, accurate switching requires that the current rotor position is accurately known. Therefore a position sensor (typically

[6]This is called commutation

shaft encoder or Hall effect sensor) is used by the electronic switching circuit to adapt the switching times to the current position.

ECU's and communication channels

In a Steer–by–Wire system, the control actions are performed by a distributed computing platform. It is comprised of a set of ECU's, which communicate over one or more communication channels. These components will not be described in detail here, since they are in Chapter 3.

2.3.4 Object–oriented software model

The preceeding sections have described the control organization and hardware components of Steer–by–Wire systems. In addition to these aspects, a central part of the organization of a Steer–by–Wire system resides in its software components. In this section the most important concepts of object–oriented programming are described. An initial analysis indicates that use of object–orientation is likely to be particularly well adapted to the software development of embedded systems such as Steer–by–Wire systems.

Object–Oriented programming

In the object–oriented approach to programming, complex systems are divided into smaller, encapsulated entities called **objects**. This approach is termed *abstraction*. Each object contains all variables that characterize it, as well as the algorithms necessary to act on them. These variables and algorithms are called **instance variables** and **methods** respectively. The division of the system into self–contained modules is termed *modularization*. Finally, the property of *encapsulation* (or information hiding) means that a given object does not have access to the internal organization of the other objects. Therefore exchanges between objects are carried out using well–defined interface methods[36].

Objects are not coded individually. They are *instantiated* from **class** definitions. The class definition of an object defines it completely, by specifying its instance variables and methods. Multiple objects (with different names) can be instantiated from the same class definition within a given object–oriented model.

A few important characteristics of object–orientation are detailed below:

- **Inheritence**: the property of inheritence allows new class definitions to "inherit" from existing class definitions. A "child" class created this way has access to all the instance variables and methods of its "parent" class. This property allows the source code to be greatly reduced, since the definition of a slightly different or enhanced class only requires that it is defined as the child of an already defined class, and that the additional instance variables and methods are defined.

- **Polymorphism**: this refers to the fact that the same name can be used for different methods belonging to different classes. This allows the programmer to considerably simplify the source code.

- **Aggregation**: this refers to the possibility for any instance variable of a newly created class to actually be an object instantiated from one of the existing class definitions.

Application to the software development of Steer–by–Wire systems

Object–oriented programming is well adapted to the software development of embedded systems in general, because of the tendency of physical entities (units) to conveniently map into software objects[10]. This represents a valuable aid in the development of the software application: for instance in the case of a Steer–by–Wire system, the helm and drive units can be modeled by `Helm` and `Drive` objects respectively. All characteristics of a given physical unit can then

be included in the corresponding class definition. In the case of the Steer–by–Wire systems described in this thesis, the software model resulting from this mapping is called the **Virtual Steering** model.

The fact that multiple objects can be instantiated from a single class is also particularly interesting for the design of a Steer–by–Wire system: for instance if multiple hardware helm stations are used, it is only necessary to define the `Helm` class once, and instantiate it twice, e.g. as objects `Helm1` and `Helm2`.

Finally, the property of encapsulation allows team development to be greatly simplified. The interface methods between objects need to be defined before the actual implementation is done. Once this interface definition has been completed, it is possible to allocate the coding work to different development teams, as the internal organization of each object is hidden to the other objects, and therefore of no importance to them. This property also allows different physical computing platforms (ECU's) to use identical high–level object–oriented models, which are interfaced to in an identical way from the programmer's standpoint, despite the low–level implementation differences. In Chapter 4 for instance, the object–oriented boat model developed for this thesis is implemented on a 16–bit embedded computer and on a 32–bit laptop.

2.4 Summary

This chapter provided a summary of pleasure boat steering systems. Classical mechanical and hydraulic steering, power steering and autopilot systems were described, thereby providing a broad overview of the systems currently available on the pleasure boat steering market. The organization of a typical marine Steer–by–Wire system was detailed from a controls standpoint, and typical hardware components of such a system were described. Finally it was shown that object–oriented technology is a well–adapted tool for the development and implementation of a Steer–by–Wire system's software components.

The description of the Steer–by–Wire systems provided in this chapter does not take into account the fault–tolerance requirement which was emphasized in 1.1.2. The following chapter details the concepts involved in fault–tolerance, and approaches to achieve it.

Chapter 3

Achieving Fault–Tolerance

3.1 Introduction

A fault–tolerant system is intended to provide continued functionality even if one or more of its components fail. Still no system, however well designed and tested, can guarantee a global catastrophic failure probability of 0. This is not specific to By–Wire systems: mechanical and hydraulic systems wear and break down from the effects of stresses and fatigue.

The goal for a By–Wire system designer should therefore be to create as robust and reliable a system as possible, i.e. to expand the range of faults tolerated by the system, so that the catastrophic failure probability is reduced as far as possible.

This chapter covers the topic of fault–tolerance. It first specifies the terminology used for the description of faults in the context of hard real–time distributed embedded systems. Common fault sources for typical By–Wire systems components are examined. Solutions for fault–tolerance of safety–critical systems are reviewed, including a description of suitable communication protocols, and various approaches to system design. Lastly an architecture is proposed for achieving fault–tolerance of distributed embedded systems in a cost–effective way.

3.2 Terminology

3.2.1 Fault model

This section details the terminology used to describe fault occurences, with fault–tolerance schemes in mind. It is essential to use a well defined, universal set of terms to accurately describe the concepts under review.

Hiller[21] provides a good summary of terms used in fault–tolerant system design. Although his work concentrates on software fault–tolerance, an identical terminology is applicable to the whole electro–mechanical system. The three most important terms to define and understand are **fault**, **error** and **failure**.

- A **fault** exists when the system is in a state which may lead to the non–respect of the specifications. Faults are called *transient* if they only exist for a limited time, as opposed to *permanent* faults. A distinction is also made between *dormant* and *active* faults. A dormant fault may never be activated, and the system may therefore obey all specifications under the encountered operating conditions despite the existence of a fault. For instance a position sensor may be mechanically damaged and therefore output correct data only within a limited position range. As long as this limited range includes the operating range the fault will remain dormant.

- An **error** is the manifestation of an active fault, and therefore an error exists when the considered component, subsystem or system deviates from its intented specifications. An error which has not been *detected* is called *latent*[30]. Errors can also be categorized according to their persistence in time: a *persistent* error is opposed to a *transient* one. The distinction between faults and errors is important. Indeed, although errors are caused by faults, only errors can be detected.

- A **failure** is the consequence of a non–treated error. Note that because of the hierarchical organization of a system, failure of a given subsystem will

be considered a fault from the higher level's perspective, and will possibly trigger an error, which may itself, if not treated, lead to a failure. This propagation of faults in the system is termed the "fundamental chain"[30] and illustrated in the following manner:

$$\ldots \longrightarrow \text{failure} \longrightarrow \text{fault} \longrightarrow \text{error} \longrightarrow \text{failure} \longrightarrow \text{fault} \longrightarrow \ldots$$

In a safety–critical system, a failure will be called *catastrophic* if it happens at a high enough level to put the user's safety at risk.

- The goal of fault–tolerance is to avoid high level system failures. Once a fault has been activated, thereby triggering an error, the first important step in this process is **error detection**. Error detection encompasses the various mechanisms, which allow the system to become aware of the occurence of an error. Depending on the type of error being monitored, the error detection process can take different forms, but the main categories of techniques are normally classified into *data replication* and *executable assertions*.

  Data replication techniques imply the existence of *redundancy*, i.e. of distinct hard or soft components being able to perform a given task. The outputs from these components can then be compared, and errors are detected if these outputs are determined to be inconsistent with each other. Redundancy can also be used to detect software transient errors, by performing computations twice and comparing the results obtained in each case. This is further detailed in 3.5.1. Note that data replication techniques are completely independent of the process being monitored.

  On the other hand, executable assertions require knowledge of the process. Indeed, they consist of checking a given output against a known, predefined system model. A position sensor for instance, will have a known

maximum travel range, and therefore any value outside of this range indicates an error. This is of course an extremely simple example, but executable assertions methods can be made as complicated and reliable as desired.

- The final step to fault–tolerance is **error treatment**. The goal of error treatment is to respond to the detected error, in a manner that allows the global system to stay in a correct operation state. Error detection and error treatment can be tightly coupled, for instance when TMR[1] is associated with a voting mechanism. This will not be discussed in detail at this point, as various approaches to this problem will be presented and discussed in the following sections.

The above definitions are summarized in fig 3.1.

3.2.2 Hard real–time requirements in distributed systems

Kopetz defines a **real–time** system as follows:

> A real–time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical time when the results are produced.[26]

In addition to this, a *hard* real–time system – as opposed to *soft* real–time system – is one for which a missed deadline implies a failure. In general, real–time computer systems interact with a physical environment, such as sensors and actuators. Safety–critical systems such as Steer–by–Wire are typical examples of hard real–time systems: a missed execution deadline is not tolerable since

[1] Triple Modular Redundancy

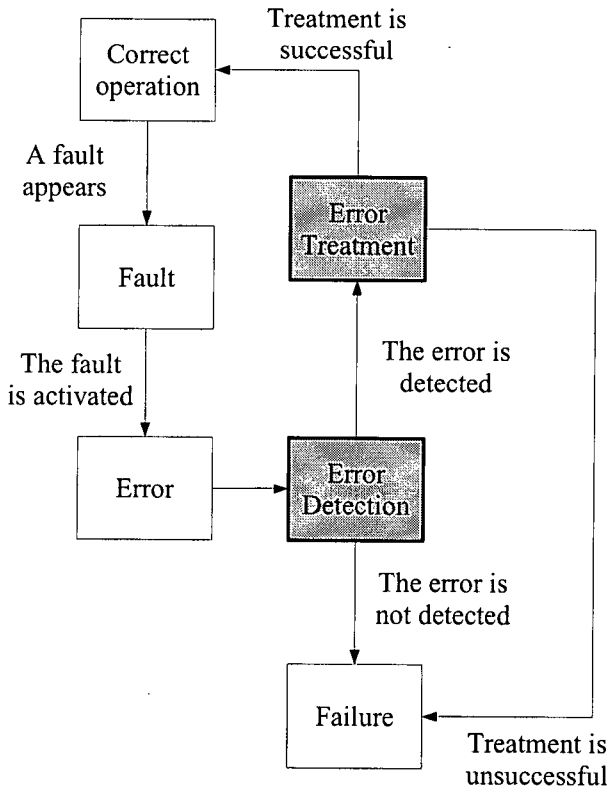

Figure 3.1: *Error detection and treatment.*

it may lead to global catastrophic failure. The manifestation of an active *time fault* will be called a *time error* and defined as a computation result which is:

- either never produced
- or produced outside its allowed predefined time zone.

A **distributed** system is comprised of several processors, which work together or in parallel to achieve a set of tasks. They are typically connected by a communication network over which they exchange information. Each computer connected to the network is called a **node**. Embedded fault–tolerant By–Wire systems are distributed by nature (it is a pre–condition for fault–tolerance: see 3.5.1).

In the types of systems of interest here, the distributed nature of the computer system is to be considered jointly with its hard real–time nature. Indeed, both the local computation itself and the communication process can induce time delays, and hence missed deadlines. In order to fulfill the distributed hard real–time requirements, it is therefore necessary to:

- have an accurate knowledge of the worst–case execution time of all software components. This pre–supposes that the execution time for a given piece of code is not subject to variations, or at least that the possible worst–case variations are known and acceptable from the points of view of performance and safety.
- be able to rely on a *deterministic* communication protocol, i.e. which exhibits a known, fixed message transmission time. This transmission time in itself, though an important factor with regard to performance, is not as vital in a hard real–time system as its accurate knowledge *a priori* and known, bounded variations.

An important definition is therefore that of **jitter**:

Jitter is the difference between the maximum and the minimum duration of an action (processing action, communication action)[26].

3.3 Fault sources

In this section the concepts introduced earlier are applied to the components of a typical By–Wire system identified in 2.3.3. It is necessary to have a good understanding of the nature of the potential faults for each of these components.

The following categories will be examined separately: sensors and actuators, ECU's and communication channels.

Sensors and actuators

In general, faults that can be expected in sensors are either due to mechanical failure of some components (e.g. from the effects of fatigue or accidental impact) or to environmental disturbances (e.g. vibrations, EMI[2], etc). Mechanical failures will usually lead to permanent faults, while electrical disturbances or vibrations may cause transient faults. "Intelligent" sensing units[3] should obviously be considered separately, since they feature logic components, which are themselves subject to EMI and other environment–caused disturbances as described in the following subsection. Although the use of intelligent sensing units simplifies the system from a global perspective, it adds a degree of complexity at the local level and therefore introduces additional fault sources in the system.

Common DC motor failures are due to wear of the brushes. The brushes tend to wear quickly because of their friction on the commutator ring. For applications where system reliability and longevity are important requirements, it is therefore preferable to use brushless DC motors. Other typical DC motor failure sources include bearings or seal failure, insulation breakdown, demagnetization

[2]Electro Magnetic Interferences

[3]As described in 2.3.3

and damaged connections.

ECU's

When examining faults on ECU's, it is necessary to distinguish physical hardware failures from software faults. Indeed, although an ECU can be considered "just another physical component" in the system, it also runs a soft application, which is itself subject to faults.

Apart from accidental physical damage to the ECU hardware, which is obviously out of the scope of normal operation, ECU hardware faults are typically caused by EMI, temperature changes or vibrations[37] and are therefore transient by nature. Such faults may corrupt the values present in memory or inside the processor's registers, and may lead to failure of the ECU if no mechanisms are implemented to detect and tolerate them. They can lead to software errors in both the time and value domains.

Software *design* faults ("*bugs*") should also be considered. Design faults exist "when the design of the system does not match the specifications"[37]. Therefore, they are not locally detectable (since the software performs according to its own – erroneous – specifications). However, especially in complex software systems, guaranteeing the absence of design faults implies that exponential numbers of combinations need to be considered and tested, which is often impossible. Ways to detect and treat software design errors are therefore required. This is usually achieved through the use of diverse redundancy[4].

In the context of distributed computing systems, malicious processor behavior – i.e. a fault which results in a processor outputing inconsistent and incorrect data at correct times – is called a "**Byzantine**" error. This refers to the classic Byzantine generals problem[29].

[4]See 3.5.1

Communication channels

A communication channel is faulty if the data it carries is corrupted during transmission or not transmitted. Again the most likely cause for corruption of the signal is electro–magnetic noise (EMI). Note that most communication protocols have built–in checks (typically CRC[5]), which allow them to detect if a message has been corrupted during its transmission. These faults are by definition transient, but they can cause higher level time faults, since the recovery mechanism may introduce non–deterministic transmission times[6].

Other types of faults can be caused by failure of the communication controller chips themselves, e.g. again from the effects of EMI or aging. Lastly damage to the physical medium itself (broken wire) will induce a permanent communication fault.

In a computing network, a faulty node, which attempts to gain access to the bus repetitively and thereby prevents the remaining fault–free nodes to communicate, is said to exhibit a **babbling idiot** behavior. Such behavior must absolutely be avoided in hard real–time, safety–critical applications, for obvious reasons.

Others

Sensor measurements acquired by ECU's can be incorrect because the – analog or digital – link from the considered sensor to the considered ECU is physically damaged. Possible damages can include broken wires or damaged contacts. In a similar manner it is possible that a correct analog command is sent in an incorrect manner to a non–faulty actuator because of a damaged link, thereby triggering an actuation error.

[5] Cyclic Redundancy Check
[6] See 3.2.2

3.4 Communication protocols in safety–critical applications

Developing a communication protocol which is suitable for hard real–time, fault–tolerant safety–critical applications has been the subject of much research effort. The communication protocol is a central component of any distributed system, as it carries information between the nodes of the network. The SAE[7] has published general requirements, which classify the various vehicle communication protocols. Class A protocols are adapted to low–speed, non real–time applications such as centralized power window or power lock systems. Class B protocols fulfill requirements for high–speed, real–time non safety–critical applications such as ABS or ESP[2]. Lastly Class C protocols have the highest performance, predictability and reliability requirements[43], as they are typically intended for use in future safety–critical By–Wire systems. They should also support fault–tolerance.

This section describes CAN[8], which is the most widely used communication protocol in current vehicle applications, as well as newer protocols and approaches which have been developed specifically for Class C applications.

3.4.1 CAN

CAN was developed by Robert Bosch GmBH from 1983 on, and introduced to the SAE in 1986. It was first used by Mercedes–Benz in production cars in 1992, and has since become the *de facto* standard for on–board car communication. The current CAN specifications[41] were published by Bosch in 1991 and since implemented by a variety of chip manufacturers.

[7] Society of Automotive Engineers
[8] Controller Area Network

The CAN protocol

CAN is a serial, multimaster protocol, which achieves maximal use of the bus while providing powerful priority, error detection and error confinement mechanisms. The basic concept used is that of non–destructive bitwise arbitration[41]. Each node can attempt transmission of a message at any time (provided that the bus is available). Messages do not contain systematic information regarding the sending or target node. Instead, each message is characterized by an identifier (29–bit field under 2.0 B specifications[41]), which holds a double role:

- Mark the message, e.g. so that other nodes know the type of information it contains. The receiving nodes can also accept only certain messages, based on their identifiers.
- Prioritize messages. This is an essential feature of CAN. Messages with the lower identifier value have the higher priority. In case two or more nodes attempt transmission of a message at the same time, thereby creating a conflict for access to the bus, the non–destructive bitwise arbitration scheme ensures that the highest priority message is transmitted with no data or time loss. Transmission of the lower priority messages will be attempted again when the bus becomes available.

This guarantees that all messages are eventually transmitted and maximizes the use of the bus on the basis of need. Therefore, CAN is particularly well suited to asynchronous, *event–triggered* communications, i.e. which happen at *a priori* unknown times.

Various error detection mechanisms are implemented within the basic CAN protocol. They include bit checking (a transmitter node checks that the bus level corresponds to what is being transmitted[9]), bit stuffing (every sequence of 5 bits of identical level is followed by a complement bit), and CRC[10]. If an

[9]Except in the arbitration and ACK–SLOT fields

[10]Cyclic Redundancy Check

error is detected, it is signalled to the other nodes through the transmission of an error flag.

Lastly, error confinement is implemented thanks to the use of transmit and receive error counts. Each detected error results in the increment of the appropriate error count by a given number, and the corresponding node is successively put into *error active*, *error passive* and *bus off* state, when predefined thresholds are overcome by the error counts. This allows the system to tolerate transient transmission faults (due to EMI for example) and to provide fail–silence of nodes which exhibit an abnormally high fault occurence rate.

Physical medium

Several physical media types can be used for implementation of the bus itself, from single wire to twisted pair or even optic fibre cable. A common solution is the use of a twisted pair. Under these conditions, according to the ISO 11898 application layer specifications[40], operation is maintained, even if:

- either wire is broken,
- either wire is shorted to ground,
- either wire is shorted to power[5].

This provides enhanced reliability of the CAN bus.

Performances

The transmission rate is fully selectable, and can be set to a maximum of 1 Mbps. Furthermore, each message frame can contain a maximum of 8 bytes of data.

Conclusions

CAN is particularly well adapted to Class B applications, but use of the arbitration mechanism implies possibility of a non-deterministic message transmission time: if two nodes request simultaneous access to the bus, the transmission time for the lower priority message will be greatly increased. The bare protocol is therefore not suitable for the tight hard real-time requirements of By-Wire systems.

3.4.2 TTP

Research led by Professor Kopetz at the Technical University of Vienna has resulted in the development of the TTP/C[11] protocol[27]. In order to avoid the non-deterministic behavior shown to be associated with event-triggered protocols in 3.4.1, TTP is based on *time-triggered* communications: each node is granted one or more time slots in the TDMA[12] round, during which it has exclusive access to the bus. This results in the elimination of all conflicts. The existence of a global time reference is necessary and obtained through tight synchronization of the nodes' local clocks. Furthermore, all communications have to be scheduled in advance for the given application.

TTP also supports fault-tolerance: there are two replicated communication channels, to which each node transmits its messages simultaneously. Management of this redundancy is an integral part of the protocol. Error detection is also integrated in the protocol: since all transmissions are scheduled in advance, any missing transmission is interpreted by the receiving node as an error from the sending node. Furthermore, data errors are detected in a similar manner as in CAN, using CRC algorithms.

TTP itself is meant to be integrated into a global system architecture. This integration has been the subject of an EU-funded project: "Time-Triggered Ar-

[11]Time Triggered Protocol for Class C applications

[12]Time Division Multiple Access

chitecture (TTA)"[13]. TTA combined the development of a fault–tolerant communication architecture using TTP/C with approaches to fault–tolerant sensor and actuator, and system software architectures. In particular it was concerned with the physical design of TTP/C controllers[19]. The global choices made in TTA to provide fault–tolerance are similar to those adopted by the X–by–Wire projects, and are described in detail in 3.5.3.

TTP/C has a maximum selectable transmission speed of 25 Mbps, as well as a maximum data size of 240 bytes within a message frame[31]. These two parameters represent a significant performance improvement from CAN[14].

3.4.3 Other approaches

Other groups have attempted to develop communication protocols adapted to SAE Class C applications. The following subsections discuss the most important alternatives to TTP/C.

TTCAN

It is desired by many users to use CAN as a base protocol, and to develop facilities for fault–tolerance in safety–critical applications on top of it[16]. This approach allows designers to use a familiar, widely tested and approved protocol for communications (CAN), while ensuring that Class C requirements are met.

Although CAN is usually not considered for safety–critical applications because of its event–triggered nature, it is important to understand that use of its non–deterministic arbitration mechanism is not necessary. Indeed the application can be scheduled in such a way that there are no conflicts for access to the bus, i.e. in a time–triggered manner.

The main effort toward the extension of CAN has been the development of

[13]ESPRIT OMI Project # 23396

[14]See 3.4.1

TTCAN[15] by Robert Bosch GmBH. In the TTCAN organization, each TDMA round is divided into so-called "time windows"[17]. *Exclusive* time windows are used for sending scheduled, periodic messages, in a similar way to the TTP approach. *Arbitrating* time windows on the other hand are used to send spontaneous, event–triggered messages which result from unpredictable asynchronous events. In an arbitrating time window, the bus behaves in a similar manner as simple CAN does, using bitwise arbitration for bus access. This organization allows both time–triggered and event–triggered transmissions to coexist within a single TDMA cycle.

For the scheduling of time–triggered transmissions to be effective, a global time is required. In the TTCAN organization, one of the nodes holds the role of time master, and periodically sends a synchronizing pulse to all other nodes. Fault–tolerance of this feature is implemented thanks to the use of other nodes, called potential time masters, which can take over in the event of a failure of the main time master[17, 18].

Müller *et al.*[33] describe ways to integrate TTCAN within a redundant bus architecture, thereby providing fault–tolerance features to the communication channels themselves.

Unfortunately TTCAN's performance is limited by CAN's performances [31], and they do therefore not reach the level set by TTP/C. In a TTCAN system as in a CAN system, data can be of a maximum size of 8 bytes within a message frame, and the highest transmission rate is inherently limited to 1 Mbps.

FlexRay

Another recently developed protocol is FlexRay. It is a hybrid protocol, much like TTCAN, providing time slots within a single TDMA round for both time–triggered and prioritized event–triggered communications[25]. It also supports redundant communication channels[31] and is intended to be more flexible than

[15] Time Triggered CAN

TTP/C. Unfortunately, FlexRay is developed by a consortium, which has so far made very little literature publicly available. It is therefore still hard to evaluate the protocol's functionalities.

3.5 Various approaches to fault–tolerance. A literature review

In this section different approaches taken to design fault–tolerant safety–critical systems are examined and contrasted. Although the projects reviewed rely on different methodologies, one concept is universal and compulsory in fault–tolerant system design: **redundancy**. Therefore the survey starts with a consideration of the need for redundancy.

3.5.1 Necessity of redundancy

Redundancy is a central concept in any fault–tolerant system. It seems obvious that if failure of some of the components of an embedded system is to be tolerated, the availability of back–up solutions is required. This indicates that redundancy is necessary for the error treatment phase. It is actually needed in most cases for error detection as well, as discussed in 3.2.1. Although executable assertions examine the correctness of individual components' outputs independently of the rest of the system, all other error detection techniques are based on the existence of redundancy at the monitored level of the system.

Hot redundancy is opposed to *cold* redundancy. A cold redundant component is not running in normal operation, but can be activated and take over if the primary component fails. A hot redundant unit or component on the other hand runs in parallel with its primary counterpart. Hot redundancy is advisable in hard real–time safety–critical environments, where the switching time must meet stringent time requirements. Furthermore, only hot redundant units can

be used for error detection purposes. Redundancy will therefore refer to hot redundancy in the rest of this chapter, unless otherwise specified.

One must also distinguish *exact* redundancy from *diverse* redundancy [37]. Exactly redundant units are identical in all points, perform the same tasks at the same times, and are therefore supposed to produce identical results at a given time. This makes bit by bit comparison usable, since any difference in the outputs implies the existence of a fault[16].

On the other hand, diversely redundant units use separate methods to perform a given task. Examples of diversely redundant units could be a potentiometer and an optical encoder for position sensing, or ECU's using different calculation methods. Because the means used to achieve the given task are not identical on all units, comparison of the various outputs must be realized within an acceptable, application specific region[17]. Diverse units will be declared in agreement if their outputs differ by less than an arbitrary percentage or absolute offset. Obviously, the choice of this offset implies a trade–off between the possibility of missing value errors and the one of detecting errors which do not actually exist.

The use of diversity allows a system to tolerate software design faults, which are not detectable using exact redundancy. It also decreases the probability of environment–caused simultaneous failures of the redundant units, by making the selection of physically different units possible at all levels. Finally, it aids in cost reduction, since high–quality, expensive units can be used in conjunction with lower performance redundant units. The main drawback of diverse redundancy lies in the application specific nature of the method, which implies that added software development is required. A method to achieve this in a systematic, efficient way is proposed in 4.4.

[16]This is only true for digital devices, since even identical analog devices output non exactly equal signals, and analog to digital conversion introduces a conversion error

[17]Note that this is also true for exactly redundant analog units

Another term, which is important to define, is that of **Triple Modular Redundancy** (TMR). TMR refers to the use of three redundant units, usually coupled with a voter, in the error detection process[37]. TMR does not necessarily imply diverse redundancy, but these two concepts are often used together.

3.5.2 Aeronautics applications

As mentioned in 1.1, Fly–By–Wire systems are commonly used in commercial aerospace applications, and have been in use since the late eighties. The first commercial airplane to be equipped with a total Fly–By–Wire system, with no mechanical back–up, was the Airbus A320, in 1988. Since then, Boeing has developed its own Fly–By–Wire aircraft (the 777), and Airbus has expanded its technology to newer models, such as the A330 and A340. Although the availability of the aeronautics example provides a good starting point for personal vehicle By–Wire systems design, it is obvious that the technological solutions adopted in Fly–By–Wire applications are not directly transferable to large scale, cost–sensitive markets.

Implementation of fault–tolerance relies for instance on triple–triple redundancy for the Boeing 777 Primary Flight Computers (PFCs)[60, 61]. This means that TMR is used at two levels in the PFCs: each of the three redundant computing units which form the PFC is itself comprised of 3 diversely redundant computers. Diversity is ensured at the hardware level by selecting hardware processors from different manufacturers (namely AMD, Intel and Motorola), and at the software level by compiling the Ada source code using three different Ada compilers[60].

3.5.3 "X–by–Wire" project

The "X–By–Wire[18]: Safety Related Fault Tolerant Systems in Vehicles" project [59] was conducted in Europe from 1996 to 1998 (funded by the European Union as part of the Brite EuRam III programme[19]). The project involved members of both the automobile industry and academia: Daimler–Benz, Centro Ricerche Fiat (CRF), Ford Europe, Volvo, Magneti Marelli, Bosch, Mecel, Technical University of Vienna, and Chalmers University of Technology.

The goal of the project was:

> "to achieve a framework for the introduction of ultra dependable electronic systems in vehicles which do not rely on conventional physical backups"[58].

The work was illustrated by the design of an example automobile steer–by–wire prototype.

Hardware

The approach taken by the XBW team is based on **exact redundancy** and **fail–silence**, at all levels in the system's organization. A Fail–Silent Unit[20] is one that outputs:

- either a correct value
- or nothing at all.

Under the XBW model, each atomic subsystem (Fault–Tolerant Unit[21]) is composed of 2 exactly redundant FSUs. This is valid at the sensor level as well as at the ECU level. Fig. 3.2 illustrates this concept in the case of ECU's.

[18] XBW
[19] Project number: BE 95/1329
[20] FSU
[21] FTU

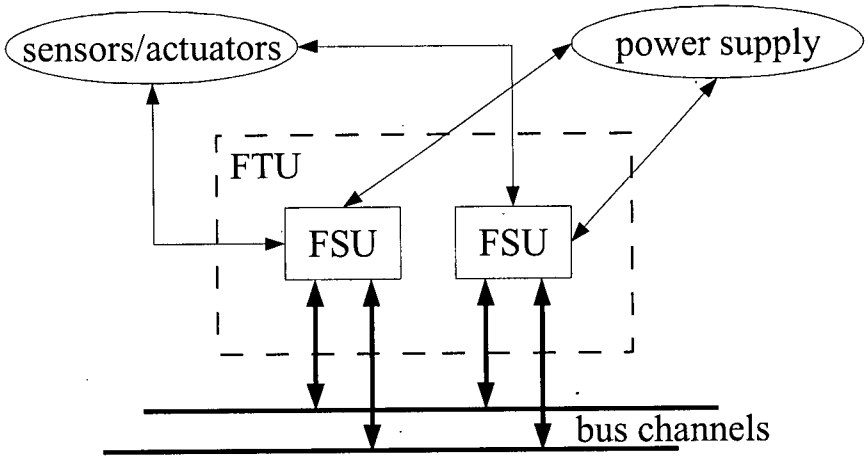

Figure 3.2: *X–by–Wire organization: Fail Silent Units and Fault Tole-rant Units.*

The obvious advantage of this method is the ensuing simplicity of the system: each unit being fault–tolerant, error detection is made easy from a global system's point of view, and coupling between the different subsystems is minimized.

However, achieving the fail–silence property is particularly challenging and cost–inefficient. At the ECU level, fail–silence implies that each FSU is able to perform self error detection, and is able to shut itself down. In practice, only the combination of two ECU's can make this possible. Therefore, quadruple redundancy is required to form a single Electronic Controller FTU.

In the same manner, a fail–silent sensor is comprised of 2 exactly redundant sensors, and a comparor, and therefore a sensor FTU necessitates 4 redundant sensing components.

Fault–tolerant actuating units are also implemented, here again using fail–silence and exact redundancy concepts. In particular, in the steer–by–wire prototype implementation, the torque is provided by 3 redundant DC motors, *via*

the use of a special gear box. Fail–silence is ensured by the choice of non–blocking motors, coupled with monitoring from the corresponding fault–tolerant computing units. Each of these motors provides one third of the torque in normal operation, and half of the torque in case of failure of one of them. This triple redundancy solution (where each motor needs to be able to provide 50% of the necessary torque) actually proves to be more cost–efficient than a double redundancy solution (where each motor needs to be able to provide 100% of the necessary torque)[59].

Communication

The XBW project was conducted in parallel and in close partnership with the TTA project[22]. Therefore the communication protocol used was TTP/C. As discussed in 3.4.2, TTP/C inherently contains exact redundancy, since it supports the use of a duplicated physical communication channel.

In the X–by–Wire implementation, the constitutive FSU's of a given FTU are assigned different time slots for transmission of their data. This factor associated with the fail–silence property of the FSU's ensures that each FTU transmits a correct message frame even if one of its FSU's is faulty.

Software

The fail–silence property is implemented in the XBW software model as well[6, 8]. The software code is conceptually divided into Basic Processing Elements[23]. A BPE is "the smallest composable software structure element"[8]. For each BPE, techniques such as double execution (a given input is processed twice and the outputs are compared), double execution with reference check (same, and treatment of a predefined input is also performed) or message validity check (coding of information) are used to detect transient errors. In the case of double

[22] See 3.4.2
[23] BPE

execution for instance, an internal comparer is then used to ensure fail-silence.

Timing errors can be detected *via* the use of a watchdog timer (external hardware). For this to be efficient, accurate worst-case computation time needs to be known for each BPE[24].

Finally, the exact hardware replication described earlier is used with an external comparer to detect permanent errors. If the outputs of the redundant processors differ, the comparer switches the whole FSU's output off, so as to provide a fail-silent behavior.

Conclusion

Despite the advantages of the method (namely simplicity and absence of cross-level coupling), hardware costs make it poorly suited to large scale production in cost-sensitive markets. Furthermore, the architecture is extremely rigid and does not make an optimal use of the available resources.

However, some of the techniques used for software error detection and for fault-tolerant actuating are particularly interesting.

3.5.4 Error Detection

Error detection is an essential step towards fault-tolerance. 3.3 exposes the levels at which errors can happen, and thus at which error detection is required. The previous sections have shown how redundancy can be used to detect errors, at any level in the system (whether sensor, actuator, or ECU level). While redundancy is required, many projects have aimed at avoiding the use of total hardware redundancy in an effort to optimize costs.

At the sensor or actuator levels, model-based methods have been developed, which exploit the "inherent redundancy contained in the dynamic system equations that relate the different sensors outputs"[15]. This is called *analytical*

[24] See 3.2.2

redundancy. The development of advanced system models makes estimation of the normal expected states of the system possible, and therefore provides a software–level pseudo hardware redundancy. [15] gives the example of a triple redundant unit being replaced by a double redundant hardware system used jointly with analytical redundancy, where significant cost decrease is achieved.

Most efforts towards error detection found in the literature concentrate on the ECU level of the system. Some of the software methods used inside the X–By–Wire organization to detect errors at the local fail–silent node level were introduced in 3.5.3. It has been attempted to design processor architectures, which inherently hold on–line self–testing capabilities[39]. In this example, a primary and back–up microprocessors are associated. Other approaches use codes to detect hardware ECU failures[48, 49]. In these applications, computation results are coded using a code generator, and later checked by a code checker, thereby providing self–test capabilities. This concept is similar to methods used for communication error detection, such as CRC. Unfortunately, it requires development of additional logic circuits, and can therefore not be directly used with off–the–shelf components.

Distributed ECU error detection is the subject of many research efforts as well. These works attempt to provide means for a distributed computing platform (network) to detect faulty nodes within itself. This implies that:

- each node is able to locally determine its opinion on the other nodes' "health" (i.e. to produce a local diagnosis of the global system state),
- and that a distributed mechanism exists, which combines all local diagnoses to produce a global assessment of the state of the nodes.

The most challenging task in this regard is the detection of Byzantine errors[12][25]. It has been shown that a number $N > 3t$ of nodes is required to tolerate t faulty

[25] See 3.3

processors with Byzantine behaviour[53]. This is usually not feasible in cost–constrained applications, and therefore illustrates the necessity to ensure that Byzantine errors do not occur. While most distributed error detection schemes are optimized for a given error type, [53] provides a more general approach by using a hybrid fault model with an on–line error detection scheme.

However, these works are targeted at large computer networks, and relatively poorly adapted to smaller embedded applications.

3.5.5 Reconfiguration and Graceful Degradation

An alternative to the methodology used by the XBW team is to use a more flexible architecture, which allows one to optimally modify the system's organization in the case that an error is detected. This approach makes use of dynamic **reconfiguration**. The induced flexibility enables the use of diverse redundancy, which was shown to exhibit interesting cost optimization properties.

However, the absence of exact redundancy implies that the functionality level may decrease (be *degraded*) following the detection of an error and subsequent reconfiguration. In a safety–critical system, the top priority is to guarantee that operation of the system is safe, i.e. that the potential degradation doesn't affect the safety of the system, and happens in an elegant, optimized way. The exact term used in the literature is **graceful degradation**[20].

Reconfiguration

A system is defined as **reconfigurable** if it can be modified, either in software or in hardware. By modified one means that the tasks which provide the system's functionalities are re–allocated among the available soft and hard components of the system. A system which can be reconfigured during run–time is called *dynamically* reconfigurable. Reconfiguration can be a useful tool for fault–tolerance purposes, as was introduced above.

Kimura *et al.* use a distributed reconfigurable network of ECU's to control hyper–redundant space robot manipulators[24]. Their system is able to adapt itself in real–time to partial failures and to the changing operating conditions through the use of a distributed consensus algorithm. The inverse kinematics problem is solved on all ECU's, and the optimal solution (most feasible) is selected and applied after exchanges over the communication channel have led to an agreement.

Oldknow has developed a dynamically reconfigurable architecture for machining applications[36]. The goal of his work is to optimize the machining process from an economic point of view. Depending on the operating conditions, different constraints are possibly active (e.g. tool edge breakage, tool shank breakage, contouring accuracy, etc). Relying on a reconfigurable architecture, his system is able to adapt itself to the current active constraint, in real–time.

Wills *et al.* present a reconfigurable architecture in [55]. The presented work is illustrated with the example application of an unmanned helicopter. In this application, altitude of the prototype helicopter is controlled either by varying the collective pitch setting of the main rotor blades or by changing its rotational speed. In normal operation, the former method is used. However, upon failure of the blades actuators, dynamic reconfiguration of the system "switches" the control to the latter possibility, thereby ensuring that the system provides continued (though slightly decreased) performance.

A further application of reconfigurable systems lies in possibilities for off–line modifications. If the system's architecture is flexible enough, modifications such as sensor upgrades are possible in a way that is transparent to higher–level software. [36] in particular details a way to use self–instantiating intelligent components, from which the code required to use them is uploaded to the high–level controller, hence providing "plug and play" functionality.

Task (re–)allocation

Two separate approaches are available to reconfigure a distributed computing system as a means of error treatment: either a look–up table is used, where component states are readily mapped into given configurations, or an automatic algorithm is used to optimally re–allocate the tasks. The former technique is extremely efficient to execute, but its implementation can prove to be particularly tedious – if not impossible – in complex systems, since the designer has to consider all possible combinations of states. In the latter technique, the optimal allocation is defined with respect to resource utilization, load balancing on the ECU's, minimized system disturbance and performance level.

Different algorithms are available in the literature to achieve optimal allocation and re–allocation of tasks among a distributed network of computers[51, 52, 57]. These algorithms ensure that tasks, which were performed by faulty processors, are re–allocated to the remaining fault–free processors. The basic problem to be solved is the bin–packing problem: a number of bins (the processors) are available, in which a number of tasks are to be packed. Both the bins and tasks are characterized by their size, which is a possibly multi–dimensional measurement of parameters such as required and available bandwidth or memory space. Each task can also be assigned an affinity for every processor, thereby making it possible to exclude some task/processor combinations (due to the absence of required I/O for example).

Unfortunately, these works are directed mostly towards large computer networks (as used in communication systems for example, where hundreds of processors are used in parallel), and are therefore difficult to directly use in a smaller embedded system context.

Furthermore, optimization algorithms such as those presented in [52, 57] reach an approximate optimum recursively, and therefore exhibit an execution time that can not be determined *a priori*. This makes them poorly suited to

hard real–time operation. An approach for dealing with this problem could be to guarantee at least minimal functionality in case of a detected error, thanks to rough, non–optimized techniques, and request that the boat or car be stopped by its driver, so that the re–allocation process can be conducted in an off–line, non safety–critical environment.

Graceful degradation: the RoSES project

As mentioned above, an approach to the fault–tolerance problem is graceful degradation. Graceful degradation is a particular combination of reconfiguration and task re–allocation, which deliberately tolerates performance decrease, but aims at ensuring that it happens in the most elegant and safe way possible. The most active research group currently tackling graceful degradation issues in distributed embedded real–time systems is RoSES (**Ro**bust **S**elf–configuring **E**mbedded **S**ystems). The RoSES project was started at Carnegie Mellon University in 2000, under the supervision of Dr. Phil Koopman.

The RoSES approach to graceful degradation is based on Product Family Architecture (PFA). A PFA is "a region of a system design space populated by different, but related products sharing similar architectures and components"[34, 35]. A PFA can be thought of as the whole range of offerings of a given manufacturer for one type of products. Depending on available options and price range, although presenting great similarities, the products differ in complexity and performance levels. The idea behind the RoSES project is that upon failure of some of the components of a system, it is possible to shift to another configuration, i.e. to another "product" in the PFA. Furthermore, the reconfiguration process is expected to be automatic and optimized.

However, the initial effort of the RoSES team concentrates on off–line reconfiguration of non safety–critical systems, and is therefore not directly applicable to the problem described in this thesis:

> Eventually reconfiguration will be done on–line in real–time [...] but in the near–term we will instead assume that the car is pulled to the side of the road for a minute or so while the reconfiguration takes place automatically[35].

Shelton and Koopman[46] present a preliminary effort to adapt graceful degradation to safety–critical applications, with the example case of an elevator control system. Their model classifies the system's functionalities into safety–critical and non safety–critical components, and ensures that all safety–critical functions are preserved through graceful degradation. In the elevator system example, this may lead to an elevator which ignores all user calls and travels one floor at a time, at low speed. Despite the substantial degradation in quality of service, the system is still able to perform its primary task (move people between floors) in a safe manner. Unfortunately, despite its promising goals, the described research is only in its early stages.

3.6 Overview of approach adopted

In this section, the approach taken for this thesis work is detailed. The general goal has been to minimize the overall system cost by lowering the number of components and optimizing the use of the available components. To achieve this, triple hardware diverse redundancy was used at both the sensor and ECU levels while the object–oriented model of the system described in 2.3.4 was combined with on–line error detection and both software and hardware dynamic reconfiguration utilities. Unless otherwise specified, all considerations made in the following subsections are from the local point of view of each ECU.

3.6.1 General description of architecture

Within the object–oriented system model developed here, the methods are categorized as either *atomic* methods or *high–level* methods. Each atomic method has a one–to–one correspondence to a given sensor or actuator. In particular redundant sensors are represented by redundant atomic methods. For instance, the object representing a physical unit comprised of 3 redundant position sensors `sensor1`, `sensor2` and `sensor3` will contain 3 atomic methods, e.g. `sense1`, `sense2` and `sense3`. On the other hand, high–level methods sit at a higher level in the organization, and are independent of the low–level hardware. A typical example of a high–level method is one that performs a closed–loop control action: it takes a reference value and a feedback value, compares them, applies a control strategy and outputs a command. By definition, high–level methods are not "aware" of the low–level redundancy: they manipulate data in *single* form, as opposed to *redundant* form. It is therefore necessary to implement a conceptually intermediate set of utilities in charge of managing the redundancy, i.e. of transforming the data from the sensing atomic methods and to the actuating atomic methods respectively from redundant form to single form and vice versa.

This hierarchically intermediate level is called the *redundancy management level.* In addition to its redundancy management role, it is also in charge of error detection, and is comprised of both local and distributed error detection utilities and of **Actuation Managers**, grouped inside the **Redundancy Manager**. In parallel with these utilities, the **Execution Table** is used for dynamic software reconfiguration in the presence of an ECU error.

Within the proposed architecture, it is assumed that a deterministic, fault–tolerant communication channel is used, which complies with SAE Class C requirements[26]. The implementation of such a communication channel requires

[26]See 3.4

some redundancy, as well as the existence of means to synchronize the nodes of the network, as detailed in 3.4. It is also assumed that all inter–ECU communications happen according to a known, predefined schedule.

The above described organization is shown schematically in Fig 3.3.

The example application of Fig 3.3 is comprised of 2 physical hardware units, each of which acquires physical data from 3 redundant sensors. Physical unit B also contains 2 redundant actuators. Data from the sensors is acquired by the ECU, and data from the ECU is sent to the actuators through I/O ports (which feature ADC and DAC facilities as required). The grey boxes represent objects instantiated from classes of the Virtual Vehicle object–oriented model[27]. Inside these boxes, white boxes or ellipses represent methods associated to the corresponding objects. It is important to note that the view given in Fig 3.3 is only locally valid from the point of view of one of the ECU's of the system. There are typically two other ECU's in the system, which hold a similar local vision of the system. In the specific described application, communication between the ECU's is used only by the Distributed Error Detection utility.

Treatment of sensor and actuator errors is done locally inside the Redundancy Manager. As mentioned above, the detection of ECU errors is, on the other hand, handled using dynamic reconfiguration strategies. The following sections describe the Redundancy Manager and Execution Table organizations as well as solutions for dynamic hardware reconfiguration.

3.6.2 The Redundancy Manager

As stated in 3.6.1, the Redundancy Manager is comprised of both Error Detection utilities and of Actuation Managers. This section describes the organization of error detection utilities within the Redundancy Manager, and concludes with a description of the Actuation Manager.

[27] See 2.3.4

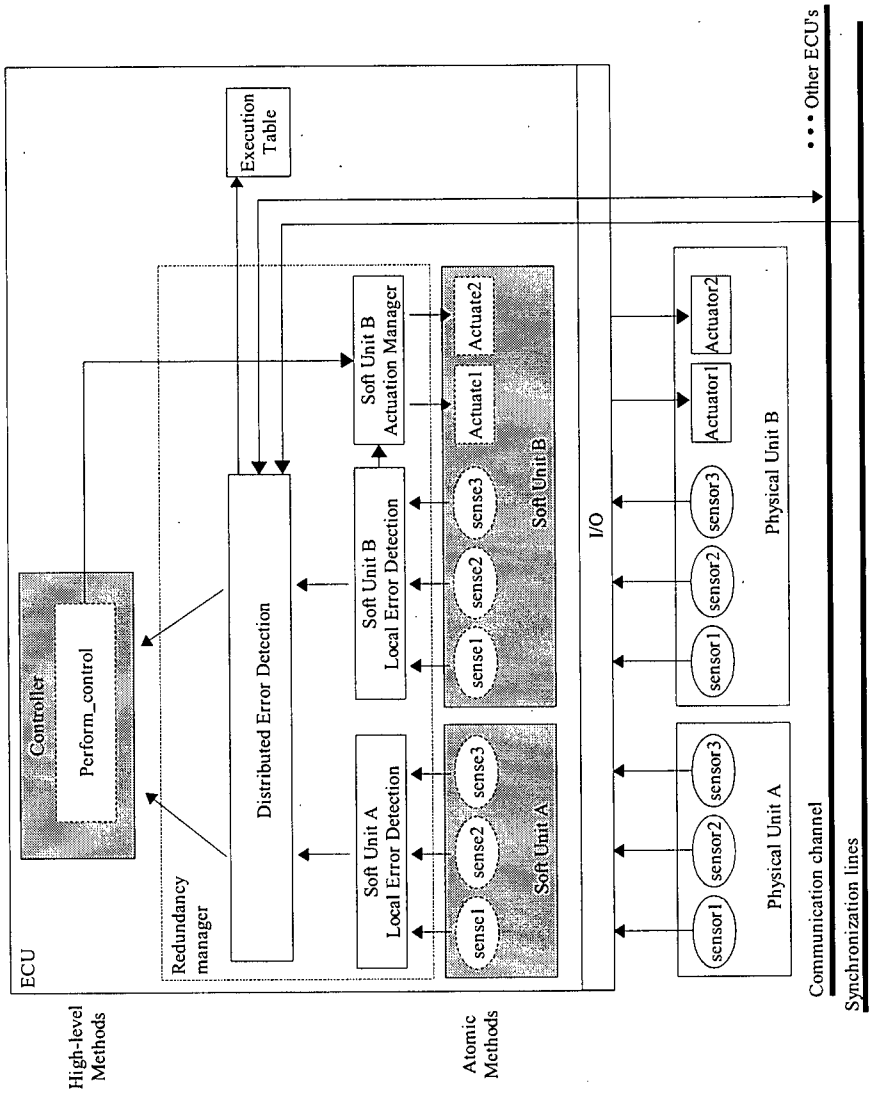

Figure 3.3: *System architecture: local view of the system*

Error detection utilities

As shown in Fig 3.3, the error detection utilities combine an error detection role and a redundancy management role. Therefore, their goal is twofold:

- determine the state of all physical devices in the system,
- and provide a correct, single reading from redundant sensors (even if one of them is faulty).

Knowledge of each component's state is essential from the point of view of error treatment: if actions are to be taken upon detection of an error, it is necessary to accurately know which components are faulty and which are not. In particular it is essential to have the ability to distinguish sensor or actuator faults from processing faults (i.e. ECU faults or failures), as the response to these two possibilities will obviously be radically different. In most cases however, because of the propagation property of faults[28], the detection of an error doesn't provide enough information to determine which component is faulty. For instance, if an ECU receives an incorrect sensor reading from another ECU over the communication channel, it is possible that the fault occured at many different levels: either the sensor itself is faulty, or the remote ECU, or the sensor/ECU link, or the communication channel. This cross-level coupling makes it difficult to develop error detection strategies targeted at specific levels. In order to solve this problem, the error detection process is provided by two distinct types of utilities:

- the **Local Error Detection** utilities are in charge of sensor redundancy management and of primary sensor error detection,
- the **Distributed Error Detection** utilities are in charge of ECU redundancy management and of ECU error detection.

[28] See 3.2.1

The following subsections detail the organization of these two types of utilities.

Local Error Detection

The Local Error Detection utilities are in essence inexact voters. There is one Local Error Detection utility for each set of diversely redundant sensors, which processes the measurements from these sensors (M_1, $M_2 \ldots M_n$ for a n–redundant sensing unit) and outputs:

- an estimated correct measurement M_e for the measured parameter,
- and a diagnosis of the correctness of the individual redundant measurements: the "measurement state" vector MS. For a triple redundant sensing unit, $MS \in \{0,1\}^3$ (a 0 indicates an incorrect measurement, a 1 indicates a correct measurement).

This is illustrated in Fig 3.4.

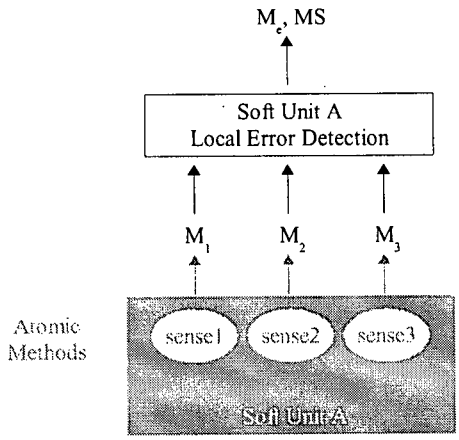

Figure 3.4: *Local Error Detection*

Different strategies can be applied to determine MS: as described in 3.2.1, executable assertion techniques compare each individual measurement to pre-

defined sensor models while data replication techniques use majority voting mechanisms. When redundant analog sensors are used, the majority voting mechanism uses characteristics of the sensors contained in the class definitions of the units that they belong to (e.g. accuracy, calibration curves, etc) to compare the sensor measurements two by two and determine whether they are in agreement or not within their accuracy limits. For this determination it is also necessary to take into account the fact that the measurements are usually conducted in a sequential manner, and that the actual parameter being measured therefore varies slightly from the first to the last measurement. From the results of these comparisons it is then possible to determine if a minority of measurements do not agree with the majority, i.e. to identify one incorrect measurement if TMR sensing units are used.

It is important to note that MS does not necessarily indicate the sensors' states. An incorrect measurement can be the result of either a faulty sensor, or a faulty sensor/ECU analog link, or a faulty data acquisition (typically analog I/O), or a computing fault at the ECU level.

Once MS has been determined, the Local Error Detection utility computes a partial averaging of all the measurements which were diagnosed correct. This provides the estimated correct measurement M_e for the given sensing unit.

The Local Error Detection utility is coded as a generic class, from which the actual utilities are instantiated as objects for each sensing unit. Considerable source code reduction is achieved by using this technique.

From the above description of the Local Error Detection process it appears that for a TMR sensing unit, both transient and permanent single sensor faults are tolerated by the system. Each measurement, whether correct or not at a given point in time, will be acquired and tested again by the Local Error Detection utility at the following loop–closing. If determined correct at this next iteration, it will be included in the partial averaging; if determined incorrect, it will not be included in the partial averaging.

The availability of MS is interesting for two reasons:

- it is useful for monitoring functions: the detection of a permanently incorrect measurement should be reported to the user, along with instructions for further testing and component replacement,
- it can be used by the Distributed Error Detection utility for ECU error detection, as detailed in the following subsection.

Distributed Error Detection

The Distributed Error Detection utility has a more complex role: it manages the ECU redundancy and provides ECU error detection in both the value and time domains. It outputs:

- agreed–on "correct measurements" M_c for each sensing unit. These measurements constitute the inputs to the high–level methods, as described in Fig 3.3.
- a diagnosis of the state of all ECU's in the system: the "ECU state" vector ES. For a triple redundant ECU architecture, $ES \in \{0,1\}^3$.
- other state vectors, such as "sensor state" vectors SS or "link state" vectors LS, depending on the desired level of system functionality.

The Distributed Error Detection utility is comprised of multiple **Value ECU Error Detection** utilities and of one **Combined ECU Error Detection** utility, as shown in Fig 3.5.

There is one Value ECU Error Detection utility for each sensing unit. For a given sensing unit, the corresponding Value ECU Error Detection utility compares the results from the Local Error Detection conducted at the local ECU and at the remote ECU's. The latter results are obtained *via* the communication channel. Operation of the Value ECU Error Detection utility is based on

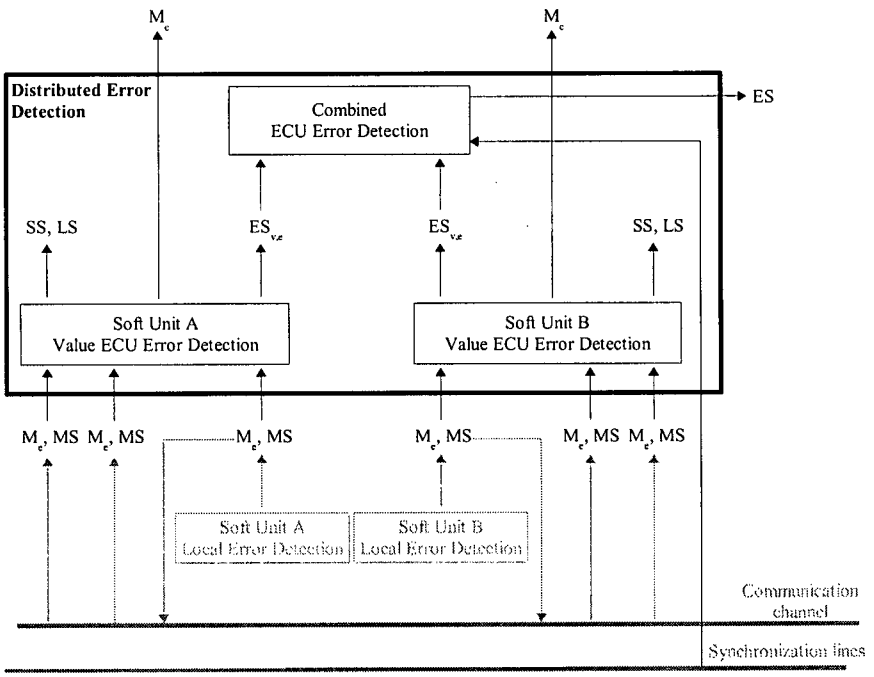

Figure 3.5: *Distributed Error Detection*

the assumption that non-faulty ECU's produce correct Local Error Detection results. In the reciprocal case, an incorrect Local Error Detection result will thus indicate a faulty ECU.

In normal operation, when triple ECU redundancy is available (i.e. no ECU has been diagnosed faulty), each Value ECU Error Detection utility applies inexact majority voting and partial averaging to the – local and remote – estimated correct measurements M_e. This process is similar to the approach used by the Local Error Detection utilities to treat redundant atomic measurements. It produces the correct measurement M_c for the considered sensing unit (result of the partial averaging), as well as an intermediate ECU state vector, the "estimated value ECU state" vector $ES_{v,e}$. $ES_{v,e}$ is the equivalent of MS: it indicates if the estimated correct measurement M_e of a given ECU is correct. It is also possible to combine the measurement state vectors MS to provide a refined diagnosis of the system components' states, e.g. by using a look-up table as follows: if all 3 ECU's indicate that M_1 is incorrect, one can conclude with reasonable certainty that `sensor1` is faulty, whereas if only one ECU indicates that M_1 is incorrect, it is very likely that `sensor1` is non-faulty but that the corresponding sensor/ECU link is faulty. From these observations it is possible to construct for instance the sensor state vector SS and link state vector LS. These indicators are useful for monitoring and repair purposes. Under certain circumstances, the redundancy level may be reduced to double. This can happen for instance if one of the ECU's is declared faulty and the system is reconfigured to 2-node operation[29]. In this case, it is not possible to use majority voting anymore, and the use of a look-up table described above can be extended to the estimation of the ECU states from the knowledge of the MS vectors. This provides the system with a downgraded but continued ECU error detection functionality.

The Combined ECU Error Detection utility combines the $ES_{v,e}$ vectors from all Value ECU Error Detection utilities, thereby building the value domain

[29] See 3.6.3

ECU state vector ES_v, which indicates which ECU – if any – is faulty in the value domain. This makes use of the intrinsic system redundancy: there are several physical units and hence several Value ECU Error Detection utilities. The Combined ECU Error Detection utility is also in charge of detecting ECU errors in the time domain. This is achieved by checking that all synchronized actions (control loop–closing, scheduled communications) happen as scheduled. For instance if a message expected from another ECU is not received during its allocated time, the Combined ECU Error Detection utility is informed and interprets this as a time error from the considered ECU by updating the time domain ECU state vector ES_t. Finally ES_t and ES_v are combined (typically by a logical "AND") to provide the overall ECU state vector ES. This is schematically shown in Fig 3.6.

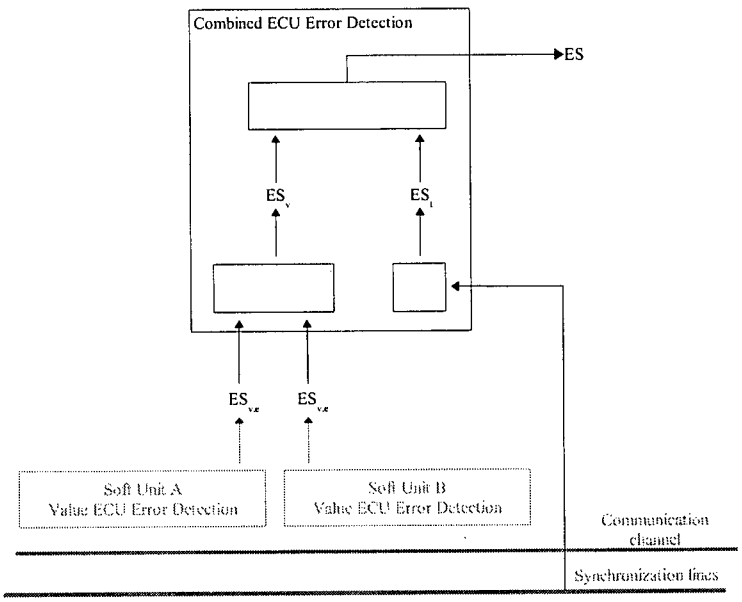

Figure 3.6: *Combined ECU Error Detection utility*

It should be noted that the approach adopted for ECU error detection makes use of data that is actually used by the application (here by the controller). This is refered to as *on–line* error detection. This technique allows one to avoid the use of totally independent ECU error detection schemes and thereby to reduce the communication and computing bandwidth requirements associated with ECU error detection.

It is also essential to bear in mind that the results from the Distributed Error Detection utility only provide the diagnosis of the system components states from the perspective of the considered ECU. Therefore these results can be erroneous if this ECU is itself faulty. 3.6.4 details how hardware majority voting is implemented to detect such a possibility and reconfigure the system in order to isolate the faulty ECU from the rest of the system.

As with the Local Error Detection utility, the Distributed Error Detection utility is defined as a class, which itself contains instance variables defined as objects from the Value ECU Error Detection and Combined ECU Error Detection classes. Here again, high code efficiency is achieved through the use of object–orientation, since the Value ECU Error Detection class definition is used for the instantiation of multiple objects.

Actuation Manager

Some high–level methods (e.g. in controller objects) output an actuation command which is generally in single form. However, the actual physical actuation is provided by redundant actuators, which are represented in software by redundant atomic methods. The Actuation Manager is therefore in charge of sending the appropriate command to each of the redundant actuating atomic methods, so that the total actuation complies with the order given by the high–level method. For example, in a torque–control system where actuation is provided by redundant DC motors, the Actuation Manager will divide the total commanded torque into partial torques to be applied by each DC motor through

the corresponding atomic methods.

The second role of the Actuation Manager is to provide actuator fault–tolerance. The behavior of the actuators is monitored using sensors, and comparison of the sensors' readings with a soft system model indicates when an actuator failure occurs. The detection of a sensor reading which does not comply with the system model is however not enough to determine if the fault is lying at the sensor or actuator level. Therefore it is necessary to either use redundant sensors to sense a given parameter, or to use redundant actuator fault identification strategies, which rely on different sensed parameters. If an actuator is declared faulty, the Actuation Manager automatically adapts the redundant actuation commands to be sent to the non–faulty actuators[30]. The Actuation Manager is schematically shown in Fig 3.7.

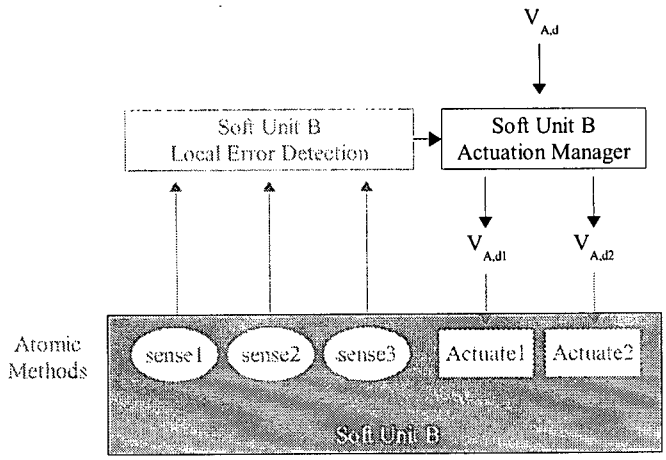

Figure 3.7: *Actuation Manager*

As with the Local Error Detection utility, the Actuation Manager is coded as a class, which is implemented as as many objects as required by the specific

[30]The choice of fail–silent actuators as described in 3.5.3 is therefore of essential importance to ensure that the faulty actuator, to which no command is sent, does not block the non–faulty actuators

implementation. For instance in a Steer–by–Wire system featuring tactile feedback as described in 2.3, there are two units to which actuation commands are sent (helm and drive units) and therefore two Actuation Manager objects would need to be instantiated.

It is important to understand that several ECU's may have the physical ability to send a command to given actuators (i.e. because they feature the I/O functionalities and analog links required for such an action). This actuation command redundancy is in fact a requirement for fault–tolerance, as detailed in 3.5.1. However, for each physical actuating unit, at a given time, the command to all actuators within this actuating unit is provided by one ECU only. Therefore there needs to be a way of dynamically switching to another command source (i.e. another ECU featuring the required I/O) in case that this primary ECU is diagnosed as faulty. This is detailed further in 3.6.4.

3.6.3 Software reconfiguration: the Execution Table

As previously mentioned in 3.6.1, detected ECU errors are treated by using a combination of software and hardware dynamic reconfiguration schemes. This section details the organization of the Execution Table, which is used for software dynamic reconfiguration[31]. The approach adopted is partially based on the features of the Forth programming language[4], in which the application was coded. Therefore this section starts by introducing some of the key concepts associated with Forth.

Some interesting properties of the Forth programming language

Forth is an "interpretive, stack–based postfix language most often used in embedded controls applications due to its tendency to produce a very efficient, compact code in resource constrained environments"[36]. A Forth system is

[31] See Fig 3.3

comprised of **words** (equivalent to functions or programs in other programming languages), which together form a **dictionary**. The dictionary can be extended by the user at any time: new words are defined as combinations of existing words. Parameters are passed to and from words using the data stack, which is a last–in–first–out (LIFO) buffer[56]. Words are usually described by their stack diagram, which represents their inputs and outputs. For instance, the stack diagram of a word `WORD1`, which requires 2 parameters `par1` and `par2` and outputs one value `val1` is as follows:

`WORD1` *(par1 par2 — val1)*

Forth's interpretive nature makes development and on–line debugging extremely efficient: each word can be compiled into the dictionary and tested independently, without requiring the compilation of the overall application or a test program. Words can also be redefined during run–time. This property provides some valuable flexibility to the embedded computer programmer, and is used extensively for software reconfiguration within the proposed architecture.

A concept of essential importance in this regard is that of **Execution Token** (XT). An XT is "a value, usually an address, that points to the execution behavior of a definition"[9]. In effect, it is a pointer to the address in memory where the execution behavior of a given word is stored. For instance, if `WORD1` is a word inside the Forth dictionary, its XT can be put on the stack by using the `['] WORD1` sequence, and calling the word `EXECUTE` will then execute it.

It is therefore possible to dynamically modify the execution behavior of a given word by redirecting its execution token to another address in memory where an alternate execution behavior is stored. This property was used to develop a dynamic software reconfiguration tool: the **Execution Table**, which is described in detail in the following subsection.

Lastly, it should be noted that Forth in itself is not an object–oriented language. However, a number of object–oriented extensions to Forth are available in the literature[36]. 4.2.2 describes the Forth object–oriented extension which

was used for the implementation of an experimental system as part of this thesis.

The Execution Table

In order to achieve maximal simplicity and predictability of the run–time behavior, the proposed architecture does not make use of multitasking. The execution flow on each ECU is instead defined as a single thread, which is called at regular intervals, according to the loop–closing frequency. Avoiding the use of a multitasker allows the designer to retain a clear knowledge of the actual code being executed at any time. In particular since concurrency is removed, the order in which the various pieces of code are executed is predefined and guaranteed.

Upon detection of a faulty ECU it is desired to dynamically modify the execution flow in order to adapt it to the new configuration. Such modifications can include the removal of all communication attempts with the faulty ECU, or the use of a downgraded ECU error detection scheme, as suggested in the description of the Distributed Error Detection utility[32]. In more advanced implementations, these modifications could also include a re–allocation of the tasks that were handled by the faulty ECU to the remaining fault–free ECU's, as described in 3.5.5. This will however not be considered in this thesis.

Within the proposed architecture, the execution flow is defined as a succession of *steps*. At each step, the executed code is chosen among several possible pieces of code, called **Maximum Size Bundles** (MSB's). A MSB is defined as the largest possible group of successive actions at a given step in the execution flow. In practice the MSB's are implemented as Forth words. They are grouped to form the Execution Table, which contains for each step:

- the list of possible execution tokens of this step's MSB's,
- and the index to the XT which should actually be executed (if any).

[32] See 3.6.2

This provides the opportunity to specify the "path" taken by the execution flow among the number of possibilities resulting from the multiple choices at each step. Once this path has been specified, the action of the high-level word called at each loop-closing (`MAIN`) consists therefore simply of scanning the Execution Table and of executing the specified XT at each step. If no XT is specified for a given step, this step is simply ignored and the next step is considered. By dynamically modifying which XT should be executed at each step, it is thus possible to modify the execution flow at run-time. The power of this approach resides in the possibility for the application being executed to modify its own execution flow. Some of the MSB's can indeed have the capability to modify the path taken by the execution flow, by simply redefining inside the Execution Table which XT should be executed at each step.

Fig 3.8 illustrates this organization with an example execution flow comprised of 6 steps.

In the example of Fig 3.8, steps 1, 3 and 4 offer a multiple choice of Maximum Size Bundles. Under the described specific configuration conditions, the MSB's represented by grey boxes are executed. In particular one can notice that no MSB is executed in step 4. Therefore the execution flow in the described configuration is:

```
MSB1a
MSB2a
MSB3b
MSB5a
MSB6a
```

Typically step 1 could involve data acquisition from the sensors, step 2 local error detection actions, step 3 exchange of data over the communication channel, etc.

The Execution Table can be implemented as a class, and instantiated as an

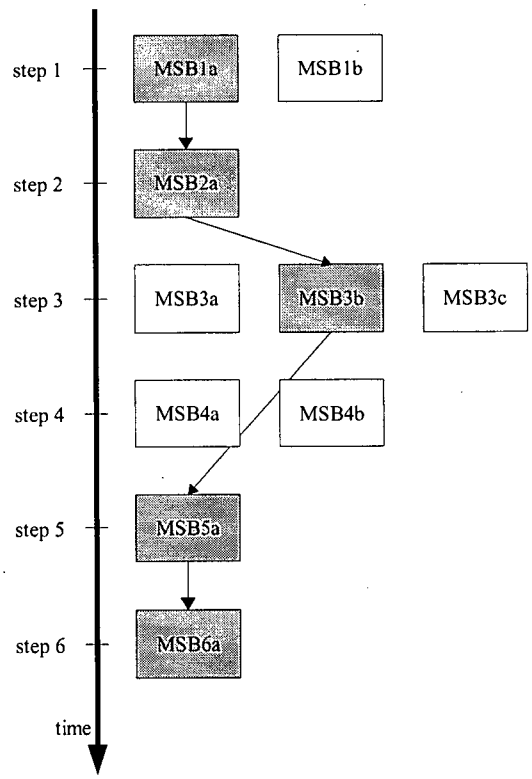

Figure 3.8: *The Execution Table: example execution flow*

object[33]. It contains the table itself, as well as all the necessary methods for its execution and modification. A look-up table can be used to indicate how the Execution Table should be modified depending on the current configuration and detected ECU error. This requires that all cases are examined at design time, but ensures that a low time overhead is associated with the run-time software reconfiguration, which is important in the safety-critical real-time context of By-Wire systems.

Here again it is important to understand that the software reconfiguration is done from the point of view of a given ECU, based on its own diagnosis of the states of the system components. A faulty ECU may produce erroneous ECU error detection results which will lead to the modification of its Execution Table and hence to dynamic software reconfiguration. Under these circumstances the fault-free ECU's should be able to recognize that this ECU is faulty, and consequently modify their own Execution Tables. It should also be noted that while the general system model, including the Virtual Vehicle and all fault-tolerance utilities, is identical on all redundant ECU's, there is a specific Execution Table and specific MSB's for each ECU. This allows the designer to define the interactions between the system's nodes. For instance one ECU may be used as a time master and trigger all synchronization signals while the other ECU's are used as time slaves, therefore waiting to get synchronization signals from the time master. In this situation it is obvious that the definition of the execution flow is dependent on the considered node.

Finally one should note that use of the Execution Table is not restricted to error treatment purposes: it can be used to simplify the source code and avoid complicated conditional "IF" loops.

[33]Note that this is not an obligation, since only one Execution Table is instantiated, and no code reduction results from the use of object-orientation. In some cases it may be more time efficient to define the Execution Table as a table, and simply write words to modify and execute it.

3.6.4 Hardware reconfiguration

As was previously emphasized in 3.6.2, the Distributed Error Detection utility outputs a diagnosis of the other ECU's states from a local point of view. These "local" diagnoses need however to be combined, in order to allow the non-faulty ECU's to reconfigure the distributed system and isolate the faulty ECU's from the rest of the system. It is for instance desired to remove the access of faulty ECU's to the communication channel, so as to avoid babbling idiot behavior[34], or as mentioned in the description of the Actuation Manager[35], it may be required to switch from an ECU which was declared faulty to one that is non-faulty for actuation command purposes. Such actions obviously need to result from a distributed, agreed-on decision. This implies that majority voting is applied to the local ECU error detection diagnoses. It is also essential to ensure that faulty ECU's do not affect the correct function of non-faulty ones.

Most ECU error detection algorithms, as described in 3.5.4, use complex schemes based on multiple rounds of communication to reach a consensus among the non-faulty ECU's: the local diagnoses resulting from each node's ECU error detection are exchanged and compared in software through majority voting. This approach is complicated to analyze, and it consumes excessively large communication and computing bandwidths.

In order to provide a simplified, more efficient way of reaching a distributed consensus, a novel approach has been developed. The basic concept used is that of an external hardware majority voting logic, to which the local ECU diagnoses are input, and which outputs a single, agreed-on distributed diagnosis for each ECU by setting the level of a dedicated digital security line to either high (faulty ECU) or low (non-faulty ECU). There is one security line for each ECU. Any device with access to these lines therefore holds a complete knowledge of the distributed ECU states consensus. This concept is illustrated in Fig 3.9.

[34] See 3.3

[35] See 3.6.2

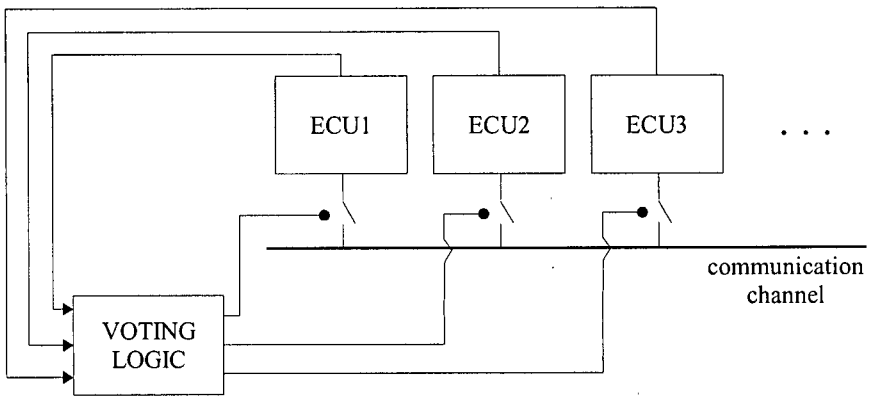

Figure 3.9: *Hardware reconfiguration (isolation of faulty nodes using external hardware voting logic)*

In the example application of Fig 3.9, the security lines (outputs from the voting logic block) control the ECU's access to the communication channel, by opening or closing digitally controlled switches. Any ECU whose the security line is high is therefore automatically denied access to the bus.

In order to adapt this concept to the proposed triple redundancy architecture, a stateline–based approach was used. Statelines are distributed, open–collector lines. This means that a given stateline is in high level (+5V) only if all the nodes connected to it write a logic high value (+5V) to it. Reciprocally if any of the connected nodes write a logic low value (0V) then the stateline is low. The resulting behavior is therefore similar to that of a "distributed" AND gate.

Fig 3.10 describes how this is used for reaching distributed consensus.

In Fig 3.10, SL_i ($i \in \{0,3\}$) represents the stateline (security line) indicating ECU_i's state. It is accessed by both other ECU's. For example both ECU_2 and ECU_3 can write a logic value to SL_1. The result is that SL_1 is at a high level (indicating that ECU_1 is faulty) only if both ECU_2 and ECU_3 write a

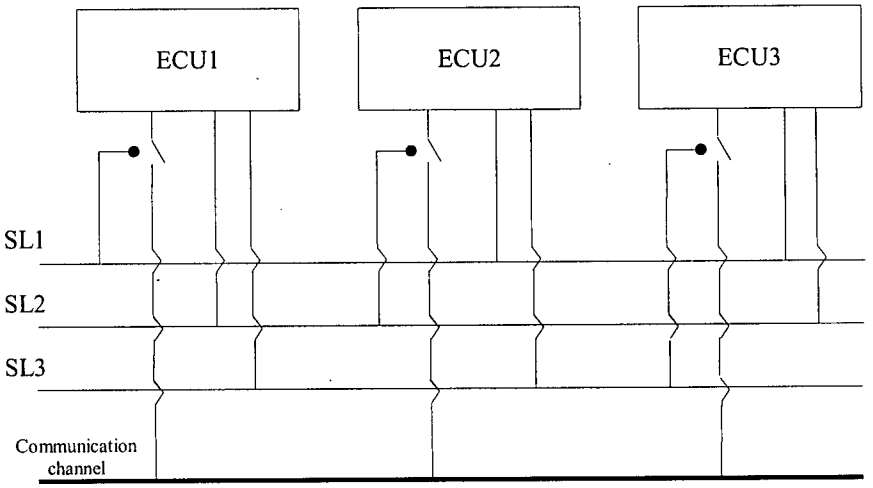

Figure 3.10: *Isolation of faulty ECU's using statelines*

logic high value to it (i.e. only if both ECU_2 and ECU_3 have diagnosed ECU_1 as faulty). SL_1's level can then be used e.g. to control a digital switch and thereby isolate ECU_1 from the rest of the system. It is important to note that using this technique, a faulty ECU is not able to switch a non–faulty ECU's communication access. Furthermore, it is possible to force faulty ECU's to write logic low values to all statelines, again typically using a switch controlled by the level of this ECU's stateline. In the case where the system has been downgraded to 2–node mode after the isolation of a faulty ECU, it is then guaranteed that none of the 2 remaining ECU's can switch the other one off.

The chosen implementation builds on the UBC Open Architecture, which has been developed at the University of British Columbia's Manufacturing Engineering Laboratory, under the leadership of Dr. Yellowley. In the UBC Open Architecture, statelines are used to monitor multiple constraints simultaneously, and adapt the control of a machining process in real–time, so that the process is done in a cost–optimized manner, under the various operating conditions

encountered[62].

3.7 Summary

In By–Wire systems in general, and marine Steer–by–Wire systems in particular, all mechanical and hydraulic links between the driver and the actuating units are removed. The main challenge in the development of such a system is therefore to achieve its fault–tolerance, that is to guarantee that continued functionality is provided even if one (or more) of the system's components is faulty.

In this chapter the terminology used to describe faults was defined along with common fault sources for each of the typical components of a marine Steer–by–Wire system as enumerated in 2.3.3. In particular it was shown that it is necessary to jointly consider the distributed and hard real–time natures of the networked computing system and to provide ways to guarantee its fault–tolerance. In this regard, communication protocols adapted to safety–critical hard real–time applications were reviewed, including TTP/C, TTCAN and FlexRay. It was shown that CAN is not suitable to the tight requirements of these applications if used in an event–triggered manner, but that it can be used in a time–triggered manner, as it is for instance within TTCAN.

An extensive literature review of research projects towards fault–tolerance of embedded systems was provided. Aeronautics applications such as the Boeing 777 Fly–by–Wire commercial airplane were described. It was shown that their design relies on a high level of redundancy, and is therefore too costly for use in pleasure boats applications.

The "X–by–Wire" project was also described. Its goal was to design an architecture for fault–tolerant, safety–critical automobile systems. This architecture is based on the fail–silent property of all components. In the X–by–Wire organization, two fail–silent units are combined to produce a fault–tolerant unit. This results in quadruple redundancy at all levels of the system.

Error detection techniques found in the literature were described and discussed, and approaches based on reconfiguration and graceful degradation were reviewed. The use of reconfiguration for fault–tolerance purposes allows one to design a flexible system, which can adapt itself to the new conditions in case that a faulty component is detected. The graceful degradation approach accepts that the level of functionality is decreased after such a reconfiguration, but aims at ensuring that this is done in the most elegant and efficient way, while basic safety–critical functions are maintained.

Lastly a novel architecture based on triple modular redundancy was proposed. Within this architecture, the hardware redundancy is managed by the Redundancy Manager, which comprises multi–level error detection utilities as well as Actuation Managers. The Local Error Detection utilities perform primary sensor error detection, and provide the data that is used by the Distributed Error Detection utility for ECU error detection. Actuation Managers distribute the high–level actuation command to the redundant physical actuators. Sensor error treatment is provided locally by the Local Error Detection utilities while ECU errors on the other hand are treated by using a combined software and hardware dynamic reconfiguration approach. The Execution Table is used to dynamically modify the execution flow on each ECU. It is based on the flexibility of the Forth programming language. Finally a stateline approach was proposed to reach a global consensus among the non–faulty ECU's and dynamically isolate the faulty ECU's from the rest of the system, in an simple and efficient manner.

This architecture has been applied to the design and test of a laboratory demonstrator, which is described in the following chapter.

Chapter 4

Design and test of a laboratory demonstrator

4.1 Introduction

In order to apply and test the concepts developed thus far, an experimental system has been designed and built in the Product Development Laboratory at UBC. This experimental system models a fault–tolerant marine Steer–by–Wire system, as described in 2.3. The system implements only simple position control functionalities (in particular it does not provide tactile feedback). It is designed to tolerate faults at the sensor and ECU levels. The design work consisted of both the hardware and software development of the prototype. This chapter provides a detailed description of the experimental system.

The first part of the chapter describes the components of the setup and the choices made for communication and synchronization. In the next section the organization of the object–oriented model of the system and of the utilities implemented to provide fault–tolerance is described. In the final section of the chapter a summary of the results achieved and limitations of the system is given.

4.2 Setup description

The demonstrator is comprised of 2 main hardware units: the **Helm Unit** and the **Drive Unit**, and is controlled by a distributed computing platform. It implements a fault–tolerant position control arrangement, which follows the guidelines described in 2.3.2.

Fig 4.1 shows a photograph of the setup.

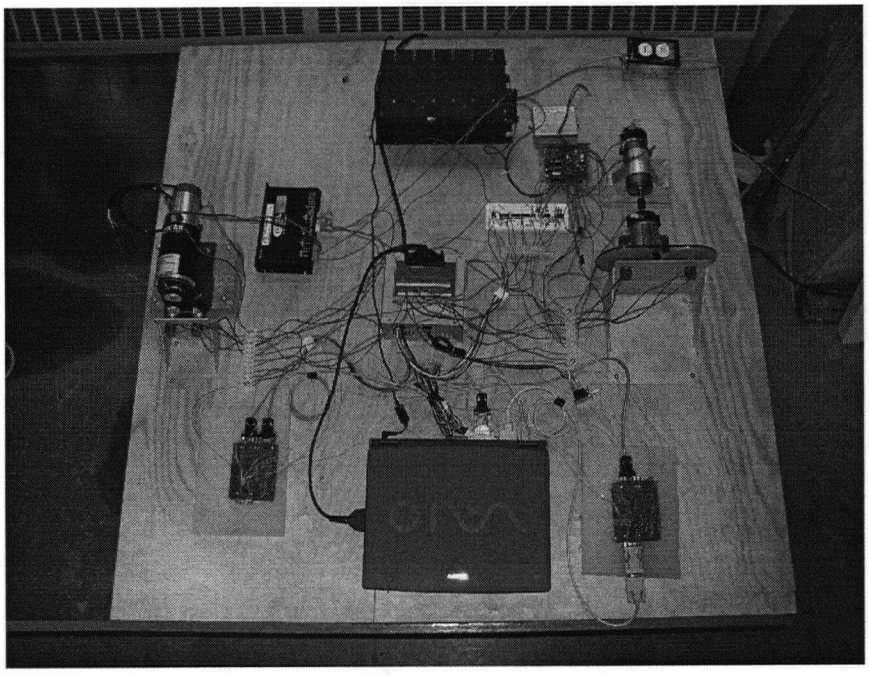

Figure 4.1: *Demonstrator set–up (photograph)*

4.2.1 Hardware components

This subsection describes the hardware components used for building the demonstrator, and provides a rationale for their choice. Table 4.1 contains a complete listing of all hardware components used.

	Component	Description
Helm Unit	DC motor	*Magmotor S28-I-300ET1*, with coupled tachometer
	Servo amplifier	*Copley Controls Inc. 513*, with velocity control mode
	Potentiometer (*2)	*Bourns 3540S-1-104ND*, 10 turns, 100 kΩ
Drive Unit	DC motor	*Pittman 14204B352 30.3*
	DC motor	*Electro-Craft 138000*, coupled to the Pittman motor and used as a tach
	Servo amplifier	*Copley Controls Inc. 4122D*, with PWM to analog converter and velocity control mode
	Potentiometer (*2)	*Bourns 3540S-1-104ND*, 10 turns, 100 kΩ
Shared	Power supply	*Copley Controls Inc. PST-075-10*
Computers	Laptop	*Sony/Intel Pentium II* 600 MHz laptop, running MS Windows 2000
	Data acquisition card	*Quatech DAQP-208*, 12-bit PCMCIA A/D and D/A card
	CAN adapter	*ProControl AG. FULL-Can Adapter*, connected through the parallel port
	Embedded computer (*2)	*Triangle Digital Services TDS2020F*, with on board Forth and attached TDS2020CAN board

Table 4.1: Components of the demonstrator

Helm Unit

The Helm Unit constitutes the input to the system. For testing purposes the Helm rotation is provided by the combination of a DC motor, a servo amplifier used in velocity mode and a function generator. However, it is also possible to manually operate the helm[1], as the user of a Steer–by–Wire system would do. For the purpose of this work it will be assumed that the input is provided manually, and command of the DC motor will therefore not be considered in further discussions.

Triple modular redundancy is achieved through the use of 2 redundant potentiometers and a tachometer, all coupled to the DC motor shaft. The tachometer signal is later modified by analog circuits as detailed in Appendix A, and integrated to provide a third position reading, as detailed in 4.4.1. This ensures diversity of the redundant position readings.

The potentiometers allow 10–turns lock–to–lock, and a $5V$ voltage is applied across them. The potentiometer gain is thus given by:

$$K_{pot,H} = 0.08V.rad^{-1}$$

The tachometer gain is supplied by the manufacturer:

$$\begin{aligned} K_{tach,H} &= 3V.kRPM^{-1} \\ &= 28.6 \cdot 10^{-3}V.\left(rad.s^{-1}\right)^{-1} \end{aligned}$$

Drive Unit

The Drive Unit is the actuating unit of the system and is shown in Fig 4.2.

The actuator itself (plant) is comprised of a servo amplifier and a DC motor. The DC motor is coupled to another DC motor, which is used as a tachometer. The servo amplifier features a velocity control mode, where the signal from the above mentioned tachometer is fed back to provide closed–loop control. This

[1] By manually turning the DC motor shaft

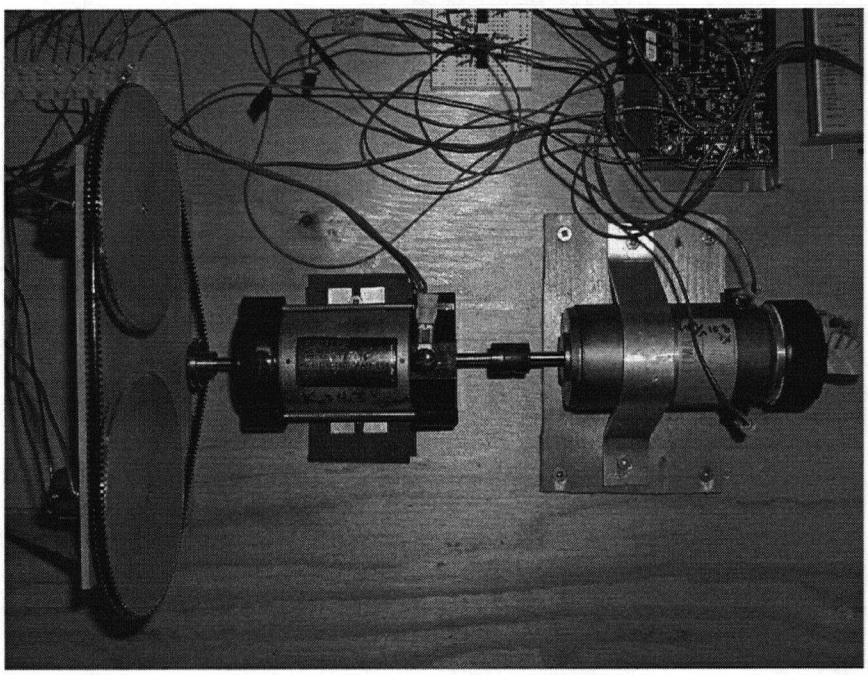

Figure 4.2: *Drive Unit (photograph)*

velocity controller is characterized by 3 tunable gains k_{ref}, k_{loop} and k_{tach}, as shown in Fig 4.3:

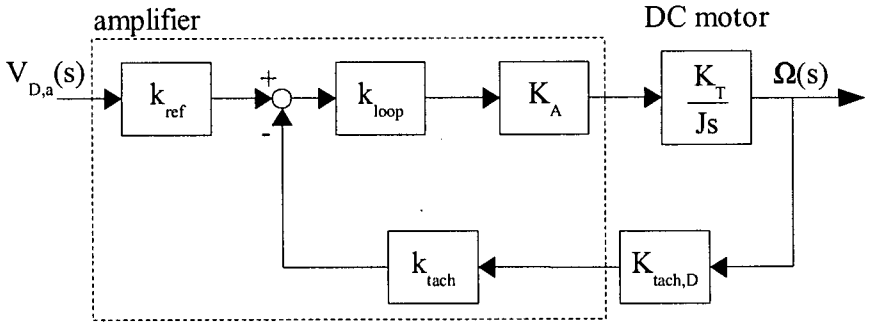

Figure 4.3: *Drive unit DC motor and servo amplifier control diagram*

From Fig 4.3 the relationship between the output velocity and the input voltage is given by Equ 4.1 in the Laplace domain:

$$\Omega(s)\left(1+\frac{K_T}{Js}\cdot K_A\cdot k_{loop}\cdot k_{tach}\cdot K_{tach,D}\right)=V_{D,a}(s)\cdot\frac{K_T}{Js}\cdot K_A\cdot k_{loop}\cdot k_{ref} \tag{4.1}$$

Therefore the global plant can be considered as a first order system:

$$\frac{\Omega(s)}{V_{D,a}(s)}=\frac{K_{vel}}{1+\tau_{vel}s} \tag{4.2}$$

where

$$\begin{cases} K_{vel} &= \dfrac{k_{ref}}{k_{tach}\cdot K_{tach,D}} \\ \tau_{vel} &= \dfrac{J}{K_T\cdot K_A\cdot k_{loop}\cdot k_{tach}\cdot K_{tach,D}} \end{cases} \tag{4.3}$$

The following numerical values characterize the system:

$$\begin{aligned} K_A &= 4A.V^{-1} \\ K_T &= 76.5 \cdot 10^{-3} Nm.A^{-1} \\ K_{tach,D} &= 41 \cdot 10^{-3} V.\left(rad.s^{-1}\right)^{-1} \\ J &= 2.61 \cdot 10^{-5} kg.m^2 \\ k_{ref} &= R^*/RH5 \\ k_{loop} &= RH10/R^* \\ k_{tach} &= R^*/RH7 \end{aligned}$$

where $RH5$, $RH7$ and $RH10$ are adjustable and R^* is an arbitrary coefficient which has the dimension of a resistance. The system was tuned to the following values:

$$\begin{aligned} RH5 &= 430k\Omega \\ RH7 &= 100k\Omega \\ RH10 &= 51k\Omega \end{aligned}$$

This yields the following first–order characteristics:

$$\begin{cases} K_{vel} = 5.7 rad.s^{-1}.V^{-1} \\ \tau_{vel} = 4 \cdot 10^{-3} s \end{cases} \tag{4.4}$$

The above parameters have been verified by applying a step input to the plant and measuring the steady–state gain and rise time.

The diverse, triple modular redundancy for position sensing is achieved in the same way as on the Helm Unit, through the use of 2 potentiometers and a tachometer. The tachometer is as described above. The potentiometers are identical to the ones used in the Helm Unit, however there is a coupling ratio of 4 to the DC motor shaft. Therefore the potentiometer gain is as follows:

$$K_{pot,D} = 0.02 V.rad^{-1}$$

Computers

The computing platform is comprised of 2 single–board embedded computers (TDS2020F)[42] and a Pentium based laptop. They will be called respectively "TDS boards" (TDS_1 and TDS_2) and "laptop" in subsequent discussions. They communicate over CAN bus[2].

The TDS boards are designed and manufactured by Triangle Digital Services Ltd; they are based upon the Hitachi H8/532 16–bit microprocessor[22]. The boards have significant analog I/O capacity (8 10–bit A/D channels and 3 8–bit D/A channels), 3 16–bit free running timers, one 8–bit timer and 2 watchdog timers. They allow 45 kbytes of address space for the user's compiled program as well as up to 512 kbytes of RAM or Flash memory. Depending on selected options (e.g. use of ADC, etc), there are between 26 and 41 parallel I/O lines available on the TDS boards, most of them being selectable as either input or output. The boards are supplied with an on–board Forth inside the H8 PROM[3]. Connection to the CAN network is achieved through the use of TDS2020CAN boards (see 4.3.1).

The range of inputs to the on–board ADC is $\{0, 5V\}$, while the on–board DAC outputs a PWM[4] signal at 19.6608kHz, which needs to be filtered (external low–pass filter) to produce an analog voltage ranging from 0 to $5V$.

The laptop is based on a Pentium II processor, running at a clock frequency of 600 MHz. The operating system used is MS Windows 2000.

Access to analog and digital I/O on the laptop is achieved through the use of a PCMCIA card[5], which offers 8 12–bit A/D channels, 2 12–bit D/A channels, 4 digital input lines and 4 digital output lines. The laptop is connected to the CAN network *via* a parallel port adapter[6].

[2] See 4.3.1
[3] See 4.2.2
[4] Pulse Width Modulated
[5] Quatech DAQP–208
[6] ProControl AG. FULL–CAN Adapter

The range of inputs to the laptop ADC is $\{-10, 10V\}$, with a selectable input gain: gain $\in \{1, 2, 4, 8\}$. This gives an effective input range selectable from $\{-1.25, 1.25V\}$ to $\{-10, 10V\}$. Output from the laptop DAC is an analog signal ranging from -5 to 5V.

The choice of non–identical ECU's was made deliberately to illustrate the concept of diverse redundancy. The main differences between the laptop and the TDS boards are:

- The Pentium II is a 32–bit processor, while the H8 is a 16–bit processor.
- The execution speed is much higher on the laptop (typically by a factor of 100).
- MS Windows 2000 is not a RTOS[7], therefore execution time for a particular word on the laptop can vary. In particular, words requiring use of DLL's (i.e. those interacting with either the data acquisition card or the CAN adapter) have a widely varying execution time. On the other hand, execution on the TDS boards is deterministic, therefore all words can be accurately timed in advance.
- Access to both analog and digital I/O is faster on the TDS boards. A single analog to digital conversion takes $21\mu s$ on the TDS boards[42], while it was measured to take between 850 and $950\mu s$ on the laptop. The bottleneck[8] is the digital to analog conversion time on the laptop. It was measured at an average of $66.5ms$.

Because of the extremely slow ADC and DAC conversion times on the laptop, one pass through the main application's loop takes about twice as much time on the laptop than it does on the TDS boards. Therefore operating all 3 nodes at the same loop–closing frequency would impose a loss of performance on the

[7] Real–Time Operating System

[8] Limiting factor for choice of sampling frequency

TDS boards. In order to avoid this, it was decided to operate the 2 TDS boards synchronously and at a higher frequency. The combination of the 2 TDS boards is considered the normal operation computing system. The laptop is operated at a lower loop–closing frequency, and is mainly present for monitoring and error detection purposes. In normal operation (i.e. when no ECU is declared faulty), TDS_2 is in charge of sending increments to the Drive DC motor. In the case that TDS_2 is declared faulty, the laptop has the capability of taking over the control actions and of commanding the Drive Unit actuation. This reconfiguration process is described in detail in 4.5 and follows the guidelines specified in 3.6.4.

It should be noted that the asynchronous use of the third ECU (here the laptop) is a realistic design choice. In a commercial integrated boat Steer–by–Wire system, one could imagine that 2 dedicated ECU's are in charge of the normal operation computing, while a third ECU provides the required triple redundancy by interacting with the system at a lower frequency, and performs other tasks the rest of the time (typically non real–time safety–critical tasks, such as user interface management, GPS management, etc).

Others

Power is provided to the servo amplifiers by a single power supply, as indicated in Table 4.1. Redundant power supplies were not used, and therefore the experimental system does not tolerate power failures.

It was necessary to amplify and shift some of the analog signals involved (particularly from the tachometers), as well as modify the actuation commands output by the laptop and TDS_2. Furthermore a switching mechanism was implemented to switch between the two aforementioned actuation command sources. All the corresponding circuits were implemented on a development board, and are described in detail in Appendix A.

4.2.2 Software components

As described in 2.3.4 and 3.6, the application is coded in Forth, and utilizes object–orientation.

ANS Forth

In order to guarantee portability of the application across hardware platforms as radically different as the TDS boards and laptop, Forth systems conforming to the ANS Forth standard[1] were selected. Forth systems anterior to the release of the ANS Forth standard were often comprised of sets of words specific to the intended applications and hardware platforms[36]. The ANS Forth standard is now the *de facto* Forth standard, and ensures that applications can be easily ported between different hardware systems and Operating Systems. Furthermore, the selected object–oriented extension described in the following subsection is ANS Forth compliant, hence confirming the value of this choice.

The Forth system supplied with the TDS boards (TDS–Forth v.4.01) is ANS Forth compliant[42]. As indicated in 4.2.1 it is the software core of the TDS boards, being burnt in PROM inside the H8 processor.

The Forth system used on the laptop is SwiftForth[14][9], which is specifically aimed at MS Windows programming, and also complies with the ANS Forth standard.

Object–oriented extension to Forth

As detailed in 2.3.4 and 3.6, object–orientation was extensively used in the definition of the system architecture. The Forth object–oriented extension used was developed by McKewan[32]. As explained above, it is ANS Forth compliant, and therefore directly usable on both the TDS boards and the laptop. It supports all major object–orientation concepts such as *inheritance* and *aggregation*[32].

[9]Developed by Forth, Inc.

Within the McKewan object–oriented extension to Forth, classes are defined between the delimiting words `:Class` and `;Class`. Inside classes, methods are defined between the delimiting words `:M` and `;M`. The last character of every method name must be a colon (":"). Once a class (e.g. `Class1`) has been defined, an object (e.g. `Object1`) can be instantiated from it by calling the following command:

```
Class1 Object1
```

It is then possible to access a method (e.g. `Method1:`) from the created object by calling `Method1: Object1`. A special method is the `ClassInit:` method, which is automatically executed when a new object is instantiated. It allows the designer to initialize the object, e.g. by setting the initial values of the instance variables.

4.3 Communication and synchronization

The chosen communication protocol is CAN. As detailed in 3.4, CAN itself is not adapted to the stringent requirements of hard real–time safety–critical applications. Therefore synchronization means were developed to ensure that communication is achieved in a deterministic way, by eliminating all conflicts for access to the bus. More generally, synchronization is also required to coordinate acquisitions and control actions across the ECU's. The following subsections detail the implementation of the CAN bus and of the synchronization lines.

4.3.1 Communication: CAN bus

Used CAN controller: Intel 82527

Both the FULL–CAN Adapter and TDS2020CAN[10] use the Intel 82527 serial communications controller[23]. It features a CAN controller, an embedded RAM

[10]See Table 4.1

and a CPU interface. There are 15 **message objects** available, which are individually software selectable as receiver or transmitter, except for the last one. Each message object can contain a maximum of 8 bytes of data, which are stored in RAM. The transmission rate is software selectable, and can be set to a maximum of 1Mbit.s^{-1}, as indicated in the CAN specifications[11].

The 14 first message objects are assigned an identifier, which can be modified at any time by the user. A receiver message object will automatically store in RAM any message with a matching identifier (which was put on the bus by a transmitter message object of another CAN node). The last message object can only receive messages, and has a programmable mask. This allows a node to receive messages with a large number of different identifiers, which match the mask properties.

High–level access to CAN: the CANmsg class

In order to provide a standardized, effective high–level access to CAN, the `CANmsg` class was implemented. This class models the low level "message objects". Each object instantiated from `CANmsg` is either a receiver or a transmitter. This is determined during the initialization of the object, depending on its number (corresponding to the hardware message object number on the CAN chip), and according to the predefined CAN object mapping[12].

In the `CANmsg` class definition of a CAN object message, messages are represented and handled as 4 2–byte values (hence reaching the maximum size of 8 bytes per message from the CAN specifications). This is well suited to this application, where CAN is mostly used to exchange digital angles, which are 11–bit integers[13], and measurement state vectors (MS), which can be expressed as 3–bit integers[14].

[11] See 3.4.1
[12] See following subsection
[13] See 4.4.1
[14] See 3.6.2

Furthermore, despite the radical differences in implementation (access to the CAN functionalities is achieved through the use of DLL functions on the laptop, while TDS provides words to access CAN directly in high–level Forth), high–level user access to the `CANmsg` objects is done in the exact same manner on both platforms.

Table 4.2 provides a detailed description of the most important instance variables and methods implemented in the `CANmsg` class. There can be up to 14 objects instantiated from the `CANmsg` class. The high–level object modelling the low–level message object #1 for instance is instantiated as `CANmsg1`.

CAN objects allocation

On each computer, CAN objects need to be allocated to application specific data exchanges. Since the CAN arbitration scheme is not used[15], there is no need to consider priority of message objects during the allocation. The attribution of the identifiers was therefore done in a simple, node independent manner: each message object is attributed an identifier equal to $\$10^{16}$ times its number. Furthermore, for the considered application, CAN message objects were only required to broadcast data from one node to the other two nodes during the Distributed Error Detection process[17]. Therefore only 3 high–level message objects needed to be implemented on each node, and the mapping described in Table 4.3 was used. Rx denotes a Receiver node, while Tx denotes a Transmitter node.

4.3.2 Synchronization scheme

The 2 TDS boards are required to operate in a synchronized manner: for instance loop–closing should happen simultaneously on the 2 boards, and acquisi-

[15] See details on synchronization in 4.3.2

[16] 10 in hexadecimal base

[17] See 3.6.2

Instance Variables	
Name	Description
`msg`	8–byte message (either to be sent or received)
`ID`	Identifier
`MsgObj#`	Message object number (1 to 14)

Methods		
Name	Stack Diagram	Description
`ClassInit:`	*(n —)*	Initializes the message object as either transmitter or receiver according to its number (*n*, specified on the stack) and the predefined CAN object mapping (see Table 4.3)
`Receive:`	*(—)*	Retrieves the received message from the CAN chip and updates `msg` (can only be used by Receiver objects)
`Send:`	*(—)*	Transfers `msg` to the CAN chip and initiates sending (can only be used by Transmitter objects)
`Create_msg:`	*(v4 v3 v2 v1 —)*	Creates the 8–byte message (`msg`) from the 4 2–byte values on the stack (for use by Transmitter objects)
`Get_msg:`	*(— v1 v2 v3 v4)*	Leaves the 8–byte message (`msg`) on the stack, in the form of 4 2–byte values (for use by Receiver objects)

Table 4.2: The CANmsg class

Object	Identifier	TDS_1	TDS_2	Laptop
CANmsg1	\$10	Tx	Rx	Rx
CANmsg7	\$70	Rx	Tx	Rx
CANmsg9	\$90	Rx	Rx	Tx

Table 4.3: CAN message objects mapping

tions should be performed at the precise same times to ensure that comparison between acquired sets of data makes sense (or at least to narrow down the maximum acceptable offset between acquired values[18]). Although it operates at a slower frequency, the laptop also needs to be synchronized with the TDS boards when it does interact with them.

Furthermore, as detailed in 3.4, the use of CAN's arbitration scheme to resolve conflicts for access to the bus implies *a priori* unpredictable variations of the transmission time, and it is therefore necessary to carefully schedule all communication exchanges in advance, so as to ensure that no two nodes can request use of the bus at the same time.

Lastly, the synchronization scheme should include some handshaking, as a means to check that other ECU's respond to requests in time, and therefore as a means to detect ECU time errors, as described in 3.6.2.

One way of achieving synchronization consists of using interrupts (typically a time master ECU triggers an interrupt on time slave ECU's by pulling a digital line high). It is however a good design practice to avoid use of interrupts in hard real–time computing systems (when possible) because the interrupt response time may cause time errors by delaying execution of tasks and having

[18]See 3.6.2

them miss their deadlines[19].

It was thus decided to use a polling mechanism instead. In essence the polling action consists of an infinite idle loop, during which the polling ECU is waiting for a given digital line to go from low to high level. To adapt this concept to the application described here, an advanced polling Forth word (`POLL`) was written. It allows an ECU to wait for the specified digital input line to switch to high level, and it outputs an error code if this doesn't happen within the specified, selectable time limit. This safeguard prevents a faulty ECU from holding another ECU in an infinite polling loop. On the other hand, careful choice of the time limit implies that its exceeding is equivalent to the detection of a time error from the expected pulling ECU. This is used at all synchronization points by the Distributed Error Detection utility, as described in 3.6.2.

Allocation of digital lines

This subsection details the actual allocation of digital lines, which was used in the implementation of the experimental system. On both TDS boards, `PortA` is an 8–bit digital output port, and `PortB` is an 8–bit digital input port. The first 2 bits of `Port7` are set as digital inputs. On the laptop, `DIN` is a 4–bit input port, and `DOUT` is a 4–bit output port.

Table 4.4 and Table 4.5 indicate the allocation of digital lines for inter–TDS synchronization and for laptop–TDS synchronization respectively. A description of the role of each line is provided. The following subsections detail the way in which these lines are used. Note that in all subsequent tables, arrows indicate the direction in which the digital signal is transmitted.

[19]If interrupts are to be used in a hard real–time system, the designer is therefore forced to apply an overly conservative worst–case execution time to the different tasks during scheduling[44]

TDS_1		TDS_2	Role
PortA(2)	$\longrightarrow$	PortB(0)	CANmsg sent notification
PortA(3)	$\longrightarrow$	PortB(1)	Loop–closing ACK
PortA(4)	$\longrightarrow$	PortB(2)	Laptop check ACK
PortB(0)	$\longleftarrow$	PortA(2)	CANmsg sent notification
PortB(1)	$\longleftarrow$	PortA(3)	Trigger loop–closing
PortB(2)	$\longleftarrow$	PortA(4)	Trigger laptop check

Table 4.4: Allocation of digital lines for inter–TDS synchronization

TDS_1		Laptop		TDS_2	Role
PortA(7)	$\longrightarrow$	DIN(0)			CANmsg sent notification
		DIN(1)	$\longleftarrow$	PortA(7)	CANmsg sent notification
Port7(0)	$\longleftarrow$	DOUT(0)	$\longrightarrow$	Port7(0)	Indicates that laptop is ready
Port7(1)	$\longleftarrow$	DOUT(1)	$\longrightarrow$	Port7(1)	CANmsg sent notification

Table 4.5: Allocation of digital lines for TDS/laptop synchronization

Inter–TDS synchronization

In normal operation (i.e. when no ECU has been declared faulty), TDS_2 takes the role of time master, while TDS_1 is a time slave. Therefore TDS_2 triggers all events which need to happen synchronously on both TDS boards. In particular the `MAIN` word executed on TDS_2 (i.e. execution of the Execution Table: see 3.6.3) is triggered at a fixed frequency by an interrupt from one of TDS_2's internal timers.

Table 4.6 describes how the synchronization lines are used by TDS_2 to trigger TDS_1's loop–closing. The table is organized in a linear manner, from top to bottom. All events described on a given row are synchronous.

TDS2		TDS1
(interrupt from internal timer triggers the loop–closing)		[POLL PortB(1), t_{m1} μs *(wait for loop–closing signal)*]
PULL PortA(3) HIGH *(trigger loop–closing)*	$\longrightarrow$	
POLL PortB(1), t_{m2} μs *(wait for ACK)*		
	$\longleftarrow$	PULL PortA(3) HIGH *(send ACK)*

Table 4.6: Loop–closing synchronization on the TDS boards

As shown in Table 4.6, when the loop starts on TDS_2, its first action (part of the first step of the execution flow as defined in 3.6.3) is to pull a digital line high. At this moment, TDS_1 is in a polling state, waiting for this line to go high (it actually entered the polling state during the last step of the previous loop). When this effectively happens, TDS_1 exits the infinite polling loop and pulls another digital line high to acknowledge having received the

loop–starting signal. Only once this has been completed and TDS_2 has received the acknowledgement do the acquisitions and the rest of the loop take place. In Table 4.6, t_{m1} and t_{m2} represent the maximum polling times respectively on TDS_1 and TDS_2. They are chosen in a manner that ensures that they allow enough time for the corresponding digital lines to be pulled high under the worst–case execution time conditions. This allows the Distributed Error Detection utility to be certain that the exceeding of the polling time limit is the result of a time fault.

The approach described in the specific case of loop–closing triggering can obviously be applied to any other points in the execution where inter–TDS synchronization is required. The next subsection describes the way in which synchronization between the laptop and TDS boards is achieved. This makes use of inter–TDS synchronization.

Synchronization with the laptop

While execution of the control loop is triggered in a synchronous way at a fixed frequency on the TDS boards, as described above, execution of the control loop on the laptop is done in a continuous manner: when the application is started on the laptop, it enters an infinite loop, which executes the control loop (i.e. as defined in the Execution Table) repetitively. This results in an asynchronous operation of the laptop with regard to the TDS boards. However, a way of synchronizing the TDS boards and laptop is required for communication to be possible between these two types of platforms. Table 4.7 details how this synchronization is achieved.

As shown in Table 4.7, when the laptop reaches a point in its execution flow where it is ready to interact with the TDS boards, it pulls a digital line high (`DOUT(0)`), and enters a polling state where it waits for both TDS boards to acknowledge that it is ready. At this instant, the step at which the TDS boards are in their own execution flow is not important. At some point in the

TDS_2		Laptop		TDS_1
		...		
	←	PULL DOUT(0) HIGH *(indicate that laptop is ready)*	→	
...		POLL DIN(0) and DIN(1) $t_{m,LT1}$ and $t_{m,LT2}$ μs *(wait for ACK from both TDS boards)*		...
				POLL PortB(2) t_{m1} μs *(wait for laptop check trigger)*
PULL PortA(4) HIGH *(trigger laptop check)*	→		→	
POLL PortB(2) t_{m2} μs *(wait for ACK)*				
	←		←	PULL PortA(4) HIGH *(send laptop check ACK)*
CHECK Port7(0) *(check if laptop is ready)*				CHECK Port7(0) *(check if laptop is ready)*
PULL PortA(7) HIGH *(acknowledge that the laptop is ready)*	→		←	PULL PortA(7) HIGH *(acknowledge that the laptop is ready)*
...		...		...

Table 4.7: Synchronization of the laptop with the TDS boards

TDS boards' execution flow, an inter–TDS synchronization similar to the one described in the previous subsection allows TDS_2 to trigger a synchronized check of the digital line controlled by `DOUT(0)`. If, as in the example of Table 4.7, the line is high, then both TDS boards send an acknowledgement to the laptop and synchronized laptop/TDS communication can then be conducted. If the line is low, then the TDS boards continue their execution flow in 2–node mode, and they will check again if the laptop is ready during the next loop.

This technique allows the TDS boards' loop–closing frequency to be set to an integer multiple of the laptop loop–closing frequency[20].

As in inter–TDS synchronization, the laptop only polls for a limited time. A Forth word `2POLL` was written, which allows the laptop to wait for 2 digital lines to go from low to high level. `2POLL` takes 4 arguments: the number of both lines to be polled, the time limit for any of these lines to go high (*primary* time limit) and the time limit for the second line to go high once the first one has (*secondary* time limit). By using this word it is possible to specify a large primary time limit which allows one to synchronize the laptop and TDS boards (the worst–case time offset before synchronization is one TDS loop–closing period), and specify a short secondary time limit used as a means to ensure that both TDS boards are properly synchronized, and therefore to detect if the second TDS board is faulty in the time domain.

Communication synchronization

In order to ensure communication determinism, all communications are:

- either initiated by the time master,
- or initiated after the time master indicated that the CAN bus is available (by pulling a digital line high).

[20]In the described experimental system the laptop loop–closing frequency is half the TDS boards loop–closing frequency

Furthermore, the digital acknowledgement scheme used is as follows: after each successful completed transmission, the transmitting node pulls a digital line high to notify the receiving node that the bus is available. All communications are scheduled in advance, and the receiving node is thus waiting (polling state) for this notification to occur. As at all other synchronization points, exceeding of the polling time limit is interpreted by the Distributed Error Detection utility as a time error from the transmitting node.

Table 4.8 provides the example of an exchange of information between the two TDS boards using the above described scheme.

TDS2		TDS1
...		...
`Send: CANmsg7` *(send CAN message)*	$\Longrightarrow$	POLL PortB(0) t_{m1} μs *(wait for CANmsg sent notification)*
PULL PortA(1) HIGH *(notify sending of CAN message)*	$\longrightarrow$	
POLL PortB(0) t_{m2} μs *(wait for CANmsg sent notification)*	$\Longleftarrow$	`Send: CANmsg1` *(send CAN message)*
	$\longleftarrow$	PULL PortA(2) HIGH *(notify sending of CAN message)*
`Receive: CANmsg1` *(retrieve the received message from the CAN chip)*		`Receive: CANmsg7` *(retrieve the received message from the CAN chip)*
... *(rest of application)*		... *(rest of application)*

Table 4.8: Synchronized communication between the TDS boards

4.3.3 Conclusions

Synchronization is essential for two reasons:

- ensure determinism of the CAN communications by removing the need of arbitration,
- and provide check points for the detection of time errors.

Using the previously described synchronization scheme for communication, it takes 2.25 ms for one 8–byte message to be sent from a TDS board to the other[21], and 2.1 ms for one message to be sent from a TDS board to the laptop.

It should be noted that (at least dual) redundant communication channels are normally required in the fault–tolerance context. Implementation of this would require the development of software level utilities for controlling access to the redundant channels. In the work presented here, only a single CAN bus was used, and communication faults are therefore not tolerated by the resulting experimental system.

It also follows from the preceeding considerations that synchronization is only possible if the worst–case execution time of all Forth words used is accurately known. This is essential to efficiently set the maximum polling times.

4.4 Object–oriented model of the system: the "Virtual Steering" model

As described in 2.3.4, the Steer–by–Wire system is modeled in software using object–oriented programming. The resulting object–oriented model is called the **Virtual Steering** model. It is comprised of several objects, which map the physical hardware units into the software model. In the experimental system

[21] This includes the physical transmission, notification and acknowledgement as well as the use of the high–level methods Send: and Receive:

described here, the objects were therefore instantiated from the `Helm`, `Drive` and `DriveController` class definitions. The objects instantiated from the `Helm` and `Drive` classes interact directly with physical hardware (sensors and actuators). Therefore their constitutive methods are *atomic* methods[22]. On the other hand, the `DriveController` object is comprised of *high–level* methods.

This section describes the organization of the three aforementioned classes.

4.4.1 Helm and Drive classes

`Helm` and `Drive` are really similar classes: indeed they model units, which are comprised of 2 position sensors and one rotational velocity sensor. The Drive Unit also includes a DC motor, which can be commanded through a servo amplifier, therefore the `Drive` class is slightly bigger than the `Helm` class. Table 4.9 provides a detailed description of the `Drive` class (only the most important instance variables and methods are shown). Features specific to the `Drive` class are shown in bold characters[23].

The *Acquire_pot1:* and *Acquire_pot2:* methods

The `Acquire_pot1:` and `Acquire_pot2:` methods are used to trigger analog to digital conversion from the potentiometers analog signals, and convert the acquired digital values to signed 11–bit integers, therefore mapping the $\{-5, 5\text{turns}\}$ potentiometer position range into the $\{-1024, 1023\}$ digital angle range. The resulting instance variables are `angle1` and `angle2`.

The *Acquire_tach:* method

The tachometer signal is particularly noisy. This is due to the physical design of the sensor, which generates noise frequencies proportional to the shaft rotational velocity[11]. It is therefore necessary to filter this signal in order to reach an

[22] See 3.6.1

[23] All other instance variables and methods are identical in the Helm class

Instance Variables	
Name	Description
`angle1`	θ_1: measured angle from pot1
`angle2`	θ_2: measured angle from pot2
`softAngle`	θ_{soft}: angle interpolated from rotational velocity measurement
`rotVelk`	$\omega_d(T_k)$: measured rotational velocity at time T_k
`rotVelk1`	$\omega_d(T_{k-1})$
`anglek`	$\theta_d(T_k)$: agreed on angle at time T_k
`anglek1`	$\theta_d(T_{k-1})$
Vd	$V_{D,d}$: actuator voltage (computed by the DriveController object)
`accuracy(3)`	Array of 3 values characterizing comparative accuracy of the sensors

Methods		
Name	Stack Diagram	Description
`Acquire_pot1:`	$(- \theta_1)$	Acquires data from pot1 and updates `angle1`
`Acquire_pot2:`	$(- \theta_2)$	Acquires data from pot2 and updates `angle2`
`Acquire_tach:`	$(- \omega_d(T_k))$	Performs 5 successive measurements from the tach, computes the mean and updates `rotVelk`
`Process_tach:`	$(\omega_d(T_k) - \theta_{soft})$	Performs interpolation and updates `softAngle`
`Acquire_sensors:`	$(- \theta_{soft}\ \theta_2\ \theta_1)$	Performs all data acquisitions
Actuate:	$(V_{D,d} -)$	Actuates the Drive actuator

Table 4.9: The Drive class

acceptable level of accuracy. Because the noise frequency is tightly coupled to the shaft rotational velocity, simple analog low–pass filters (which have a fixed corner frequency) are poorly suited to this task. Digital adaptive filtering schemes can be implemented.

The solution adopted for the design of the experimental system was a simple averaging scheme: `Acquire_tach:` performs 5 successive acquisitions (analog to digital conversions) and averages them to output the tachometer reading. This is obviously the simplest possible digital filtering scheme, but it could be made more effective if required.

Determination of *softAngle*; the *Process_tach:* method

Position sensing redundancy is achieved through the use of 2 redundant potentiometers and a tachometer. A first order interpolation scheme is used to estimate the current digital position ($\theta_d(T_k)$) from the knowledge of the last iteration's correct digital position ($\theta_d(T_{k-1})$) and current and last iteration's measured digital rotational velocities (respectively $\omega_d({T_k}')$ and $\omega_d({T_{k-1}}')$), where T_k is the time when the position is acquired and ${T_k}'$ is the time when the rotational velocity is acquired (it is assumed that $T_{k-1} < {T_{k-1}}' < T_k$). The instance variable resulting from this integration is called `softAngle`, or θ_{soft} and the method in charge of this integration is `Process_tach:`.

The exact analog position can be integrated in the following manner in continuous time:

$$\theta_a(T_k) = \theta_a(T_{k-1}) + \int_{T_{k-1}}^{T_k} \dot{\theta}_a(t)dt \tag{4.5}$$

This is approximated in the discrete time domain, giving the estimated analog angle $\hat{\theta_a}$:

$$\hat{\theta_a}(T_k) = \theta_a(T_k) + ({T_k}' - {T_{k-1}}')\frac{\omega_a({T_k}') + \omega_a({T_{k-1}}')}{2} \tag{4.6}$$

It is then required to put equation 4.6 in digital form. This is done by considering the following:

$$\begin{cases} \theta_d &= K_{A/D,\theta} \cdot \theta_a \\ \omega_d &= K_{A/D,\omega} \cdot \omega_a \end{cases} \tag{4.7}$$

Equations 4.6 and 4.7 are combined to give Equ 4.8:

$$\hat{\theta_d}(T_k) = \theta_d(T_{k-1}) + \frac{K_{A/D,\theta}}{2K_{A/D,\omega}}\left(T_k{}' - T_{k-1}{}'\right)\left[\omega_d\left(T_k{}'\right) + \omega_d\left(T_{k-1}{}'\right)\right] \tag{4.8}$$

In the implementation described here the acquisition times were approximated in the following manner[24] :

$$\begin{aligned} T_k{}' &= T_k \\ T_k{}' - T_{k-1}{}' &= T_s \end{aligned} \tag{4.9}$$

Where T_s is the loop-closing period. Therefore the formula, which was used in the `Process_tach:` method is as follows:

$$\theta_{\text{soft}}(T_k) = \theta_d(T_{k-1}) + \frac{K_{A/D,\theta}}{2K_{A/D,\omega}} T_s\left[\omega_d(T_k) + \omega_d(T_{k-1})\right] \tag{4.10}$$

The different constants found in Equ 4.10 are:

	Laptop		TDS	
	Helm	Drive	Helm	Drive
$K_{A/D,\theta}\ [\text{rad}^{-1}]$	32.6	8.15	32.6	8.15
$K_{A/D,\omega}\ \left[\left(\text{rad}.s^{-1}\right)^{-1}\right]$	235.4	33.7	58.9	8.4

Table 4.10: Constants in Equ 4.10

[24]This is acceptable considering the small expected displacements between two consecutive acquisitions

The *accuracy* instance variable

As detailed in the description of the Local Error Detection utility[25], the outputs from diversely redundant analog sensors can only be compared within hardware dependent tolerance limits. These limits depend for instance on the accuracy and calibration of the sensors, which are intrinsic characteristics of the Helm and Drive Units, and should therefore be included in their object–oriented model, i.e. respectively `Drive` and `Helm` classes.

In the implementation described here, these "tolerance" characteristics are modelled by the `accuracy` instance variable. `accuracy` is an array of 3 values representing the maximum acceptable offset between sensors readings. `accuracy(1)` is the maximum allowable offset between `angle1` and `angle2`, `accuracy(2)` is the maximum allowable offset between `angle1` and `softAngle` and `accuracy(3)` is the maximum allowable offset between `angle2` and `softAngle`.

In the current implementation, the `accuracy` array is set in advance and is kept constant over the whole range of acquired positions. Further implementations could easily modify it in real–time to adapt it to the operating conditions (position, rotational velocity, etc).

The *Actuate:* method

The `Actuate:` method is used to send the velocity command to the Drive Unit's servo amplifier. This command is computed by the `DriveController` object[26]. The form in which it is sent to the actuator is greatly dependent on the platform considered (TDS_2 or laptop)[27]. In the TDS code, the command has first to be translated into an 8–bit value to be output on the PWM "analog" output channel, and a direction bit, to be output using a simple digital I/O line. This

[25] See 3.6.2

[26] See 4.4.2

[27] See Appendix A

gives an effective 9–bit precision for the commanded value on TDS_2.

On the laptop, the 12–bit value to be output is sent directly to the DAC.

Conclusions

Despite important differences in the actual source code, high–level use of the `Helm` and `Drive` classes respectively is identical on the TDS boards and on the laptop. The two classes are instantiated as the `HelmTDS` and `DriveTDS` objects, and `HelmLT` and `DriveLT` objects respectively on these two types of ECU's.

4.4.2 DriveController class

The `DriveController` class defines the controller which is used for position control of the Drive Unit. As indicated in 2.3.2, this controller is comprised of both a *nominal angle calculation* block and a *control strategy* block.

Nominal angle calculation

As was previously discussed in 2.3.2, it is possible to modify the reference angle sent to the controller so as to enhance the functionality and performance levels of the system. In the application as presented here, the nominal angle is simply calculated by multiplying the Helm angle by a constant gain K_{nA}. As detailed in 2.3.2, in more complex implementations K_{nA} could be modified at run–time by the user to allow switching from off–shore navigation to docking maneuvers, or it could be a function of the speed of the vessel. The implementation of such functionalities is out of the scope of this thesis.

Control strategy

The controller used is a "lead/lag" filter. It can be represented by the following transfer function in the Laplace domain:

$$H_{ll}(s) = K_{ll}\frac{s+a}{s+b} \tag{4.11}$$

Proper tuning of the lead/lag filter can bring great simplifications to the analysis of the system's behavior. The plant to be controlled (Drive Unit servo amplifier and DC motor) was shown to have a first–order transfer function in 4.2.1. The position control block diagram in the Laplace domain is therefore as shown in Fig 4.4.

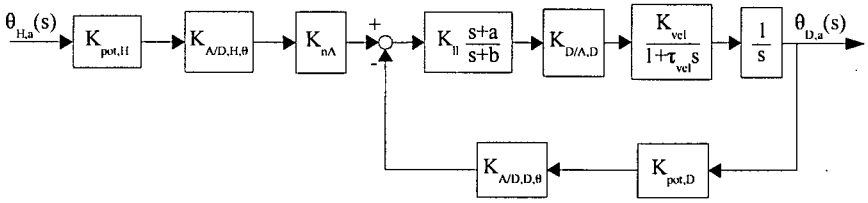

Figure 4.4: *Lead/lag filter in the Laplace domain (position control)*

The global transfer function $H(s) = \frac{\theta_{D,a}(s)}{\theta_{H,a}(s)}$ is obtained from Fig 4.4:

$$H(s) = \frac{\frac{K_{vel}\cdot K_{D/A,D}\cdot K_{ll}\cdot(s+a)\cdot K_{nA}\cdot K_{pot,H}\cdot K_{A/D,H,\theta}}{s(1+\tau_{vel}s)(s+b)}}{1+\frac{K_{vel}\cdot K_{D/A,D}\cdot K_{ll}\cdot(s+a)\cdot K_{pot,D}\cdot K_{A/D,D,\theta}}{s(1+\tau_{vel}s)(s+b)}} \tag{4.12}$$

Obviously, choosing $a = \frac{1}{\tau_{vel}}$ allows one to cancel the effect of τ_{vel}. Furthermore, it is possible to obtain a second order global transfer function characterized by its natural frequency ω_n and its damping ratio ζ, by selecting b and K_{ll} in the following manner[45]:

$$\begin{cases} b &= 2\zeta\omega_n \\ K_{ll} &= \dfrac{{\omega_n}^2\cdot\tau_{vel}}{K_{pot,D}\cdot K_{A/D,D,\theta}\cdot K_{D/A,D}\cdot K_{vel}} \end{cases} \tag{4.13}$$

This gives:

$$H(s) = K_{\text{global}}\frac{{\omega_n}^2}{s^2+2\zeta\omega_n s+{\omega_n}^2} \tag{4.14}$$

where:

$$K_{\text{global}} = K_{nA} \frac{K_{pot,H} \cdot K_{A/D,H,\theta}}{K_{pot,D} \cdot K_{A/D,D,\theta}} \tag{4.15}$$

This indicates that the steady–state ratio of displacements at the Drive and at the Helm is set by the chosen value of K_{nA}.

In order to adapt this analysis to discrete time (Z–transform) form, the following simple transformation is used, where T_s is the loop–closing period[13]:

$$s = \frac{z-1}{T_s} \tag{4.16}$$

This gives the following discrete expression for the controller[36]:

$$K_{ll} \frac{s+a}{s+b} = K_{ll} \frac{z-A}{z-B}, \text{ where } \begin{cases} A &= 1 - aT_s \\ B &= 1 - bT_s \end{cases} \tag{4.17}$$

Fig 4.5 shows the control diagram of the system, where the controller is expressed in the Z–domain. $G_0(s)$ is the transfer function of a zeroth–order hold.

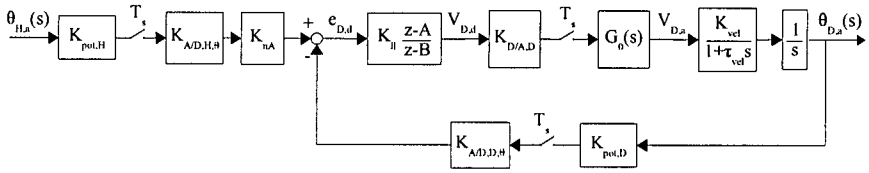

Figure 4.5: *Lead/lag filter in the Z–domain*

The following relationship is deduced from Equ 4.17:

$$V_{D,d}(T_k) = K_{ll} \frac{1 - Az^{-1}}{1 - Bz^{-1}} e_d(T_k) \tag{4.18}$$

From Equ 4.18 it is possible to obtain the digital controller in the form in which it is actually implemented in the `DriveController` class (`Lead_lag:` method):

$$V_{D,d}(T_k) = B \cdot V_{D,d}(T_{k-1}) + K_{ll} \cdot e_d(T_k) - A \cdot K_{ll} \cdot e_d(T_{k-1}) \qquad (4.19)$$

Table 4.11 provides a detailed description of the most important instance variables and methods implemented in the `DriveController` class. It should be noted that additional instance variables such as τ_{vel}, K_{vel}, $K_{A/D,H,\theta}$, A, B, K_{ll}, etc are also included in `DriveController`. For the sake of simplicity they are not shown in Table 4.11.

The `DriveController` class is implemented on the TDS boards and laptop as the `DCTDS` and `DCLT` objects respectively. The controller is tuned to predefined target values of ω_n and ζ (`ClassInit:` method), but it can be re–tuned off–line at any time for different natural frequency and damping ratio using the `Tune_leadlag:` method. This can be used to adapt the handling characteristics of the steering system to the driver's preferences.

4.5 Fault–tolerance utilities

In order to achieve the desired fault–tolerance property of the system, the general architecture detailed in 3.6.1 was implemented. This section describes the software implementation of the Redundancy Manager and of the Execution Table, as well as the hardware implementation of the statelines which are used for hardware reconfiguration.

4.5.1 Implementation of the Redundancy Manager

The Redundancy Manager, as described in 3.6.2, was partially implemented in the experimental system. The use of redundant sensors justified the implementation of the various error detection utilities. On the other hand, in the absence of redundant actuators, the Actuation Manager was not needed and thus not

Instance Variables	
Name	Description
`ek`	$e_d(T_k)$: error at time T_k
`ek1`	$e_d(T_{k-1})$
`Vdk`	$V_{D,d}(T_k)$: applied "digital voltage" at time T_k
`Vdk1`	$V_{D,d}(T_{k-1})$
`omega`	ω_n: natural frequency of the overall second order transfer function
`zeta`	ζ: damping ratio of the overall second order transfer function

Methods		
Name	Stack Diagram	Description
`ClassInit:`	*(—)*	Initializes the object and tunes the lead/lag filter
`Tune_leadlag:`	*(—)*	Tunes the lead/lag filter according to the values of ω_n and ζ
`Compute_nA:`	*(θ_{in} — $\theta_{nA,d}$)*	Computes the nominal angle
`Compute_errors:`	*($\theta_{nA,d}(T_k)$ $\theta_{out}(T_k)$ — $e_d(T_k)$)*	Computes $e_d(T_k)$ and updates $e_d(T_{k-1})$
`Lead_lag:`	*($e_d(T_k)$ — $V_{D,d}(T_k)$)*	Performs the lead/lag filter action, and updates $V_{D,d}(T_k)$ and $V_{D,d}(T_{k-1})$
`Perform_control:`	*($\theta_{in}(T_k)$ $\theta_{out}(T_k)$ — $V_{D,d}(T_k)$)*	Performs all control actions

Table 4.11: The DriveController class

implemented[28].

The LocalErrorDetection class

The objects representing the Local Error Detection utilities are instantiated from the `LocalErrorDetection` class. This class contains all the features described in 3.6.2 and is instantiated as `HelmLED` and `DriveLED` for local error detection of the Helm and Drive units respectively. Its most important instance variables and methods are detailed in Table 4.12.

As shown in Table 4.12, a method performing executable assertions on the redundant sensors readings is implemented (`E_A:`). The simple checks conducted verify that the acquired measurements are within the sensors physical limits, and that they differ from the last correct measurement by less than the maximum allowable offset. A method performing data replication tests is also provided (`D_R:`). This latter method uses the `accuracy` instance variable of the considered unit (Helm or Drive), which it obtains from the `Helm` or `Drive` object as described in 4.4.1. Lastly a single method is provided, which performs both executable assertions and data replication tests, and outputs the estimated correct angle along with the Measurement State vector MS, expressed as a 3–bit integer (`Detect_errors:`). This method is meant to be the only one called by the `MAIN` word, as defined in the Execution Table[29]. Note that because majority voting is used, it is not possible that only one measurement is declared correct. If however the sum of the bits of MS is 1, then this indicates that a computing error occured, and MS is then updated to 0 before it is output by `Detect_errors:`. If a higher–level error detection utility receives a measurement state vector MS for which the sum of the bits is 1, it can therefore conclude that the ECU which performed the considered local error detection is faulty.

[28]This results in the impossibility for the system to tolerate actuator faults

[29]See 3.6.3

Instance Variables	
Name	Description
`angles(3)`	Array of 3 angles to be compared
`MS(3)`	Measurement state vector
`DR_ind(3)`	Data replication indicator
`anglek`	$\theta_e(T_k)$: estimated correct angle at time T_k
`anglek1`	$\theta_e(T_{k-1})$

Methods		
Name	Stack Diagram	Description
`ClassInit:`	*(n —)*	Initializes the object respectively as Helm or Drive Local Error Detection if n is 0 or 1
`OutOfRange:`	*(u — flag)*	Outputs a flag indicating whether the input lies within the acceptable range or not
`E_A:`	*(—)*	Performs `OutOfRange:` for each redundant measurement and updates `MS` (Executable Assertions)
`D_R:`	*(—)*	Compares the acquired values 2 by 2 and updates `DR_ind` (Data Replication)
`Compute_angle:`	*(—)*	Computes the partial average of the correct measured angles
`Detect_errors:`	*(θ_3 θ_2 θ_1 — $\theta_e(T_k)$ MS)*	Performs all local error detection actions

Table 4.12: The LocalErrorDetection class

The 2NodeErrorDetection class

In certain cases as mentioned in 3.6.2, only two nodes are available for distributed error detection. Specifically in the experimental system described here, it is the case:

- at every loop where the laptop does not interact with the TDS boards,
- and when the system is downgraded to 2–Node mode, i.e. when one of the ECU's is declared faulty.

In such cases the Distributed Error Detection utility used is instantiated from the `2NodeErrorDetection` class. This class contains two Value ECU Error Detection utilities (`Helm2ED` and `Drive2ED`), which are themselves instantiated as instance variables from the `2NodeValueErrorDetection` class. As described in 3.6.2, since it is not possible to use majority voting algorithms, a look–up table is used by `Helm2ED` and `Drive2ED` to approximate the ECU state, sensor state and link state vectors from the knowledge of the Local Error Detection utilities' results, and to determine the correct angle for the considered unit. A description of the most important instance variables and methods of the aforementioned classes is provided in Appendix B. The detail of the look–up table used is also provided (see Table B.3).

The 3NodeErrorDetection class

The Distributed Error Detection utility used in normal operation is instantiated from the `3NodeErrorDetection` class. This class has been created following the guidelines described in 3.6.2; in particular it contains two Value ECU Error Detection utilities `Helm3ED` and `Drive3ED`, which are instantiated from the `3NodeValueErrorDetection` class. Within `Helm3ED` and `Drive3ED`, majority voting is used to compare the estimated correct angles from each ECU's considered unit's Local Error Detection utility. From these comparisons, each unit's

correct angle is determined using partial averaging, and the estimated value ECU state vectors $ES_{v,e}$ are computed.

The time domain ECU state vector is included in the `3NodeErrorDetection` class as an instance variable. It can be updated using the `Time_fault:` method at any time, i.e. whenever a missed deadline is detected through the synchronization mechanism. The value domain and time domain ECU state vectors are then combined by the `Compute_ES:` method to update the ECU state vector ES.

A method is also provided (`Pull_SL:`), to directly pull any stateline high following the detection of an error on the corresponding ECU[30]. Table B.4 and Table B.5 detail `3NodeErrorDetection` and `3NodeValueErrorDetection`'s most important instance variables and methods. In the current version of the system, the comparison of the Local Error Detection utilities' Measurement State vectors to obtain the Sensor and Link State vectors is not implemented.

4.5.2 Implementation of the Execution Table

As described in 3.6.3, on each ECU the execution flow is divided into steps, and for each step the code to be executed (if any) is to be chosen among a selection of Maximum Size Bundles. This choice is made according to the state of the system's ECU's. The system is in one of three possible mode:

- 3–Node mode: this is true when all ECU's are non–faulty, and the laptop is available for interaction with the TDS boards
- 2–Node mode: this is true either when all ECU's are non–faulty but the laptop is not available for interaction with the TDS boards, or when one of the ECU's is declared faulty
- 1–Node mode: this is true if both other ECU's are declared faulty.

[30]Note that it is also possible to pull a stateline low, for instance after a transient error is terminated

At any time that the system goes from one "mode" to another, the path followed by the execution flow inside the Execution Table is dynamically modified. Table 4.13 provides TDS_2's Execution Table as an example.

Execution Table		
1	`MSB1a`: Trigger loop–closing locally and on TDS_1 (**synch** with TDS_1)	`MSB1b`: Trigger loop–closing locally
2	`MSB2a`: Acquire all sensors Perform Local Error Detection	
3	`MSBS3a`: Send LocalED results (**synch** with TDS_1)	`MSB3b`: Send LocalED results (no synch)
4	`MSB4a`: Receive LocalED results from TDS_1 (**synch** with TDS_1)	
5	`MSB5a`: Check if laptop is ready	
6	`MSB6a`: Receive LocalED results from laptop (**synch** with laptop)	
7	`MSB7a`: Perform 3–Node Value Error Detection	`MSB7b`: Perform 2–Node Value Error Detection
8	`MSB8a`: Perform control Send actuation command to the Drive Unit	

Table 4.13: Execution Table for TDS_2

All Maximum Size Bundles from Table 4.13 which contain the "**synch**" mention update the Distributed Error Detection's time ECU state vector ES_t in case that a time error is detected.

In the specific example of Table 4.13, if the system is in 3–Node mode, then all Maximum Size Bundles of the first column are successively executed.

If all ECU's are non–faulty but the laptop is not available at the considered loop, the followed path is:

`MSB1a`

`MSB2a`

`MSB3a`

`MSB4a`

`MSB5a`

`MSB7b`

`MSB8a`

In a similar manner, for each possible mode, all possible combinations of ECU states are considered, and the execution flow path is pre–determined. In the example provided, `MSB1a`, `MSB3a`, `MSB4a` and `MSB6a` have the ability to dynamically modify the execution flow after the detection of an ECU time error, `MSB7a` and `MSB7b` have the ability to dynamically modify the execution flow after the detection of an ECU value error, and `MSB5a` has the ability to modify the execution flow depending on the availability of the laptop[31]. Appendix C provides the Execution Table used on TDS_1 and the laptop.

As indicated in 3.6.3, the Execution Table is implemented as an object. The `ExecutionTable` class contains the table itself, and all necessary methods to modify and execute it. This class is detailed in appendix, in Table C.3.

4.5.3 Implementation of the statelines

In the current version of the experimental system, only TDS_2's stateline (SL_2) is implemented. The open–collector line described in Fig 3.10 is emulated with a simple AND gate, which is accessed by both TDS_1 and the laptop. As mentioned in 4.5.1, each ECU can write a logical high value to the other ECU's statelines by

[31] In this last case the Execution Table is not used for error treatment but to simplify the source code, as indicated in 3.6.4

using `3NodeErrorDetection`'s `Pull_SL:` method. In the implemented system the only way for SL_2 to go high is therefore if both TDS_1 and the laptop write a logical high value to it.

SL_2's level is then used to command the switch which directs the actuation command from either TDS_2 or the laptop to the Drive Unit's servo amplifier and DC motor[32]. This technique then dictates that, if TDS_2 is declared faulty by both TDS_1 and the laptop, the actuation command source is automatically switched from TDS_2 to the laptop.

4.6 Results achieved and limitations

On the current version of the experimental system, the sampling frequency is 20Hz on the TDS boards, and 10Hz on the laptop (the laptop is synchronized with the TDS boards as described in 4.3.2, therefore its sampling time is twice greater than the TDS boards'). The factor limiting the maximum possible sampling frequency on the TDS boards is the execution speed. The sampling time could therefore be decreased by optimizing the source code, e.g. by coding the most often used Forth words in assembler, so that the program can run at close to machine code speed[42]. The factor limiting the maximum possible sampling frequency on the laptop is the DAC time. This is due to the data acquisition card used, which would not be a normal choice for a commercial Steer–by–Wire system. However, as described in 4.2.1, use of a third ECU at a slower sampling frequency primarily for error detection and fault–tolerance purposes constitutes a reasonable system design choice. In the manual testing done in the laboratory, at the sampling frequencies described above, the behavior of the system from a controls point of view was satisfactory. However, specific control testing was not performed, since the emphasis was put on the implementation of fault–tolerance rather than of the controller.

[32]This switch is shown in Fig A.4

At both units (Helm and Drive), a single sensor failure is tolerated. This has been verified by performing simple tests, such as removing the power to one of the potentiometers. The system is able to detect the sensor error, and to provide continued functionality. Both permanent and transient sensor faults are tolerated.

The ECU error detection was tested. Permanent time errors were simulated by stopping the application on one of the ECU's. The remaining two ECU's were able to detect the error and dynamically modify their Execution Tables to switch to 2–Node mode. When the application was stopped on TDS_2, the stateline mechanism allowed the actuation command to be sent by the laptop, and continued functionality was provided. ECU value error detection was not tested, because the injection of value faults is more difficult to achieve. Note that in the current version of the experimental system, all ECU errors are treated as permanent. No mechanism has been implemented to re–integrate into the application an ECU which has been previously declared faulty.

Utilities to tolerate communication errors were not implemented, since a single communication channel was used. Therefore a faulty communication channel is not tolerated by the current version of the system[33]. Similarly, redundant actuators and power supplies were not used, and thus actuator faults and power failures are currently not tolerated by the system.

4.7 Summary

In this chapter the concepts detailed in Chapter 2 and Chapter 3 are illustrated by the description of a laboratory experimental system which was designed and built as part of this thesis. This system is a marine Steer–by–Wire prototype,

[33]Note that transmission value errors caused by EMI for example are tolerated by CAN's integrated CRC algorithm. However, CAN's automatic error treatment scheme involves that the message is re–sent, and can therefore cause an excessively high transmission time, which will be interpreted as a time error

which implements closed–loop position control. It is comprised of a Helm Unit and a Drive Unit, and a distributed computing platform.

The Helm Unit provides the input to the system, and contains triple redundant position sensors. The Drive Unit contains triple redundant position sensors and a DC motor. The three redundant ECU's constituting the distributed computing platform are two identical single–board H8–based computers and an Intel II based laptop. On all ECU's ANS Forth is used with an object–oriented extension.

Communication between the ECU's is done using CAN. In order to avoid the non–determinism shown to be associated with CAN's arbitration mechanism[34], a synchronization scheme was developed, which ensures that no two nodes request use of the bus at the same time. This synchronization scheme can also be used at any point in the execution flow where synchronization between the ECU's is required.

The Virtual Steering object–oriented model mentioned in 2.3.4 is detailed for the specific case of the experimental system described here. It is comprised of the `Helm`, `Drive` and `DriveController` classes.

The partial implementation of the architecture described in 3.6 to provide cost–effective fault–tolerance is detailed. Local and Distributed Error Detection utilities, and Actuation Tables are described. The implementation of a single stateline and its use to dynamically modify the actuation command source of the Drive Unit is also detailed.

Lastly, the level of performance and fault–tolerance of the current version of the experimental system is assessed, and its limitations are indicated. In particular it is shown that single sensor faults of any type are tolerated at each unit, and that one permanently faulty ECU is tolerated.

[34] See 3.4.1

Chapter 5

Conclusions and recommendations for future work

5.1 Conclusions

The use of By–Wire systems in car or pleasure boat applications is expected to be wide–spread in the not–too–distance future. The functionality, safety and performance improvements which are associated with these systems make them desirable components of tomorrow's land and water personal vehicles. However, to gain public acceptance and meet stringent safety requirements while adapting such systems to their cost–constrained markets, it is necessary to define an architecture which guarantees a high level of fault–tolerance at an optimal cost.

Approaches developed by other research groups to attain these goals have been based on a high redundancy level: the Boeing 777 controller is comprised of three triple redundant ECU's, and the automobile Steer-by-Wire prototype implemented as part of the X-by-Wire project is controlled by three fault-tolerant nodes, each of which is locally comprised of four ECU's.

On the other hand, the architecture presented in this thesis contains a variety of novel features which result in a considerable reduction of the required redundancy level.

- It combines Local Error Detection, Distributed Error Detection and Actuation Manager utilities to provide sensor, actuator and ECU error detection. In addition to their error detection role, these utilities manage the redundancy of the system by ensuring that the high–level components of the object–oriented software system view each redundant sensing or actuating unit as a single sensor or actuator. Sensor errors are handled by the Local Error Detection utility through the use of a partial averaging scheme.

- The Execution Table is used to dynamically modify the execution flow of a given ECU after, for example, it has detected that one of the other ECU's is faulty. This results in the property that is termed dynamic reconfiguration. Using this mechanism, it has been shown that it is possible for the non–faulty ECU's to adapt their behavior to the new system configuration by, for instance stopping all communication attempts with the faulty node, or by using a downgraded Distributed Error Detection utility.

- Distributed ECU diagnosis consensus is obtained in an extremely efficient way through the use of a stateline–based architecture. Using this approach it is possible to automatically isolate a faulty node from the rest of the network, or to automatically switch between potential actuation command sources, and to dramatically reduce the communication and computation bandwidth associated with the consensus–making process.

A marine Steer–by–Wire system is a typical By–Wire system, and can therefore be used to illustrate the concepts and architecture described above. An experimental prototype of such a system was designed and built as part of this thesis. Sensor and ECU triple redundancy were used, and the utilities described above implemented. Simple tests have shown that a single transient or permanent sensor fault is tolerated at each redundant sensing unit, and that one faulty ECU is tolerated by the system. It is important to note that the above–described

architecture has allowed the global ECU redundancy level over the whole network to be reduced to three, to compare to the levels of nine and twelve used in the particular cases of the Boeing 777 control system and X-by-Wire project's Steer-by-Wire prototype respectively.

5.2 Recommendations for future work

Communication fault–tolerance was considered as a pre–requirement in the definition of the proposed supervisory architecture. Communication channels are obviously not intrinsically fault–tolerant. It is therefore required that dual redundant channels are used, and that utilities are developed, which manage the communication redundancy at each node.

In the current implementation of the system, transient ECU errors are considered as permanent, and result in the isolation of the considered ECU from the rest of the system. This obviously implies a non–optimal use of the available resources. Therefore means need to be developed to allow an ECU which was declared faulty because of a transient fault to be re–incorporated in the application once this transient fault has disappeared.

There is a general trend towards reduced wiring in automotive and marine applications. It should be noted that the number of parallel digital lines required by the synchronization scheme re–introduces extra wiring, which goes against this trend. This also introduces a multitude of fault sources into the system. The development of a serial protocol for synchronization is foreseen as a solution to these problems.

Bibliography

[1] ANSI X3.215-1994. Programming Languages - Forth, March 1994.

[2] Bannatyne, R. The Sensor Explosion and Automotive Control Systems. *Sensors online*, May 2000. `http://www.sensorsmag.com/articles/0500/92/main.shtml` [accessed on 02/15/2001].

[3] Barron, J. Fly by Wire. *Bass & Walleye Boats, the Magazine of Performance Fishing Boats*, August 1998.

[4] Brodie, L. *Starting Forth.* Prentice Hall, second edition, 1987.

[5] CAN Protocol. Published on the internet. `http://www.csn.ul.ie/~ado/docs/CAN/CAN.html` [accessed on 02/06/2001].

[6] Claesson, V. Prototype Implementation using the XBW Software Model. In *National Swedish Conference on Real-Time Systems, SNART*, 1997.

[7] Cetrek Ltd. 930609/619 Pilot Computer Installation Guide, Issue 06. Published on the internet. `http://www.cetrek.co.uk/Manuals/609-619.pdf` [accessed on 06/20/2001].

[8] Claesson, V., S. Poledna, and J. Söderberg. The XBW Model for Dependable Real-Time Systems. In *Proceedings of the International Conference on Parallel and Distributed Systems*, pages 130–138, 1998.

[9] Conklin, E. K. and E. D. Rather. *Forth Programmer's Handbook.* Forth Inc., second edition, 1998.

[10] Cooling, J. E. *Software Design for Real-Time Systems.* Cornwall: Chapman and Hall, 1991.

[11] de Silva, C. W. *Control Sensors and Actuators.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[12] Dolev, D., N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching Approximate Agreement in the Presence of Faults. In *Journal of the Association for Computing Machinery*, volume 33, pages 499–516, July 1986.

[13] Dutton, K., S. Thompson, and B. Barraclough. *The Art of Control Engineering.* Addison-Wesley, Harlow, England, 1997.

[14] Forth Inc. *SwiftForth Development System for Windows: Reference Manual.* Second edition, 1998.

[15] Frank, P. M. Advanced Fault Diagnosis Techniques in Aerospace Systems. In *Proceedings of CompEuro '89, 'VLSI and Computer Peripherals. VLSI and Microelectronic Applications in Intelligent Peripherals and their Interconnection Networks'*, pages 3/136–3/143, 1989.

[16] Fredriksson, L.-B. CAN for Critical Embedded Automotive Networks. *IEEE Micro*, pages 28–35, July-August 2002.

[17] Führer, T., B. Müller, W. Dieterle, F. Hartwick, R. Hugel, and M. Walther (Robert Bosch GmbH). Time Triggered Communication on CAN (Time Triggered CAN-TTCAN). In *Proceedings of the 7th International CAN Conference, Amsterdam*, 2000.

[18] Hartwick, F., B. Müller, T. Führer, and R. Hugel (Robert Bosch GmBH). Timing in the TTCAN Network. In *Proceedings of the 8th International CAN Conference; Las Vegas, Nv*, 2002.

[19] Heiner, G. and T. Thurner. Time-Triggered Architecture for Safety-Related Distributed Real-Time Systems in Transportation Systems. In *Digest of Papers on the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pages 402–407, 1998.

[20] Herlihy, M. P. and J. M. Wing. Specifying Graceful Degradation. In *IEEE Transactions on Parallel and Distributed Systems*, volume 2, pages 93–104, January 1991.

[21] Hiller, M. Using Software to Handle Data Errors in Embedded Control Systems. Licentiate of Engineering Thesis, Chalmers University of Technology, 1999.

[22] Hitachi. Hitachi Single-chip Microcomputer H8/532 Hardware Manual.

[23] Intel Corporation. 82527 Serial Communications Controller Architectural Overview, January 1996.

[24] Kimura, S. and T. Okuyama. Consensus Making of Multi-CPU Control of Hyper-redundant Manipulator. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 4, pages 3623–3628, 1998.

[25] Koopman, P. Critical Embedded Automotive Networks. *IEEE Micro*, July-August 2002.

[26] Kopetz, H. *Design Principles for Distributed Embedded Applications.* Kluwer Academic Publisher, 1997.

[27] Kopetz, H. and G. Grünsteidl. TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems. In *Digest of Papers, The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 524–533, August 1993.

[28] Kurishige, M., T. Kifugu, N. Inoue, S. Zeniya, and S. Otagaki. A Control Strategy to Reduce Steering Torque for Stationary Vehicles Equipped with EPS. *SAE Technical Papers*, 1999.

[29] Lamport, L., R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[30] Laprie, J. C. Dependability - Its Attributes, Impairments and Means. In *Predictably Dependable Computing Systems*, pages 3–24. Springer-Verlag, Berlin; Heidelberg; New York, 1995.

[31] Maier, R., G. Bauer, G. Stöger, and S. Poledna. Time-Triggered Architecture: a Consistent Computing Platform. In *IEEE Micro*, volume 22, pages 36–45, July-August 2002.

[32] McKewan, A. Object-Oriented Programming in ANS Forth. In *Forth Dimensions*, pages 14–29, March-April 1997.

[33] Müller, B., T. Führer, F. Hartwick, R. Hugel, and H. Weiler (Robert Bosch GmBH). Fault Tolerant TTCAN Networks. In *Proceedings of the 8th International CAN Conference; Las Vegas, Nv*, 2002.

[34] Nace, W. and P. Koopman. A Product Family Approach to Graceful Degradation. In *Proceedings of DIPES 2000, International IFIP WG 10.3 / WG 10.4 / WG 10.5 Workshop on, Distributed and Parallel Embedded Systems*, October 2000.

[35] Nace, W. and P. Koopman. Graceful Degradation in Distributed Embedded Systems. *Dr. Dobb's Embedded Systems*, 2001. `http://www.ddjembedded.com/ressources/articles/2001/0106em001/0106em001a.htm` [accessed on 08/30/2001].

[36] Oldknow, K. D. A Dynamically Reconfigurable System Architecture and FPGA Based Servo Controller for Distributed Machine Tool Control. MASc Thesis, University of British Columbia, 2000.

[37] PÅLBUS team. PÅLBUS - Validation of Dependable Distributed Computing Systems. Published on the internet, 2001. `http://www.sp.se/electronics/rnd/palbus/` [accessed on 06/14/2001].

[38] Peter, D. and R. Gerhard. Electric Power Steering - The First Step on the Way to Steer by Wire. *SAE Technical Papers*, 1999.

[39] Pflanz, M., F. Pompsch, and H. T. Vierhaus. An Efficient On-line-Test and Back-up Scheme for Embedded Processors. In *Proceedings of the International Test Conference*, pages 964–972, 1999.

[40] Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. ISO 11898, 1993.

[41] Robert Bosch GmBH. CAN Specification; Version 2.0, September 1991.

[42] Rush, P. *TDS2020F Technical Manual.* Triangle Digital Services Ltd, Harlow, England, 2000.

[43] SAE. Class C Application Requirements-J2056/1, /2. In *SAE Handbook*, pages 23366–23390. SAE Press, Warrendale, PA, 1994.

[44] Sandström, K., C. Eriksson, and G. Fohler. Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System. In *Proceedings of the Fifth International Conference on Real-Time Computing Systems and Applications*, pages 158–165, 1998.

[45] Seethaler, R. and I. Yellowley. The Regulation of Position Error in Contouring Systems. *International Journal of Machine Tools Manufacture*, 1996.

[46] Shelton, C. P. and P. Koopman. Developing a Software Architecture for Graceful Degradation in an Elevator Control System. In *Workshop on Reliability in Embedded Systems (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001), New Orleans, LA*, October 2001.

[47] Shimizu, Y., T. Kawai, and J. Yuzuriha. Improvement in Driver-Vehicle System Performance by Varying Steering Gain with Vehicle Speed and Steering Angle: VGS (Variable Gear-ratio Steering System). *SAE Technical Papers*, 1999.

[48] Sogomonyan, E. S. and M. Gssel. Concurrently Self-Testing Embedded Checkers for Ultra-Reliable Fault-Tolerant Systems. In *Proceedings of the 14th VLSI Test Symposium*, pages 138–144, 1996.

[49] Tarnick, S. and A. P. Stroele. Embedded Self-Testing Checkers for Low-Cost Arithmetic Codes. In *Proceedings of the International Test Conference*, pages 514–523, 1998.

[50] Teleflex Inc. Steering: Full Line Catalog and Tech Reference. Published on the internet. `http://www.teleflexmorse.com/scripts/PDF/steering.pdf` [accessed on 06/15/2001].

[51] Trotter, J. and T. A. Varvarigou. Distributed Reconfiguration of Multiprocessor Systems. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 212–221, 1994.

[52] Trotter, J. and T. A. Varvarigou. Reconfiguring Multiprocessor Systems while Minimizing Disturbance. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 122–131, 1995.

[53] Walter, C. J., P. Lincoln, and N. Suri. Formally Verified On-Line Diagnosis. In *IEEE Transactions on Software Engineering*, volume 23, pages 684–721, November 1997.

[54] Werninck, E. H. *Electric Motor Handbook.* McGraw-Hill, UK, 1978.

[55] Wills, L., S. Kannan, S. Sander, M. Guler, B. Heck, J. V. R. Prasad, D. Schrage, and G. Vachtsevanos. An Open Platform for Reconfigurable Control. In *IEEE Control Systems Magazine*, pages 49–64, June 2001.

[56] Woehr, J. *Forth: The New Model.* M&T Books, San Mateo, 1992.

[57] Murray Woodside, C. and G. G. Monforton. Fast Allocation of Processes in Distributed and Parallel Systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4, pages 164–174, February 1993.

[58] X-By-Wire project's website. `http://www.vmars.tuwien.ac.at/projects/xbywire/` [accessed 05/09/2001].

[59] X-By-Wire team. X-By-Wire, Safety Related Fault Tolerant Systems in Vehicles, Final Report. Published on the internet, 1998. `http://www.vmars.tuwien.ac.at/projects/xbywire/docs/final.doc` [accessed on 05/09/2001].

[60] Yeh, Y. C. Triple-Triple Redundant 777 Primary Flight Computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, volume 1, pages 293–307, 1996.

[61] Yeh, Y. C. Design Considerations in Boeing 777 Fly-By-Wire Computers. In *Proceedings of the Third IEEE International High-Assurance Systems Engineering Symposium*, pages 64–72, 1998.

[62] Yellowley, I. and P. R. Pottier. The Integration of Process and Geometry within an Open Architecture Machine Tool Controller. *International Journal of Machine Tools and Manufacture*, 1994.

Appendix A

Treatment of analog signals

The potentiometer signals range from 0 to 5V, as detailed in 4.2.1. They can therefore be directly fed to the computers' ADCs. On the other hand, the tachometers' signals need to be modified before they are sent to the computers' ADCs.

The analog circuitry described in the following subsections has been implemented on the development board shown in Fig A.1.

A.1 Modification of the tachometers signals for TDS input

The TDS ADC can only convert analog signals ranging from 0 to 5V. By definition, a tachometer's output is proportional to the rotational velocity, and therefore it can take on negative values. It is thus necessary to shift this signal so that it eventually lies within the $\{0, 5V\}$ range. Furthermore, the expected rotational velocities are low (especially at the Helm Unit), hence amplification of the tachometer signals is also required.

For this purpose, the circuit shown in Fig A.2 has been implemented, for both the Helm Unit's and the Drive Unit's tachometer signals.

This circuit is comprised of a follower, a summing amplifier used as range shifter and an inverter (inverting amplifier of gain 1). Equ A.1 is obtained from Fig A.2.

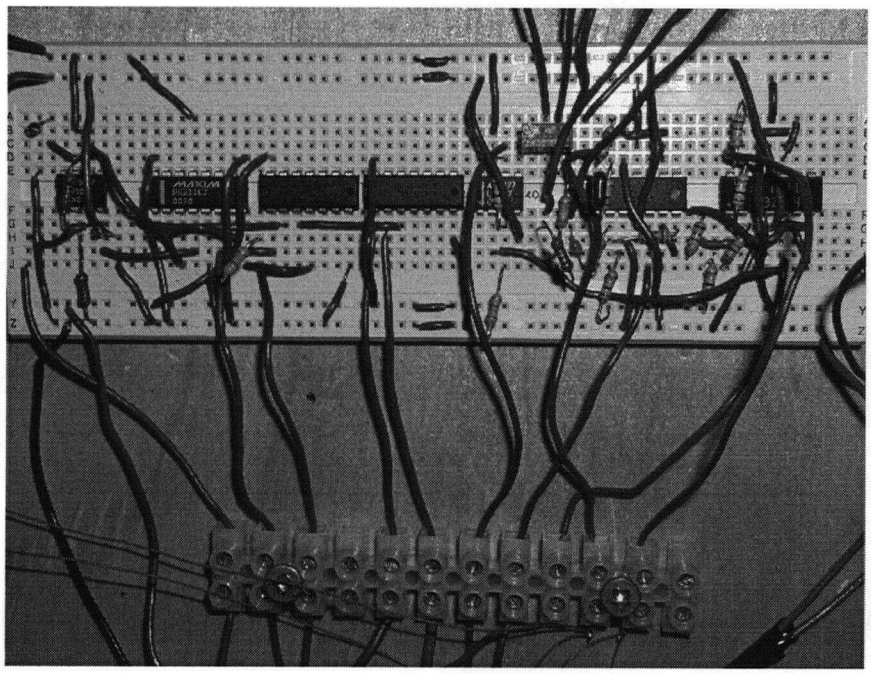

Figure A.1: *Treatment of analog signals (photograph)*

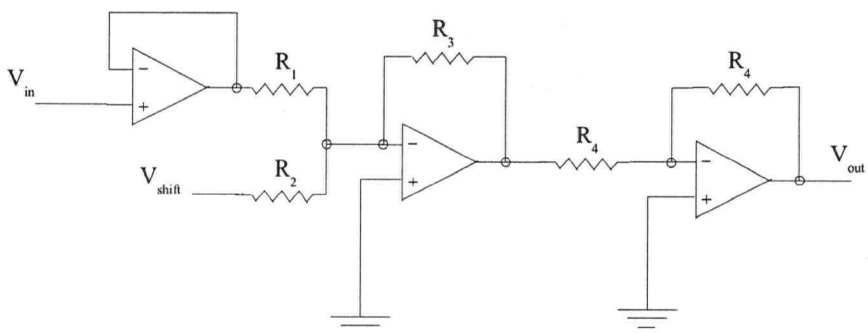

Figure A.2: *Analog treatment of the tachometer signal (for TDS input)*

$$V_{out} = \frac{R_3}{R_1} \cdot V_{in} + \frac{R_3}{R_2} \cdot V_{shift} \tag{A.1}$$

A.1.1 Parameters used for the Helm Unit

The following parameters were chosen for the Helm Unit's tachometer signal:

$$\begin{cases} \text{gain} = 10 \\ \text{shift} = 2.5V \end{cases} \Longrightarrow \begin{cases} R_1 = 5.1k\Omega \\ R_2 = 100k\Omega \\ R_3 = 51k\Omega \\ R_4 = 100k\Omega \\ V_{shift} = 5V \end{cases} \tag{A.2}$$

This enables the system to measure any Helm Unit rotational velocity between -83 and 83 RPM[1], as shown in Table A.1.

Rotational velocity (RPM)	-83	0	83
Rotational velocity ($rad.s^{-1}$)	-8.7	0	8.7
Tachometer signal (V)	-0.25	0	0.25
Processed tachometer signal (V)	0	2.5	5

Table A.1: Helm Unit tachometer signal treatment for TDS input

A.1.2 Parameters used for the Drive Unit

The Drive Unit parameters were chosen as follows:

[1]This complies with typical maximum expected helm rotational velocities of approximately 60 RPM

$$\begin{cases} \text{gain} &= 1 \\ \text{shift} &= 2.5V \end{cases} \Longrightarrow \begin{cases} R_1 &= 51k\Omega \\ R_2 &= 100k\Omega \\ R_3 &= 51k\Omega \\ R_4 &= 100k\Omega \\ V_{shift} &= 5V \end{cases} \tag{A.3}$$

This allows acceptable rotational velocities at the Drive Unit to range from -580 to 580 RPM, as shown in Table A.2.

Rotational velocity (RPM)	-580	0	580
Rotational velocity ($rad.s^{-1}$)	-60.7	0	60.7
Tachometer signal (V)	-2.5	0	2.5
Processed tachometer signal (V)	0	2.5	5

Table A.2: Drive Unit tachometer signal treatment for TDS input

A.2 Modifications of the tachometer signals for laptop input

The Drive Unit's tachometer signal can be directly input to the laptop ADC (selecting an ADC gain of 4 allows the $\{-580, 580\text{RPM}\}$ Drive Unit rotational velocity range to map directly into the whole $\{-10, 10V\}$ input range). It is however necessary to amplify the Helm Unit's tachometer signal. The circuit shown in Fig A.3 is used to achieve this.

The circuit consists of a non–inverting amplifier, which has the behavior described in Equ A.4:

$$V_{out} = \left(1 + \frac{R_2}{R_1}\right) \cdot V_{in} \tag{A.4}$$

The following parameters were selected:

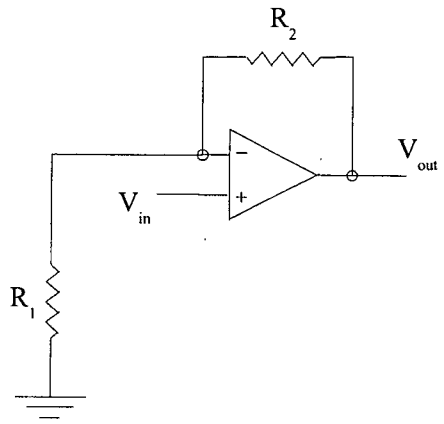

Figure A.3: *Analog treatment of the helm unit's tachometer signal (for the laptop)*

$$\text{gain} = 5 \Longrightarrow \begin{cases} R_1 &= 24k\Omega \\ R_2 &= 100k\Omega \end{cases} \tag{A.5}$$

This allows one to map the $\{-83, 83\text{RPM}\}$ rotational velocity range into the $\{-1.25, 1.25V\}$ laptop ADC input range, therefore making it possible to use the whole 12 bits of precision, with an internal ADC gain of 8. This is detailed in Table A.3:

Rotational velocity (RPM)	-83	0	83
Rotational velocity ($rad.s^{-1}$)	-8.7	0	8.7
Tachometer signal (V)	-0.25	0	0.25
Processed tachometer signal (V)	-1.25	0	1.25

Table A.3: Drive Unit tachometer signal treatment for laptop input

A.3 Treatment of possible sources for DC motor command

As detailed in 4.2.1, both the laptop and TDS_2 can send increments to the Drive Unit's servo amplifier. The output from the TDS DAC is a PWM signal, which needs to be converted to an analog signal before it is fed to the Drive Unit's servo amplifier. This is done using a PWM to analog converter embedded inside the servo amplifier. This converter takes two inputs: the PWM signal itself, and a direction (DIR) digital signal. Correspondance between the input and output is shown is Table A.4 (the PWM column indicates the ratio of time spent at HIGH level – +5V – to time spent at LOW level – 0V –).

Input		Output
PWM	DIR	
100%	LOW	-10V
50%	LOW	-5V
0%		0V
50%	HIGH	+5V
100%	HIGH	+10V

Table A.4: PWM to analog converter

On the other hand, output from the laptop DAC is limited to the $\{-5, 5V\}$ range, hence it needs to be amplified to reach the $\{-10, 10V\}$ range output by the PWM to analog converter. A non–inverting amplifier of gain 2 is used to this means.

Finally, there needs to be a way of physically switching from one actuation command source (TDS_2) to the other (laptop) and vice versa. A digital switch was used to achieve this. The switch itself is commanded by a digital level (V_{switch}): if V_{switch} is low then the effective command to the servo amplifier

comes from TDS_2, while it comes from the laptop if V_{switch} is high. V_{switch}'s level is determined by TDS_2's stateline, as detailed in 4.5.3.

Fig A.4 summarizes the above considerations.

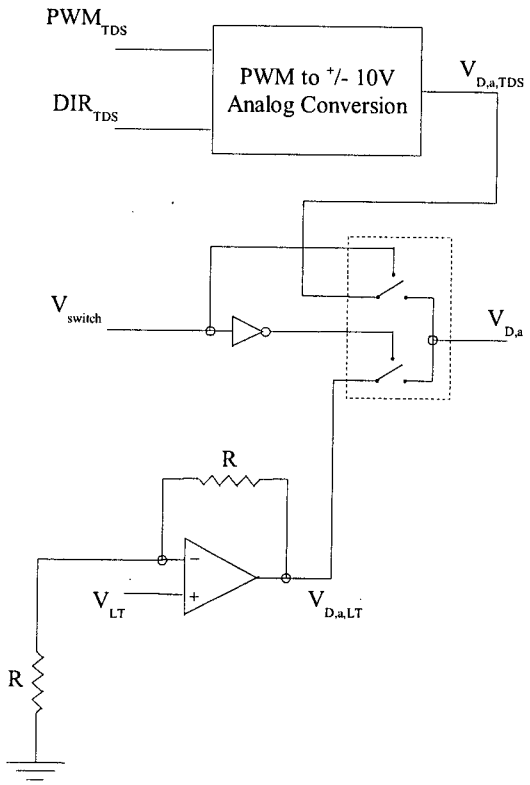

Figure A.4: *Treatment of possible sources of control signal*

Appendix B

Class definitions of the Distributed Error Detection utilities

This appendix provides a detailed description of the classes from which the Distributed Error Detection utilities are instantiated. It is meant to supplement the description given in 4.5.1.

- Table B.1 describes `2NodeErrorDetection`'s most important instance variables and methods
- Table B.2 describes `2NodeValueErrorDetection`'s most important instance variables and methods
- Table B.3 details the look–up table used by `2NodeValueErrorDetection`'s `Compare_MS:` method. As shown, all possible combinations of local and remote measurements state vectors are listed in the table[1], and in each case, an estimation of the sensors state, links state and ECU's state vectors is provided[2]. Furthermore, the – correct – angle which should be used by high–level methods is also indicated (chosen amongst the local and

[1]The "detail" columns provide example possibilities

[2]Note that the estimations shown in the table correspond to the example Measurement State vectors of the "detail" columns

remote estimated correct angles). In cases where the indications from the measurements state vectors are contradictory, the safest measure is to use the last iteration's agreed–on correct angle (`anglek1`), at least as an emergency solution.

- Table B.4 describes the `3NodeErrorDetection`'s most important instance variables and methods
- Table B.5 describes `3NodeValueErrorDetection`'s most important instance variables and methods.

Instance Variables	
Name	Description
`Helm2ED`	Helm 2–Node Value ECU Error Detection utility
`Drive2ED`	Drive 2–Node Value ECU Error Detection utility
`ESt`	ES_t: time domain ECU state vector (2–bit integer)
`ES`	ES: ECU State vector (2–bit integer)

Methods		
Name	Stack Diagram	Description
`Update_l:`	$(\ \theta_{H,e}\ \ MS_H\ \ \theta_{D,e}\ MS_D\ —)$	Updates the local data of `Helm2ED` and `Drive2ED`
`Update_r:`	$(\ \theta_{H,e}\ \ MS_H\ \ \theta_{D,e}\ MS_D\ —)$	Updates the remote data of `Helm2ED` and `Drive2ED`
`Time_fault:`	$(\ —)$	Updates ES_t to indicate that the remote ECU is faulty in the time domain
`Detect_errors:`	$(\ —\ ES\ \theta_H\ \theta_D)$	Performs error detection on both `Helm2ED` and `Drive2ED`. Combines the results and updates ES. Outputs ES and the correct θ_H and θ_D.

Table B.1: The 2NodeErrorDetection class

Instance Variables	
Name	Description
`anglek`	$\theta(T_k)$: correct angle at time T_k
`anglek1`	$\theta(T_{k-1})$
`angle_l`	$\theta_{e,l}$: angle output by the Local Error Detection utility on the local ECU
`angle_r`	$\theta_{e,r}$: angle output by the Local Error Detection utility on the remote ECU
`MS_l`	MS_l: local measurement state vector (3–bit integer)
`MS_r`	MS_r: remote measurement state vector
`SS`	SS: sensor state vector (3–bit integer)
`LS`	LS: link state vector (6–bit integer)
`ESve`	$ES_{v,e}$: estimated ECU value state vector

Methods		
Name	Stack Diagram	Description
`Update_l:`	*(θ_e MS —)*	Updates $\theta_{e,l}$ and MS_l
`Update_r:`	*(θ_e MS —)*	Updates $\theta_{e,r}$ and MS_r
`Compare_MS:`	*(— $\theta(T_k)$)*	Compares the measurement states vectors using the look–up table described in Table B.3, updates SS, LS and $ES_{v,e}$ and outputs the correct angle.

Table B.2: The 2NodeValueErrorDetection class

MS_l		MS_r		SS	LS	ESve	angle
sum	detail	sum	detail				
3		3		1 1 1	1 1 1 1 1 1	1 1	angle_l
3		2	1 1 0	1 1 1	1 1 1 1 1 0	1 1	angle_l
3		1		1 1 1	1 1 1 1 1 1	1 0	angle_l
3		0		1 1 1	1 1 1 1 1 1	1 0	angle_l
2	1 1 0	3		1 1 1	1 1 0 1 1 1	1 1	angle_l
2	1 1 0	2	1 1 0	1 1 0	1 1 1 1 1 0	1 1	angle_l
	1 1 0		1 0 1	1 1 1	1 1 0 1 0 1	1 1	angle_l
2	1 1 0	1		1 1 0	1 1 1 1 1 1	1 0	angle_l
2	1 1 0	0		1 1 0	1 1 1 1 1 1	1 0	angle_l
1		3		1 1 1	1 1 1 1 1 1	0 1	angle_r
1		2	1 1 0	1 1 0	1 1 1 1 1 1	0 1	angle_r
1		1		1 1 1	1 1 1 1 1 1	0 0	anglek1
1		0		1 1 1	1 1 1 1 1 1	0 0	anglek1
0		3		1 1 1	1 1 1 1 1 1	0 1	angle_r
0		2	1 1 0	1 1 0	1 1 1 1 1 1	0 1	angle_r
0		1		1 1 1	1 1 1 1 1 1	0 0	anglek1
0		0		1 1 1	1 1 1 1 1 1	0 0	anglek1

Table B.3: Look–up table used by the 2NodeErrorDetection class

Instance Variables	
Name	Description
`Helm3ED`	Helm 3–Node Value ECU Error Detection utility
`Drive3ED`	Drive 3–Node Value ECU Error Detection utility
`ESt`	ES_t: time domain ECU state vector (3–bit integer)
`ES`	ES: ECU state vector (3–bit integer)

Methods		
Name	Stack Diagram	Description
`Time_fault:`	*(n —)*	Updates ES_t when a ECU_n is faulty in the time domain
`Update_data:`	*($\theta_{H,e}$ MS_H $\theta_{D,e}$ MS_D n —)*	Updates `Helm3ED` and `Drive3ED`'s data from ECU_n
`Compute_ES:`	*(—)*	Combines the $ES_{v,e}$ vectors from `Helm3ED` and `Drive3ED` with ES_t. Updates ES.
`Pull_SL:`	*(—)*	Pulls the statelines according to the ECU state vector ES
`Detect_errors:`	*(— ES θ_H θ_D)*	Performs `Helm3ED` and `Drive3ED` error detection, computes ES and pulls the statelines accordingly. Outputs ES and the correct helm and drive angles

Table B.4: The 3NodeErrorDetection class

Instance Variables	
Name	Description
`anglek`	$\theta(T_k)$: correct angle at time T_k
`anglek1`	$\theta(T_{k-1})$
`angles(3)`	Array containing the 3 angles from the Local Error Detection utilities
`MS(3)`	Array containing the 3 measurement state vectors from the Local Error Detection utilities
`accuracy(3)`	Array of 3 values specfiying the maximum acceptable offsets between estimated correct angles from the 3 ECUs' Local Error Detection utilities
`DR_ind`	Data replication indicator (3–bit integer): indicates the results from the inter–ECU comparison (Compare_angles: method)
`ESve`	$ES_{v,e}$: estimated value ECU state vector (3–bit integer)

Methods		
Name	Stack Diagram	Description
`Update_data:`	*(θ_e MS n —)*	Updates θ_e and MS for ECU_n
`Compare_angles:`	*(— $ES_{v,e}$)*	Compares the 3 angles (majority voting), using `accuracy`, and outputs the estimated value ECU state vector $ES_{v,e}$
`Compute_angle:`	*(— $\theta(T_k)$)*	Computes the partial averaging of correct angles. Outputs the resulting correct angle

Table B.5: The 3NodeValueErrorDetection class

Appendix C

Execution Tables

This appendix provides the Execution Tables of TDS_2 and of the laptop in Table C.1 and Table C.2 respectively. It also describes the `ExecutionTable` class in Table C.3.

Execution Table		
1	`MSB1a`: Acquire all sensors Perform Local Error Detection	
2	`MSB2a`: Receive LocalED results from TDS_2 (**synch** with TDS_2)	
3	`MSB3a`: Send LocalED results (**synch** with TDS_2)	`MSB3b`: Send LocalED results (no synch)
4	`MSB4a`: Check if laptop is ready	
5	`MSB5a`: Receive LocalED results from laptop (**synch** with laptop)	
6	`MSB6a`: Perform 3–Node Value Error Detection	`MSB6b`: Perform 2–Node Value Error Detection
7	`MSB7a`: Wait for TDS_2 to trigger loop–closing (**synch** with TDS_2)	`MSB7b`: Wait for T_{lc} ms to close the loop

Table C.1: Execution Table for TDS_1

Execution Table			
1	`MSB1a`: Acquire all sensors Perform Local Error Detection		
2	`MSB2a`: Notify that laptop is ready Wait for ACK from TDS_1 and TDS_2 (**synch** with both TDS boards)	`MSB2b`: Notify that laptop is ready Wait for ACK from TDS_1 (**synch** with TDS_1)	`MSB2c`: Notify that laptop is ready Wait for ACK from TDS_2 (**synch** with TDS_2)
3	`MSB3a`: Receive LocalED results from both TDS boards	`MSB3b`: Receive LocalED results from TDS_1	`MSB3c`: Receive LocalED results from TDS_2
4	`MSB4a`: Perform 3–Node Value Error Detection	`MSB4b`: Perform 2–Node Value Error Detection	
5	`MSB5a`: Perform control Send actuation command to the Drive Unit		

Table C.2: Execution Table for the laptop

Instance Variables	
Name	Description
`ET`	Execution Table: this is the table itself, containing for each step the Execution Tokens of the MSB's, and the index of the MSB to be executed

Methods		
Name	Stack Diagram	Description
`ClassInit:`	*(—)*	Initializes `ET`
`Fill_step:`	*(XTn ... XT1 ExecInd n k —)*	Fills step k with XT's *XT1* to *XTn*, and update index of the XT to be executed to *ExecInd*
`Change_ind:`	*(i k —)*	Changes the XT to be executed at step k. If $i = 0$, no XT will be executed at this step, if $i \neq 0$ then *XTi* will be executed
`kExec:`	*(k —)*	Executes the specified XT (if any) at step k
`Exec:`	*(—)*	Executes all steps of `ET` successively

Table C.3: The ExecutionTable class