THE TIME-BASED APPROACH TO EMBEDDED PROGRAMMING: A HARDWARE-IN-THE-LOOP SIMULATION FRAMEWORK

by

Judith Liebman

Memorandum No. UCB/ERL M02/16

16 May 2002

Cover

THE TIME-BASED APPROACH TO EMBEDDED PROGRAMMING: A HARDWARE-IN-THE-LOOP SIMULATION FRAMEWORK

by

Judith Liebman

Memorandum No. UCB/ERL M02/16

16 May 2002

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

The Time-Based Approach to Embedded Programming: A Hardware-in-the-Loop Simulation Framework

by Judith Liebman

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II.**

Approval for the Report and Comprehensive Examination:

Committee:

Shankar Sastry
Research Advisor

Date

* * * * * *

Thomas A. Henzinger
Second Reader

Date

Abstract

One of the most troublesome issues in embedded controller design today is the failure to account for the interaction of the control laws with their implementation. Another problem in current embedded software design is the lack of subsystem re-usability. This dissertation proposes an embedded software design methodology that will connect the mathematical control laws with their software implementations and will build in component reuse. To accomplish this goal we explore the notion of time-based control and the tools of platform-based design. A key feature of our design is the use of Giotto, a high level language developed for programming time-based embedded systems. In this dissertation we also propose and use a hardware-in-the-loop simulation framework to test embedded systems and further explore the connection between control laws and their embedded realizations. This simulation framework provides repeatable, safe testing of the implemented control system. We have built both the embedded control system and a hardware-in-the-loop simulator for a helicopter uninhabited aerial vehicle (UAV). We present the detailed design of these two halves working in real-time. We conclude this dissertation with the experimental results of the embedded control system flying the simulator.

Acknowledgments

First, I would like to acknowledge my co-workers on this project, Cedric Ma and Ben Horowitz, who have truly been invaluable, hardworking, and inspiring.

I would especially like to thank David Shim for going out of his way to share his practical and vital knowledge about the helicopters and their control.

Thanks also goes to John Koo, who initiated this research and has been an important mentor for me. I am also grateful for his efforts to spear-head the writing of several papers that include work covered in this dissertation.

I would like to thank Ron Tal for his help in providing useful resources and adding practical structure and guidelines to this project from its conception.

I would also like to thank Professor Henzinger for his support of this research project.

I greatly appreciate the time, energy, and enthusiasm Professor Alberto Sangiovanni-Vintencelli added to this project and his willingness to meet and correspond with us regarding the details of the design.

I would also like to thank Peter Ray for keeping me sane and motivated, and my family and roommates for their continual support.

Lastly, I'd like to extend my greatest thanks to my advisor, Professor Shankar Sastry. His guidance and insight started me on this project, and continued to support this research through to this day. I greatly appreciate all of the meetings and frank conversations with Professor Sastry that contributed to my personal development as well as to my research and graduate career throughout the past two years.

Contents

List of Figures

Chapter 1

Introduction

In this dissertation we present a design methodology for embedded control software and a hardware-in-the-loop simulation framework for evaluating that software. Our strategy utilizes *platform-based design* and *time-based control.* Platform-based design is a methodology that builds in abstraction layers to promote modularity and a 'meet-in-the-middle' approach to development. Time-based control, also referred to as synchronous control, is a control configuration in which control tasks are run based on time, as opposed to being based on events. We apply this method to the example system of a helicopter UAV and present the final design of both the embedded control software and the hardware-in-the-loop simulator. Finally, we present results of the helicopter embedded control software flying the hardware-in-the-loop simulator.

1.1 Problem Statement

Automation of traditionally human-controlled domains has long been a driving force within the controls research community. Now, automation systems thrive within industrial plants, vehicles, airplanes and even home appliances. Their

pervasive growth has been aided by advances in integrated circuit capabilities and control theory. The design process for these systems, for the most part, has been a two step process. First the theoretical control laws are chosen, then an implementation method is selected and built. Exhaustive testing is the general method for determining if the implementation achieves a realization that is close enough to the theoretical one. However this may not be the ideal method for bridging control laws with their implementations.

Originally, custom hardware was the most economical way to implement control laws. However, more recently increases in the computing power of microprocessors have shifted the majority of implementations to software. Unfortunately, currently pervasive high level languages do not provide for all the needs of an embedded systems programmer. 'Real-time' programs that utilize standard languages are often a combination of nondeterministic timing and quick fixes. Furthermore, software structures for embedded systems are reinvented with every system. These methods did produce functional systems when the overall complexity was low. However, recent incidents where software bugs caused severe problems in very expensive systems such as the Mars Polar Lander and the Ariane rocket, point out the risks associated with using an outdated approach when developing embedded controllers.

One of the most serious problems in controller design is the current disregard for the interaction of the control laws with their implementation. When a control law is designed, the computational power of the implementation platform is only grossly estimated. This neglect leads to long re-design cycles when the timing requirements of the applications are not met. On the other hand, implementations alter control algorithms by introducing delays or other non-idealities. This situation arises from the difficulty of mixing implementation and functional design.

Another problem in current embedded software controller design is the deficiency in component re-usability. Reusing components that are specific to

function or implementation decreases time-to-market and validation time. Assuming a component is fully validated, the only obstacle is composing the objects to work properly. Components can be full subsystems such as engine controllers and ABS for cars, or sensors such as GPS for airplanes, or software modules such as device drivers, operating systems and algorithms.

1.2 Solution Methods

This dissertation focuses on two main methods to bridge the divide between control laws and their implementations: (1) the embedded software design of the implementation itself, and (2) testing the finalized implementation.

1.2.1 Embedded Software Design

This dissertation proposes and displays an embedded software design methodology that will bridge the functional description of the control laws with their implementations and will maximize component reuse. In order to achieve the first goal, we will first explore two general implementation strategies: time-based control and event-based control. This dissertation will chose to focus on time-based control due to its advantages in validation and scheduling analysis. Through the use of time-based control and a newly developed high level language for time-based embedded systems, we will create a reasonable and supportive connection between the control laws and their implementations. In particular, the choice of a specific software platform to guarantee correct timing performance for the control laws is of interest. Here we focus on the Giotto software platform and we show how this platform substantially aids the development of correct embedded controller software in comparison to other approaches.

In order to achieve the second goal we will draw on the principles *platform-based design* [San02]. A platform, in this context, is a layer of abstraction that hides the unnecessary details of the underlying implementation and yet carries enough information about the layers below to prevent design iterations.

1.2.2 Hardware-in-the-Loop Simulation

In this dissertation we also propose and use a hardware-in-the-loop simulation framework for the evaluation and further exploration of embedded systems. This simulation framework can provide the key service of verifying that the embedded implementation of the control laws is similar to the theoretical control law behavior, without the risk of running the embedded implementation in the physical environment. This is accomplished by running the actual embedded control system, but simulating the plant, sensors and actuators. Therefore, as can be seen in Figure 1.1, the hardware that is 'in-the-loop' is the embedded control system hardware. This type of simulation is vital for the embedded control community. As can be seen in Figure 1.2 the control laws themselves are tested using a simulation during their development. Since the transition from the control laws to the embedded control system is a rocky one, both manual and currently unstructured as discussed above, the embedded control system should similarly be tested in simulation, i.e. a hardware-in-the-loop simulation. Furthermore a hardware-in-the-loop framework provides repeatable, convenient tests for embedded hardware and software systems. Otherwise, testing these systems needs to include the environment, which produces inconclusive results due to the inherent variability of the environment. The nature of the hardware-in-the-loop framework allows for a controlled study of the delay, guarantees, and specific behavior of the embedded software layer. Also, testing in the environment can be hazardous. HIL simulation allows potentially dangerous tests to be carried out, such as the response under duress or multi-vehicle coordination. The final benefit to this approach is the ease of transfer between control sys-

tems developed on the HIL simulator and the final embedded product. Unlike other simulation environments, the HIL simulator is a practical step directly before using the final embedded control system in the environment.

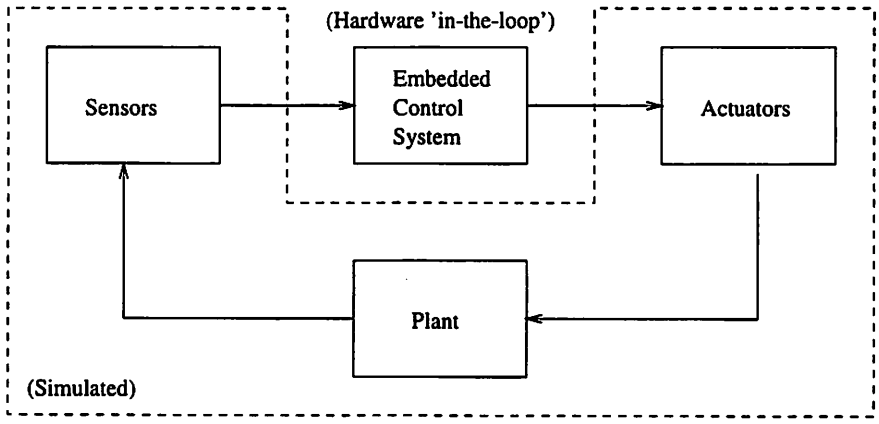


Figure 1.1: Hardware-in-the-loop simulator description

1.3 Application to Helicopter UAV

We present the platform-based design methodology for embedded controller design and the hardware-in-the-loop simulation framework via a challenging example of automatic control: a helicopter based UAV. We have built both the embedded control system and the hardware-in-the-loop simulator for this nontrivial system. With these two halves working together in real-time we conclude this dissertation with the experimental results of our embedded control system design flying our simulator. The difficulty and complexity of the application serve well the purpose of underlining the features of the design method and demonstrating its power. The choices of design solutions are somewhat application-dependent but the overall scheme is not, so that the example could provide a general guideline for the application of our method. The methodology we propose works particularly to integrate systems with the following key traits:

1. They contain embedded, real-time computers.

2. They often integrate subsystems that were designed to work independently — for example, sensors from different vendors.

3. Their proper operation is important to ensure human safety.

4. They can utilize legacy code such as device drivers or controllers. This is due to the fact that automation projects have long lives and many design iterations. Careful reuse of code often saves money and increases product quality.

1.4 Dissertation Structure

The structure of this dissertation is as follows. In Chapter 2, we lay the groundwork for the helicopter example. In Chapter 3 we explore the applicability of time-based control. Next, in Chapter 4, we introduce the reader to the principles of platform-based design. In Chapter 5, we describe a software platform for programming time-based controller applications. Then, in Chapter 6, we present helicopter UAV embedded software designs that use the concepts of the previous chapters, and we discuss how to compare designs using hardware-in-the-loop simulation. In Chapter 7 we present in detail the implementation of the embedded controller software and the hardware-in-the-loop simulation software. Finally, in Chapter 8 we present resulting data from the working combination of the UAV embedded control system and hardware-in-the-loop simulator.

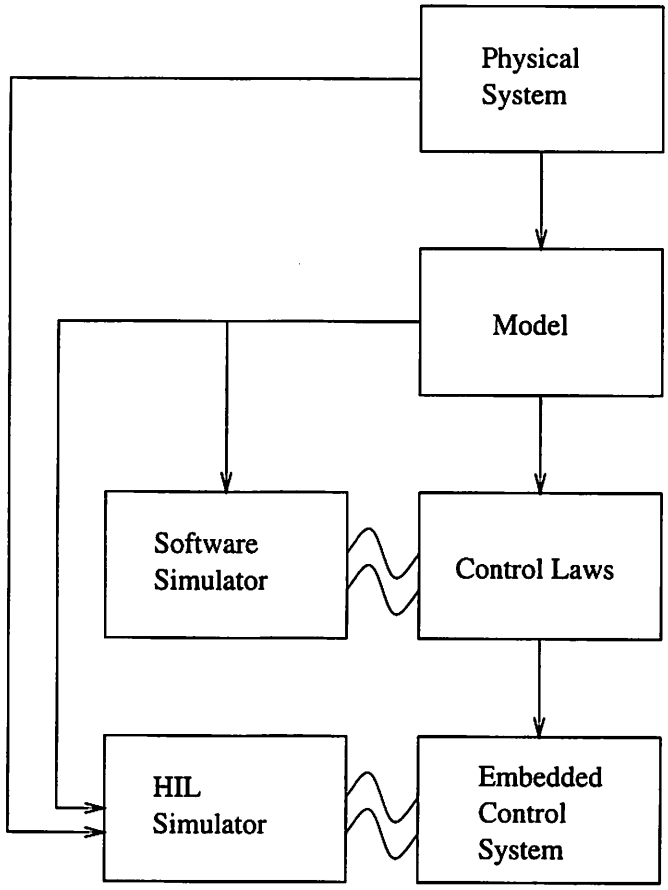


Figure 1.2: Design flow for an embedded control system: On the right side of the diagram is the design flow of the embedded control system starting from the physical system, on the left side of the diagram are testing methods for different pieces of the design flow on the right. A software simulator can test the control laws, for example in a Simulink environment, while a HIL simulator tests the final embedded control system itself.

Chapter 2

Background for a Model Helicopter

In this section, we introduce the Berkeley Aerial Robot (BEAR) helicopters, and motivate the redesign of their embedded software. We begin with a brief description of the BEAR helicopters and of why autonomous flight is difficult (Section 2.1). We next discuss the development of the first generation flight control system (Section 2.2). Section 2.2 includes a description of the sensors needed for autonomous flight, an overview of the system identification process and the final dynamic model, and an overview of the control laws developed for and used on the first generation helicopters. Finally, in Section 2.3 we describe some of the limitations of this first generation embedded control system.

2.1 The BEAR Helicopters

The first goal of the BEAR project was to build a flight control system for small, remotely controlled helicopters. The aim was to fly autonomously and to provide a base for research in other areas such as vision, pursuit evasion games,

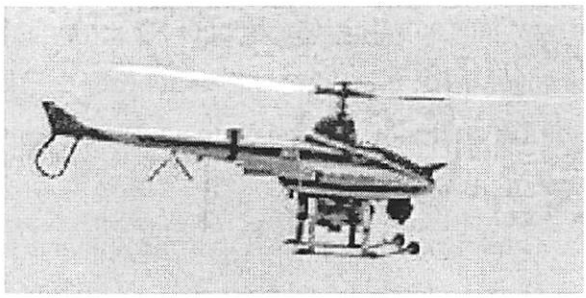


Figure 2.1: A Yamaha R-50 Berkeley autonomous helicopter

hybrid systems, etc. Basic autonomous flight capabilities include hovering, forward flight, turning at a fixed point, and so on. More advanced maneuvers include formation flying and obstacle avoidance. However, it is difficult to achieve even basic autonomous flight, for the following reasons:

- The helicopter is unstable during hover. It will tip over within a few seconds if left alone. Therefore, the flight control system needs to take an active role in the stabilization of the helicopter.
- A crash is very dangerous, even at low speeds.
- The helicopter is an intricate machine, whose mechanical and electronic systems must operate harmoniously under harsh conditions, such as physical vibration and electromagnetic interference.

Moreover, it is difficult to obtain an accurate dynamic model of the helicopter:

- The helicopter controls are often coupled. For example, changing the collective pitch affects the amount of power available to the tail rotor, which temporarily affects the yaw characteristics.
- The behaviors of the helicopter are dissimilar in different flight regimes, such as hover and forward flight.

- The airflow surrounding the helicopter body is chaotic, especially near the tail rotor. In addition, the helicopter is affected by wind, and its aerodynamic behavior changes when it hovers near the ground.

In spite of the challenges, the BEAR team managed to build a working flight control system that makes autonomous flight possible. One of the autonomous helicopters, the Yamaha R-50, is shown in flight in Figure 2.1. In the Section 2.2 we introduce the structure of the controller for this system in more detail.

2.2 Flight Control System Development

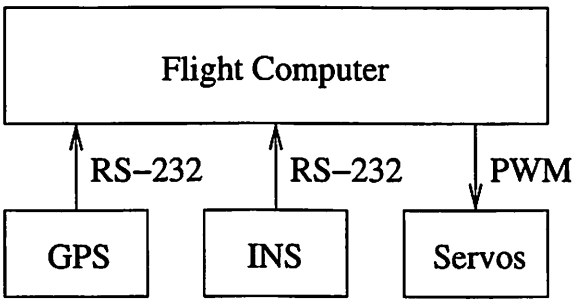


Figure 2.2: Structure of the first generation flight control system

Figure 2.2 illustrates the primary components in the first generation flight control system. The actuators consist of servo-motors controlling the throttle, main rotor collective pitch θ_M, longitudinal cyclic pitch B_1, lateral cyclic pitch A_1, and tail rotor collective pitch θ_T to generate forces and torques applied to the helicopter. We assume the use of an engine governor to regulate the main rotor RPM so the throttle is not directly controlled by our flight control system, and therefore does not appear as an input in the dynamic equations.

The primary sensors of the flight control system are as follows:

Inertial Navigation System (INS). The INS consists of accelerometers and

rotational rate sensors that provide frequent estimations of the helicopter's position, velocity, orientation, and rate of rotation. Although this estimate is provided at a high rate —roughly 100 Hz— the error in estimate could grow unbounded over time, due to sensor noise and limits in sensor accuracy.

Global Positioning System (GPS). The GPS solves the INS drift problem by providing a position measurement whose error is small —on the order of 1 cm— and bounded over time. However, this accurate measurement is also infrequent —roughly 5 Hz.

An integrated INS-GPS solution uses a Kalman filter to provide frequent update of the estimated state of the helicopter.

2.2.1 System Identification

The helicopter is a dynamical system and its equations of motion can be derived from the Newton-Euler Equation[MLS94] for a rigid body subject to body forces $\mathbf{f} \in \Re^3$ and torques $\tau \in \Re^3$ applied at the center of mass. The position and velocity of the center of mass are given by $\mathbf{X^b} \in \Re^3$ and $\mathbf{V^b} = \dot{\mathbf{X}}^\mathbf{b} \in \Re^3$, respectively, expressed in terms of the inertial frame, also called body coordinates, using the North-East-Down orientation. We parameterized the orientation $\mathbf{R} \in SO(3)$ of the helicopter relative to the inertial frame by ZYX(or "roll, pitch, yaw"), and Euler angles are denoted by $\Gamma = [\Phi\ \Theta\ \Psi]^T$. Let $\omega^b \in \Re^3$ and $\mathbf{a^b} \in \Re^3$ be the body angular velocity vector and the body linear acceleration vector. The equations of motion of the helicopter model can be written as:

$$\begin{cases} \dot{\mathbf{X}}^{\mathbf{b}} &= \mathbf{V}^{\mathbf{b}} \\ \dot{\mathbf{V}}^{\mathbf{b}} &= \frac{1}{m}\mathbf{R}(\Gamma)\mathbf{f}(\mathbf{u}) \\ \dot{\Gamma} &= \Omega(\Gamma)\omega^b \\ \dot{\omega} &= \mathbf{I}^{-1}(\tau(u) - \omega^b \times \omega^b) \end{cases} \tag{2.1}$$

where m is the body mass, $\mathbf{I} \in \Re^{3\times3}$ is the inertial matrix, $\mathbf{\Omega} : \Re^3 \rightarrow \Re^{3\times3}$ maps the body rotational velocity to Euler angle velocity, and $\mathbf{u} = [\theta_M \ \theta_T \ B_1 \ A_1]^T$ is the input vector. In [KS98], the above system is characterized to be non-minimum phase, i.e. it has unstable internal dynamics. Since input $\mathbf{u}$ effects both the body forces and the torques, the linear and rotational dynamics are coupled. Therefore, controlling a helicopter is an extremely difficult task.

The above rigid body motion equations shown in equation(2.1) have been expanded and applied to the helicopter [Shi00]. Many aerodynamic properties need to be measured and accounted for in this process. The force and torque matrices involve separate forces in the X, Y, and Z directions as well as separate torques in the roll, pitch, and yaw directions. The interacting forces are also broken down to denote their relation to the main rotor, tail rotor, fuselage, horizontal stabilizer, and vertical stabilizer. The goal of this system identification process is to accurately formulate each force and moment term and the geometric constants specific to the component and center of mass locations. Unfortunately, these parameters are very difficult to find, and still more difficult to verify. However, a certain amount of accuracy in the model is needed for simulation as well as for formulating a controller. Therefore, a different approach to system identification was also employed [Shi00].

The new approach uses empirical parameter identification rather than the theoretical model. Here, a template model for the LTI MIMO parametric identification that is proposed by Mettler *et al* is the starting point [MTK99]. This template can be derived from the linearization of the general theoretical model by discarding terms with negligible contributions [Shi00]. The template model is as follows:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{2.2}$$

where:

$$\mathbf{x} = [\, u\ v\ p\ q\ \Phi\ \Theta\ a_{1s}\ b_{1s}\ w\ r\ r_{fb}\ \Psi\,]^T \in \Re^{12} \tag{2.3}$$

$$\mathbf{u} = \left[\, u_{a_{1s}}\ u_{b_{1s}}\ u_{\theta_M}\ u_{r_{ref}}\,\right]^T \in \Re^4 \tag{2.4}$$

In this model the state is made up of: (u, v, w), the body velocities, (p, q, r), the roll pitch and yaw rates, (Φ, Θ, Ψ), the Euler angles, (a_{1s}, b_{1s}), the longitudinal and lateral flapping angle of the main rotor, and r_{fb}, the feedback gyro system state. The control vector in this framework is different than in the original more general situation. Here it is made up of: $u_{a_{1s}}$, the input to the longitudinal flapping, $u_{b_{1s}}$, the input to the lateral flapping,u_{θ_M}, the input to the main rotor collective pitch, and $u_{r_{ref}}$, the input to the yaw rate feedback system on the R-50.

The matrices $\mathbf{A} \in \Re^{12\times12}$ and $\mathbf{B} \in \Re^{12\times4}$ have the following form:

$$\mathbf{A} = \begin{bmatrix} X_u & 0 & 0 & 0 & 0 & -g & X_{a_{1_s}} & 0 & 0 & 0 & 0 & 0 \\ 0 & Y_v & 0 & 0 & g & 0 & 0 & Y_{b_{1_s}} & 0 & 0 & 0 & 0 \\ L_u & L_v & 0 & 0 & 0 & 0 & L_{a_{1_s}} & L_{b_{1_s}} & 0 & 0 & 0 & 0 \\ M_u & M_v & 0 & 0 & 0 & 0 & M_{a_{1_s}} & M_{b_{1_s}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & \frac{-1}{\tau_f} & A_{b_{1_s}} & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & B_{a_{1_s}} & \frac{-1}{\tau_f} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & Z_{a_{1_s}} & Z_{b_{1_s}} & Z_w & Z_r & 0 & 0 \\ 0 & 0 & N_p & 0 & 0 & 0 & 0 & 0 & N_w & N_r & N_{rff} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & K_r & K_{rfb} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.5}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ A_{u_{a1_s}} & A_{u_{b1_s}} & 0 & 0 \\ B_{u_{a1_s}} & B_{u_{b1_s}} & 0 & 0 \\ 0 & 0 & Z_{u_{\theta_M}} & 0 \\ 0 & 0 & N_{u_{\theta_M}} & N_{u_{\theta_T}} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.6}$$

The practicality of this method is that the parameters in the matrices above are found using flight data. The first step, therefore, is to equip the helicopter with an onboard computer which can log the pilot's control input and the vehicle response. The next step is to stage a flight test in which the pilot issues frequency-sweeping commands to each channel, roll, pitch, yaw, and vertical and the inputs and vehicle response is recorded. This flight test can also be carried out with more complicated stages in order to capture the cross-couplings of channels [Shi00].

Once the flight data is obtained, the parameters can be obtained using a prediction-error method supported in the MATLAB System Identification Toolbox. This method seeks to minimize the quadratic error between the predicted and actual position by finding the optimal system model and error model. This method is extremely sensitive to the initial guess of the parameters, therefore the parameters are identified in stages, with the first stage consisting of the angular rate/rotor dynamics. These dynamics only involve a few variables and are known to be stable, so a consistent solution is readily attained. Now, using these parameters, the next stage, the horizontal dynamics, are sought. This process continues until all parameters are obtained. Then the entire process is iterated in order to fine tune the model [Shi00].

The main benefit of this method is that our own specific flight data is used to obtain the parameters in the template. This takes into account all of the custom flight hardware. Using these methods the **A** and **B** matrices were accurately filled in and the model template is therefore complete and tailored to our exact helicopter R-50 configuration [Shi00].

$$\mathbf{A} = \begin{bmatrix} -0.1257 & 0 & 0 & 0 & 0 & -g & -g & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.4247 & 0 & 0 & g & 0 & 0 & g & 0 & 0 & 0 & 0 \\ -0.1677 & 0.0870 & 0 & 0 & 0 & 0 & 36.7050 & 161.1087 & 0 & 0 & 0 & 0 \\ -0.0823 & -0.0518 & 0 & 0 & 0 & 0 & 63.5763 & -19.4931 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & -3.4436 & 0.8287 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & .3611 & -3.4436 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -38.9954 & 9.6401 & -0.7598 & 8.4231 & 0 & 0 \\ 0 & 0 & -1.3300 & 0 & 0 & 0 & 0 & 0 & 0.0566 & -5.5105 & -44.8734 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.8157 & -11.0210 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{2.7}$$

$$\mathbf{B} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ -0.8417 & 2.8231 & 0 & 0 \\ -2.4090 & -0.3511 & 0 & 0 \\ 0 & 0 & 70.5041 & 0 \\ 0 & 0 & 23.6260 & 44.8734 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.8}$$

2.2.2 Control Techniques

The general nonlinear model, whose framework was presented in equation(2.1), was used to build a nonlinear controller with the techniques of approximate linearization and approximate state trajectory generation based on differential flatness [KS98]. Exact input-output linearization fails to transform the system into a linear and controllable system due to the couplings between the rolling moments and the lateral acceleration, and between the pitching moments and the longitudinal acceleration. These couplings are introduced due to the presence of a_{1s}, b_{1s} and T_T, the torque generated by the tail rotor. Therefore, in order to use approximate linearization, the coupling terms must be neglected by assuming that a_{1s}, b_{1s} and T_T are small. Simulations of this controller, using the model that the controller was derived from, showed that the approximate linearization controller was able to stabilize the system and provide good tracking performance [KS98]. However, performance of the approximate linearization controller degraded with increases in uncertainty and external disturbances. Also, due to the approximation assumptions that have been introduced, the actual performance in the presence of model perturbation and sensor noise may deviate from the simulated performance [Shi00].

Linear control theory was also explored. Of course, the helicopter is a highly nonlinear system. However, many practitioners choose linear control theory based on its consistent physical performance and well-defined theoretical background. It has been proved that linear control theory is able to stabilize unstable nonlinear dynamics consistently, as long as the system stays in the region where the linearity assumption holds. Therefore, the deficiency of the linear approach can be addressed with a coordinate transformation algorithm. Another difficulty with this approach is that the helicopter system is inherently MIMO with couplings among the longitudinal, lateral, vertical, and yaw dynamics. However, the cross-couplings among these channels are mild and the SISO approach has been shown to function well in practice. For these reasons, one approach to flying the first generation helicopters was to develop the controller using linear control theory and the experimental model shown above 2.2 [Shi00].

The linear controller development considers the helicopter system to be broken into four subsystems: roll, pitch, yaw, and heave. Each channel is then stabilized using proportional-differential controllers. By studying the damping responses and the root loci of the different modes, the following controller was conceived and flown on the first generation helicopter [Shi00]:

The static stabilizing feedback law is of the general form:

$$\mathbf{u}_{fb}(t) = \mathbf{f}(\mathbf{y}_{ref}(t)) \tag{2.9}$$

where the control $\mathbf{u}_{fb}$ steers the vehicle to follow the desired trajectory, specified by the difference between the current position and the desired position in the x, y, and z directions, and the difference between the current heading, Ψ, and the desired heading:

$$\mathbf{y}_{ref}(t) = \left(p_{x_{ref}}(t), p_{y_{ref}}(t), p_{z_{ref}}(t), \Psi_{ref}(t)\right) \tag{2.10}$$

The resulting specific control equations developed:

$$\begin{aligned} u_{a_1} &= -K_\Phi \Phi - K_v v - K_{p_y} \Delta_{p_y} \\ u_{b_1} &= -K_\Theta \Theta - K_u u - K_{p_x} \Delta_{p_x} \\ u_{\Theta_M} &= -K_w w - K_{p_z} \Delta_{p_z} \\ u_{r_{ref}} &= -K_\Psi \Delta_\Psi \end{aligned}$$

where:

u_{a_1}	= the input to the longitudinal flapping
u_{b_1}	= the input to the lateral flapping
u_{θ_M}	= the input to the main rotor collective pitch
$u_{r_{ref}}$	= the input to the yaw rate feedback system on the R-50
u, v, w	= body velocities
Φ, Θ, Ψ	= roll, pitch, yaw Euler angles
Δ_{p_x}	= $p_{x_{actual}} - p_{x_{ref}}$
Δ_{p_y}	= $p_{y_{actual}} - p_{y_{ref}}$
Δ_{p_z}	= $p_{z_{actual}} - p_{z_{ref}}$
$K_{p_x}, K_{p_y}, K_{p_z}$	= feedback gain for p_x, p_y, p_z: -.01, -.01, .12
K_u, K_v, K_w	= feedback gain for u, v, w: -.02, -.02, .035
K_Φ, K_Θ, K_Ψ	= feedback gain for roll, pitch, yaw: -.55, .55, 1

In this project we utilize the above control law because of its proven robustness for the specific helicopter and configuration that we are developing the embedded software for. For more information regarding the system identification process, the dynamical model of the helicopter, and the controllers, refer to [SKHS98, Koo00, Shi00].

2.3 Limitations of the First Generation System

With basic autonomous flight successfully demonstrated, the BEAR team then set off to equip a number of helicopters with a similar flight control system. Over time, two new and unfamiliar challenges emerged.

1. The first challenge resulted from a widening choice of devices: as the fleet of helicopters became more diverse, so did the selection of sensors, actuation schemes, and computing hardware. Each device provided or received data at different speeds, used different data formats, communicated using different protocols, and so on. To take just one example, the actuators of the first generation helicopter expected PWM signals as input, whereas the actuators of later helicopters had a serial interface.

   The first generation flight control system reflected the desire to demonstrate the feasibility of autonomous flight, rather than elegance of design. Consequently, the design of the initial system emphasized fast flight computer reaction, achieved by means of tightly coupled sensors, actuators, computer hardware, and embedded software. The tightly integrated flight control system was not prepared to handle the diverse assortment of new devices. Inevitably, any change to the original system required an extensive software rewrite followed by an extended verification process.

   In short, the original embedded software was not written with modularity in mind. Yet it would be prohibitively expensive to rewrite all of the software for each particular combination of devices. Instead, we would like to develop embedded software that is simple to configure, so that new components can be added or substituted with relative ease.

2. The second challenge resulted from the event-based nature of the first generation flight control computer. To ensure the fastest possible response,

the computer was set up to process the incoming sensor data as soon as it arrived and to immediately send the control output to the actuators.

As an example of the problems that arose in this event-based system, consider the following first generation setup. The GPS and INS were synchronized with each other but not with the control computer. The GPS sent readings to the control computer at 5 Hz. The INS sent readings at 100 Hz. The control computer ran the control task at 50 Hz. Because of the lack of synchronization, the sensor data seen by the control computer ranged from 0 ms to 10 ms out of date. Due to clock drift, this amount of time was nondeterministic. Similarly, the servos were triggered by a clock whose rate was independent of the control computer's clock. Since the servos were triggered at 23.78 Hz, by the time the actuators used the control data, these data could be 42 ms out of date.

Unfortunately, the different rates of the sensors, actuators, and computer resulted in a system whose timing behavior was not particularly easy to analyze. Consequently, the physical behavior of the helicopter could vary greatly from the simulation results. The event-based nature of this system will be further discussed and challenged in Chapter 3.

We have presented several limitations of the first generation helicopter system. In the next section, we discuss properties that the second generation should have in order to lessen these limitations.

Chapter 3

The Time-Based Control Philosophy

As discussed in the introductory chapter, this dissertation attempts to connect the control laws with their implementation by presenting a design methodology for the implementation of embedded control systems and by using appropriate testing methods. In this chapter we focus on the background needed for the first strategy: the design of the embedded software. Here we will explore two predominant implementation methods for embedded control systems: basing interactions on events or basing interaction on time. In Section 3.2.3 we support the decision of using time-based interfaces for automatic control systems. Next, in Section 3.3 we present extensions to a time-based embedded control system to further connect the control laws to their implementations and to place higher level control into this structure. Finally, in Section 3.4 we draw on the concepts from the previous sections and construct the goals for the second generation helicopter UAV embedded software.

3.1 Decomposition

Complex automation control systems, which the helicopter UAV is one example of, almost always require many real-time components that will need to communicate across well defined interfaces. The complexity of the systems under study is the reason for this 'divide and conquer' strategy. One of the best ways to reduce complexity in such a system is to partition the system and then reduce the subsystem interactions. This strategy suggests that the subsystem interfaces should be as small as possible [Kop98].

In keeping with the above themes, the first generation helicopter system was indeed built with many communicating subsystems. The subsystem interactions were based on events, as shown in Figure 3.1 and discussed above in Section 2.3. For example, the 'INS data ready' event causes the INS handler subsystem to begin and the 'combined data ready' event causes the control subsystem to run. Every subsystem is triggered by a specific preceding event, which may occur at any point in time. Looking at the overall system, one can see that the initial driving events that begin to trigger the software subsystems are the raw sensor data readings. Therefore, the entire embedded software system can be seen as a chain reaction entirely based on the various sensor timings. In the remainder of this section we evaluate the role of these event-based interfaces and consider their replacement with time-based interfaces.

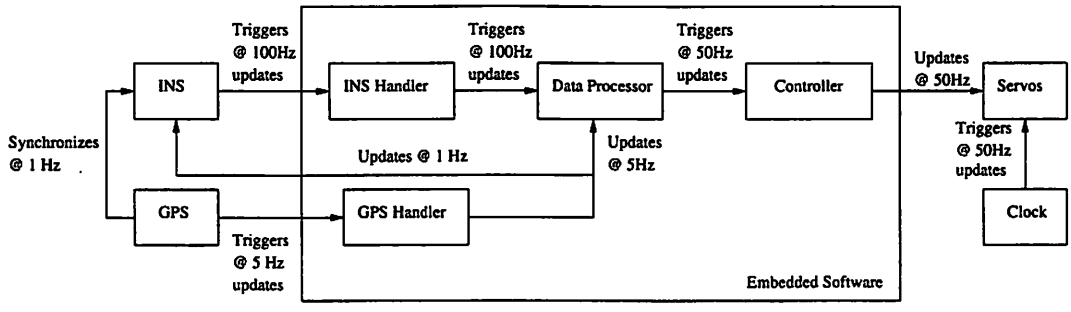


Figure 3.1: First generation helicopter software system diagram

3.2 The Form of Real-Time Interfaces

Earlier in this chapter, we explained that complex real-time systems are often broken down into smaller subsystems bridged by interfaces. In this section, we first discuss characteristics of decomposed systems that can be used to compare different interface strategies [Kop98].

Composability A system is *composable* with respect to a given property if: when the property is established at the subsystem level, system integration does not invalidate the property at the system level. Therefore, if you have a composable system, each subsystem can be tested independently, greatly reducing the overall complexity.

Reusability When multiple components satisfy well defined interfaces, they can be exchanged with one another. This allows for the reuse of legacy subsystems that have been heartily tested and whose inclusion thereby reduces complexity.

Error Detection As data is passed from one component to another, the particular interface may be able to guard against or detect errors in control flow or in data format. Subsystem errors can then be detected and contained from other subsystem processes.

Schedulability In automation control systems, subsystems need to run concurrently and often on one processor. It is crucial to verify that subsystems can be scheduled satisfactorily, although in many cases this proves a difficult problem.

Jitter Reduction The *jitter* is the variability of the time interval between data observation and use. Many controllers are designed to expect a certain fixed amount of delay between the data observation and the input of that data to the controller. However, variation in that delay time is problematic and degrades the control performance. The jitter in a

system may be a consequence of many combined factors, but is always highly dependent upon the interface type.

Stale Data Minimization Here, we refer to the mean time between data measurement and data usage as the stale data time.

The ideal interface would be designed to enable composable, reusable subsystems, to detect errors at the interface level, and to reduce jitter and stale data. First we will examine how event-based interfaces and time-based interfaces incorporate the above properties. Finally, we will contrast these two interface strategies based on their usefulness for automation control systems.

3.2.1 Event-Based Interfaces

An event-based interface bridges subsystems using event information. The format of this event information needs to be specific to the particular subsystems in question and the temporal properties of the interface also must conform to the independent timing of each subsystem. Therefore, systems using event-based interfaces can be accurately defined as tightly coupled.

Kopetz redefines event-based interfaces as *composite interfaces* because they are examples of interfaces in which unidirectional information flow is dependent upon the receiving subsystem [Kop98]. An example of this dependence on the receiving subsystem is where the event information is stored by the interface in a queue. Then the information consumer must service the queue before the next event is produced. In our first generation helicopter UAV example, again referring to Figure 3.1, although the event-based interface does not make use of a queue, the control subsystem must be ready to input the sensor data event information and begin control computations. Otherwise the subsystem producing the event information will be blocked and will not be able to continue its proper execution cycle. The conclusion is that event based interfaces create

a system in which communicating subsystems are mutually dependent.

Due to this mutual dependence found in event-based systems, several of the interface characteristics are negatively affected, while there is one main advantage:

- Since subsystems depend on each other, some properties, especially temporal properties, verified for one subsystem will not be guaranteed to hold when the system is combined. These properties may be affected by the interaction with other subsystems.
- The timing of each subsystem must be tailored to the specific system setup since the interface does not establish a common timing scheme. This interconnectedness hinders the reusability of each subsystem.
- Scheduling for event-based systems is accomplished using the infrastructure of the real-time OS. This means that the programmer sets priorities for different concurrent tasks and analyzes the schedulability by hand.
- In the above paragraph, event-based interfaces were identified as producing a system in which the subsystem sending data is dependent upon the subsystem receiving the data. This dependence enables errors to propagate back from the subsystem receiving the data. In fact, an error in either the information provider or in the information receiver will propagate to the adjoining subsystem [Kop98].
- Jitter is neither reduced nor greatly exaggerated in a well designed event-based system. Assuming transmission time is minimal in comparison with sensor output delays and computation times, an event based interface merely propagates the data from its observation to use without drastically affecting its variation in arrival time.
- Stale data is potentially optimally minimized. event-based interfaces in general have a fast response due to the fact that data is relayed across the interface immediately.

3.2.2 Time-Based Interfaces

A time-based interface bridges subsystems based on a strict adherence to a common clock. In these systems, subsystems run at a certain predetermined time, as opposed to at the occurrence of an event. Since there is no control signal crossing the interface, the subsystems execute independently. New versions of the data simply replace the previous versions without any activity needed from the receiving subsystem. The receiver fetches the most recent version of the data from the interface.

Time-based interfaces are better suited for most of the desirable interface characteristics, but not all:

- Subsystem properties can be validated in isolation since the interface composition allows each subsystem to execute independently.
- Similarly, subsystem independence maximized subsystem reuse.
- Time-based interfaces build in error detection in two ways. First of all, they do not propagate errors in the receiving subsystem backwards to the sending subsystem. Secondly, since it is known a priori when the program functions should occur, if any transmissions are erroneous in time the subsystems can detect that error [Kop98].
- Scheduling in time-based systems allows more readily for subsystem additions and schedule validation.
- Jitter is reduced but not eliminated in time-based systems. Jitter in these types of systems is mainly a result of sensor inaccuracies in time. In a time-based system, the subsystem receiving the data has a certain set period which can bound the jitter if that subsystem replaces `no-data` with either a safe input or the previous sensor reading.

- Time-based interfaces trade off all of the above advantages for an increase in stale data time. This increase can be bounded and minimized but is always a feature of the system. The cause of the stale data delay is that although our software system is time-based, the environment will always operate on an event basis. Therefore, sensor readings will come in at any time. In an event-based system, those sensor readings were the events that triggered their processing, thereby minimizing the stale data time. However, in time-based systems sensor readings are not processed until the correct instant in time. Therefore, the time dependence inherently creates stale data. Please refer to Figure 3.2 for a graphical representation.

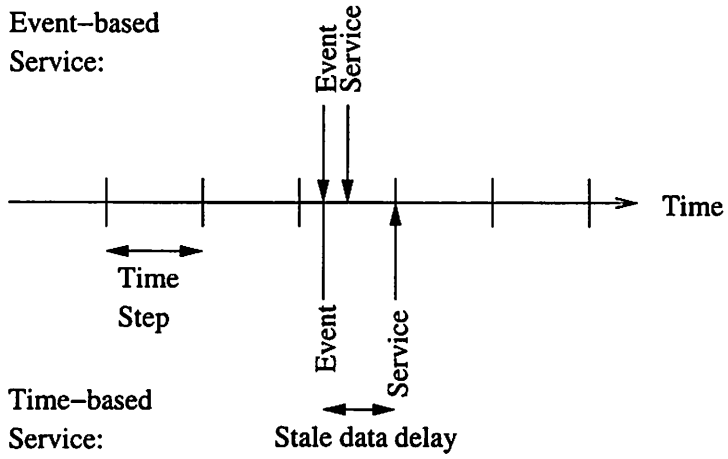


Figure 3.2: Stale data delay in event-based systems vs. time-based systems

3.2.3 Comparison of Interface Alternatives

Event-based interfaces require a close interaction between coupled subsystems. This tight integration does bring the advantages of short stale data times and ease of construction. On the other hand, event based systems allow for limited composability, reusability, scheduling analysis, jitter reduction, and may propagate control errors.

Time-based interfaces are less complex due to the independence of the subsystems. This simplicity allows time-based systems to excel in most of the interface categories. The main trade-off is the increase in stale data time depicted in Figure 3.2. One other draw back is the comparably greater initial investment needed to build the interface structure to ensure timeliness.

In the case of automation control systems the trade off is an interesting one. Here, validation and scheduling analysis are paramount due to the emphasis on safety. However, control performance is also important. Jitter reduction is a crucial factor in control performance. Stale data can also degrade control performance, even with the use of a control algorithm designed to handle specific data delays [ÅBE+99]. In this project we focus on validation and scheduling analysis for control systems. Therefore we choose to explore and implement a time-based helicopter UAV controller. To mitigate the drawbacks of this choice we concentrate on bounding the stale data time, shown in Section 6.2.1, and we employ a time-based language to ensure the timeliness of our control system.

3.3 Extending a Time-Based System

The chosen time-based interface system does require a more difficult setup process to build the low level control embedded software. However, when this is accomplished it not only has the advantages discussed above, but also leaves open simple ways to further explore control performance and to enlarge the overall scope of the system. In this section we briefly touch on a few possible extensions for a time-based real-time control system.

3.3.1 Scheduling and Control Integration

In a time-based system, information about the control law should influence the scheduler. There are many types of information that could be passed, with the most extensive information transfer resulting in the best control performance.

Information From the Control Code for the Scheduler

At the most basic level, worst-case-execution-times (WCET's) for the subsystem tasks can be used to verify schedulability. A WCET should provide a prediction that is a tight upper bound of the longest possible execution time. There are two main components that go into WCET estimation [Kop97]:

Structure-level analysis: determining the longest possible path, or the "critical path", in the code. This analysis requires solving the difficult problem of bounding loop iterations and recursion depths. Both symbolic execution and integer linear programming have been employed in this arena [ÅBE+99, LMW95]. In order to make structure-level analysis a more tractable problem, a few restrictions can be imposed on the code [Kop97]:

- No unbounded control statements at the beginning of a loop,
- No recursive function calls,
- No dynamic data structures.

Hardware-level analysis: determining the WCET of one "atomic" block of object code on a particular computing platform. In general, techniques such as caching, pipelining, and speculative execution, which improve *average* execution time, complicate hardware-level analysis. Assuming the worst case happens every time results in estimates that are off by a factor of 10-100 or more, and is a recipe for wasting performance. An

extended integer linear programming (ILP) approach has been used in this area as well [LMW95]. Other approaches are overviewed in [Kop97].

Combining these two analyses provides an estimate for the WCET of a particular piece of code on a particular computing platform.

As an alternative, worst-case-response-time (WCRT) can also be used to analyze schedulability. A task's WCRT is the maximum possible time between when the task was enabled and when it completes execution. The main difference here is that this time includes interference from other (higher priority) tasks. The algorithm to compute WCRT begins with the assumption that an upper bound is the length of the longest interval in which the processor is continuously busy and no tasks of priority lower than the one to be estimated execute. A more detailed explanation and the continuation of the algorithm can be found in [BLMSV98].

The above information, to the extent that it can be calculated, along with control task periods with period ranges and control task deadlines are typically transfered to a scheduler. This information allows the scheduler to choose a scheduling scheme and determine schedulability.

Information From the Scheduler For the Control Code

On the other hand, information is rarely passed from the scheduler to the control code. However, closing the loop in this manner has been proposed in order to explore the following compelling goals [ÅBE+99]:

- The control algorithm should take into account the availability of computing resources.
- When the scheduler is in an overloaded situation the control tasks should be able to adapt task parameters so that stability and minimal control

performance are maintained. That is, the system should be able to dynamically trade off control performance with CPU utilization.

This integrated control and scheduling approach to embedded system design can be much more readily explored on a time-based system than on an event-based system due to the relative ease of timing analyses.

3.3.2 High Level Control and Hybrid Systems Applications

Another extension to time-based low level control systems is the addition of higher level algorithms.

As applications become more complex, hierarchical control levels are often employed both to reduce complexity and to allow for validation. For example, in the avionics domain, hierarchical control structures have been employed for strategic planning and multi-vehicle flight [KSS+98]. In [KLMS01] the case for using a time-based lowest level is presented. Each of the higher level applications discussed in [KLMS01] rely on guarantees of execution times that are available from a time-based lower level. The four higher level applications discussed are used in the design of large-scale autonomous control systems:

Control Law Synthesis Since high level control laws are designed using information about the vehicle dynamics and lower levels of embedded software, validation requires guarantees from the lower levels. Interesting and applicable high level control exploration areas include least restrictive control, where a game theoretic approach is used to synthesize a control law that keeps the vehicle states within a safe flight envelope and computes the maximal safe set by solving the corresponding Hamilton-Jacobi equation [LTS99].

Control Mode Switching Synthesis A variety of high level tasks can be completed by appropriately switching between a set of low level controllers. One example is the reachability task, where the goal is to reach a final control mode from an initial one. This control mode switching problem is shown to be solvable in [KPS01]. A control mode graph is formulated, and from there the exploration can be conducted in the semantics of directed graphs.

Maneuver Sequence Synthesis The goal of this application is to synthesize the parameters of a maneuver. An example would be finding a sequence of flight modes and the condition for switching between them that would result in a maneuver that is proven to be within the safety limits of the particular model.

Routing Sequence Verification By making assumptions about mobility within a discretized map, the map may be reformulated as a directed graph. Then the existence of a route from an initial to a final node may be verified using discrete reachability computations [KLMS01].

These higher level applications may need to be asynchronous, such as control mode switching synthesis where mode switches are not governed by the global clock. However, they can work in conjunction with an underlying time-based level. This project focuses only on the low level control system, but in doing so in a time-based manner it leaves open the possibility of adding the above or other higher level applications.

3.4 A Second Generation System

As discussed in Section 3.2.3, in this project we choose to develop a time-based system whose overall physical behavior can be analyzed and predicted. Our new helicopter UAV system will utilize the dynamic model and control

algorithm presented in Section 2.2 but will strive to ameliorate the faults found in the first generation system. To this end, we need a unified approach to the timing behavior of the elements —sensors, actuators, and computer— of the control system. We believe the key to this unified approach lies in a time-based, modular design:

A time-based design. The system should be time-based in order to allow easy analysis of its closed loop behavior. However, the system must maintain compatibility with existing devices such as sensors, which are not time-based. A clear boundary between the system's synchronous and asynchronous elements must be drawn, and provisions must be made to bridge the gap.

A modular design. The new system must allow the designer to choose from a diverse mix of sensors, actuation schemes, and controllers. The new system must allow a configuration of the same software to run on different helicopters, which may have very different physical dynamics and devices.

Chapter 4

Platform-Based Design Methods

Automation control systems, such as the BEAR helicopters, can be designed with legacy code reuse and safety guarantees, and without deficiencies in subsystem integration. This chapter presents the building blocks that will later be used to design such a system.

The building blocks we use are those of platform-based design. The main tenet of platform-based design is that systems should employ precisely defined layers of abstraction through which only relevant information is allowed to pass. These layers are called *platforms*. For example, a device driver provides a layer of abstraction between an operating system and a device. This layer hides most of the intricacies of the device, but still allows the operating system to configure, read from, and write to the device. Designs built on top of platforms are isolated from irrelevant subsystem details. A good platform provides enough useful information so that many applications can be built on top of it. For example, the C programming language, despite its flaws, provides an abstraction of instruction set architectures that is versatile enough to allow many applications to be written in C.

A system can often be usefully presented as the combination of a top level

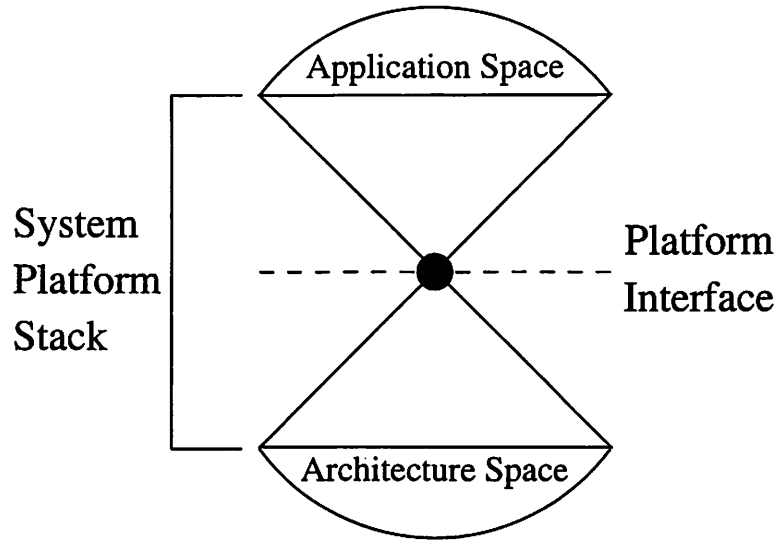


Figure 4.1: The system platform stack

view, a bottom level view, and a set of tools and methods to map between the views. On the bottom, as depicted in Figure 4.1, is the architecture space. This space includes all of the options available for implementing the physical system. For example, a PC can be made from a CPU from Intel or AMD, motherboards from a variety of vendors, etc. On the very top is the application space, which includes high level applications for the system and leaves space for future applications. These two views of the system, the upper and the lower, should be decoupled. Instead of interacting directly, the two design spaces meet at a clearly defined interface, which is displayed as the shared vertex of the two triangles in Figure 4.1. The thin waist of this diagram conveys the key idea that the platform exposes only the necessary information to the space above. The entire figure, including the top view, the bottom view, and the vertex, is called the *system platform stack.*

The platform-based design process is a "meet-in-the-middle" approach, rather than being top-down or bottom-up. Top-down design often results in unimplementable requirements, and bottom-up design often results in a mess. In platform-based design, a successive refinement process is used to determine the

abstraction layer. In this process, an initial application design helps to define a provisional platform interface. This platform interface in turn suggests what the architecture implementation needs to provide. The architecture space can then be explored to find an implementation that comes closest to satisfying both the platform interface and the preset physical requirements. The platform interface may need modification, and the application design may need some rethinking. This process repeats until an appropriate platform interface has been defined. At this point the platform interface is a reasonable and well-specified point of contact between the application and architecture spaces. As a result, new applications may be developed to use the same platform, and new architectures may be explored for future support of the same platform interface. As we have seen, the focus of platform-based design is the correct definition of the platform interface, a process that may involve feedback loops.

Chapter 5

A Time-Based Control Platform: Giotto

Control laws for autonomous vehicles are typically implemented on top of microprocessors using an RTOS and device drivers. These software implementations of control laws have real-time constraints: the code running the control laws must execute periodically, or at some minimum rate. However, it is difficult to ensure that these constraints are satisfied using only the tools provided by the RTOS and device drivers. The main problem is that the only scheduling parameter available to the programmer is the choice of priority for each task. The programmer therefore must be concerned with complicated issues of scheduling such as priority inversion. Furthermore, the programmer is responsible for analyzing the overall schedulability of the system by hand. A second difficulty is that the array of possible inter-task communication schemes supported by the specific RTOS all require detailed setups. Due to these difficulties the development of the control laws, and the scheduling of the embedded software tasks are often constructed entirely in isolation from each other. This results in problems ranging from transient timing errors to costly but necessary redesigns.

In order to ameliorate these difficulties, we introduce a new abstraction layer which will sit between the RTOS and the functional description of the control laws. This abstraction layer will provide the control designer with a more relevant and very simple method for programming the control laws to meet real-time constraints. However, the control designer must adhere to the simple guidelines that this abstraction allows. In this way, the abstraction layer will restrict the design space available to develop the control laws, but will significantly shorten the time to market and increase the correctness of the design.

To illustrate this idea using the hourglass platform-based design figure, we place the possible control laws in the application space on the top, and the RTOS in the architecture space on the bottom as shown in Figure 5.1. The proposed abstraction layer makes up the interface between these two views. Ideally, this platform interface should pass the timing constraints of the application downwards, and should pass the performance capabilities of the architecture instance upwards. On the basis of these constraints, the platform's tools should be able to determine if the timing requirements of the application can be fulfilled. In this section, we discuss in detail an abstraction layer between the RTOS and the real-time control laws that is used in the helicopter embedded software. This abstraction layer is the Giotto programming language.

Giotto consists of a formal semantics [HHK01b] and a retargetable compiler. Giotto has already been used to reimplement the control system onboard a small autonomous helicopter developed at ETH Zürich [KSHP02]. In this chapter, we first present a brief introduction to Giotto (Section 5.1). We then discuss the tools that may be used to map a Giotto application to its possible implementations (Section 5.2). Finally, we compare Giotto with related technologies (Section 5.3). The reader wishing a more detailed introduction should consult [HHK01a].

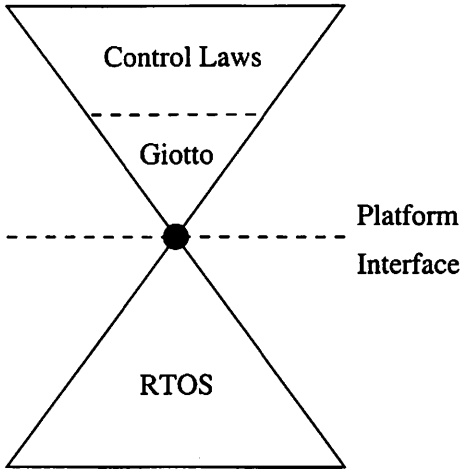


Figure 5.1: The Giotto programming language as a platform interface

5.1 The Giotto Programmer's Abstraction

In this section, we discuss the abstraction that Giotto presents to the programmer. Control applications often have periodic, concurrent tasks. For example, the helicopter control application runs a measurement fusion task at a frequency of 100 Hz, and a control computation at 50 Hz. Typically, the periodic tasks communicate with each other. The mechanism used to implement such communication —whether message queues, shared memory, or some other mechanism— may vary depending on the operating system. Control applications also need a means to input from and output data to their physical environment. Finally, control applications often have distinct *modes* of behavior; in two different modes, different sets of concurrent tasks may need to run, or the same set of tasks may need to run but at different rates. For example, a robot on a discovery mission may first need to run one set of tasks to navigate to a location; once that location is found, the robot may need to run a different set of tasks to query its surroundings. Giotto provides the programmer a way to specify applications with periodic, concurrent, communicating tasks. Giotto also provides a means for I/O interaction with the physical environment, and for mode switching between different sets of tasks.

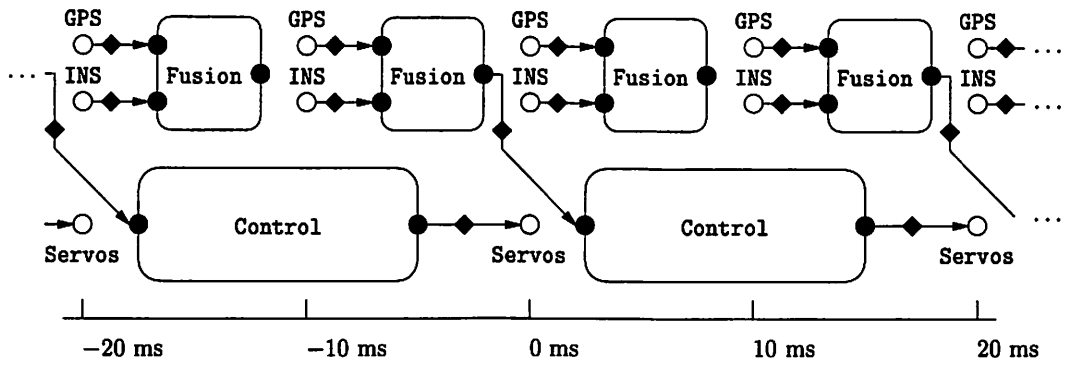


Figure 5.2: An example Giotto program

Consider the example program of Figure 5.2. The concurrent tasks — `Fusion` and `Control`— are shown as rectangles with rounded corners. Each task has a logical execution interval. In our example, `Fusion` logically executes from 0 ms to 10 ms, from 10 ms to 20 ms, etc., whereas `Control` logically executes from 0 ms to 20 ms, from 20 ms to 40 ms, and so on. Each task has *input ports* and *output ports*, shown as black circles. A task's input ports are set at the beginning of its logical execution interval. During its execution, the task computes some function, and the results are written to its output ports at end of its logical execution interval. For example, the input ports of `Fusion` are set at 0 ms; between 0 ms and 10 ms, `Fusion` computes its function; at 10 ms, the result of this function is written to `Fusion`'s output ports.

A Giotto program may also contain *sensors* and *actuators*, both of which are depicted as white circles. Rather than being actual devices, sensors and actuators are programming language constructs which let the programmer define how to input data to and output data from a Giotto program. Logically, sensors and actuators are passive: they are *polled* at times specified in the Giotto program, and cannot push data into the program at their own times. Our example program has two sensors, `GPS` and `INS`, and one actuator, `Servos`. The sensors are read at 0 ms, 10 ms, 20 ms, etc., and the actuator is written at 0 ms, 20 ms, and so on.

Tasks communicate with each other, and with sensors and actuators, by means of *drivers*, which are shown as diamonds. In Figure 5.2, the drivers connect the `GPS` and `INS` sensors to the input ports of the `Fusion` task. They also connect the output port of `Fusion` to the input port of `Control`, and the output of `Control` to the `Servos` actuator. Thus, the `Fusion` task which executes between 0 and 10 ms receives its inputs from the `GPS` and `INS` readings at 0 ms. Similarly, the `Control` task which starts at 0 ms receives its inputs from the `Fusion` task which finishes at 0 ms, and writes its outputs to the `Servos` actuator at 20 ms.

In order to carry out scheduling and analysis, Giotto does require some of the information about the control code that was described in Section 3.3.1. Specifically, the desired period and elusive WCETs for each task must be given by the programmer.

In this section, we have described the abstraction that Giotto presents to the programmer. In the next section, we will discuss the Giotto compiler, which transforms Giotto programs into real-time operating system applications.

5.2 Tools to Implement Giotto

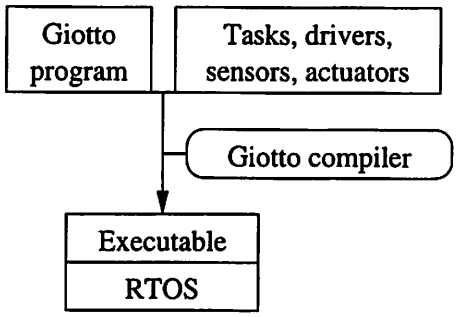


Figure 5.3: Design flow for the Giotto compiler

The platform-based design methodology advocates the use of tools to map

from high level abstractions to the underlying architecture. Here, the Giotto language is the abstraction, and real-time operating systems (RTOS) constitute the architecture. This section describes the Giotto compiler, which maps Giotto programs to RTOS executables. Just as a conventional C compiler transforms C programs into object files for an instruction set architecture, the Giotto compiler transforms Giotto programs into object files for a real-time operating system (see Figure 5.3).

The input to the Giotto compiler is a Giotto program, together with code to implement the tasks, drivers, sensors, and actuators. These other pieces of code may be written in a conventional programming language such as C. These pieces of code are annotated with worst-case execution times. In effect, these annotations allow constraints to pass upwards from the architecture to the platform. The Giotto program also specifies timing constraints that pass downwards towards the architecture. Using both sets of constraints, the compiler performs schedulability analysis, which ensures that all deadlines in the executable it produces will be met. The compiler then generates an object file which can be run on any RTOS. This object file contains instructions for the Embedded Machine, which is an RTOS-independent virtual machine [KH01]. At run-time the Embedded Machine sequences and schedules the tasks, drivers, sensors, and actuators of the Giotto program.

An RTOS typically supports applications with multiple threads of control, whether they be called threads, processes, or tasks. In addition, an RTOS usually provides a means for scheduling these threads, whether by priorities, deadlines, or round robin. The Giotto compiler aims to make efficient use of these RTOS services. The Giotto compiler currently uses heuristics for developing a pre-runtime schedule: drivers, sensors, and actuators are executed at the fixed times given by the Giotto program, whereas tasks are scheduled using earliest deadline first. For example, in the program of Figure 5.2, `GPS` and `INS` are executed at 0 ms, 10 ms, and so on, and the deadline of `Fusion` is always 10 ms after its start time. Using the techniques of [LH95], the Giotto compiler checks

that, in the schedule it has developed, all deadlines are met. Work is currently underway to enable the Giotto compiler to schedule programs for which its present heuristics are not sufficient, using techniques similar to [Bła76, CSB90].

5.3 Giotto Compared to Related Technologies

The services provided by Giotto are similar to those provided by other technologies with which the reader may be familiar. In this section, we compare Giotto to real-time operating systems, the synchronous programming languages, and the time-triggered architecture, in order to clarify the niche that Giotto is meant to occupy.

As discussed in section 5.2, Giotto tasks are transformed by the Giotto compiler to operating system threads. At run-time, these threads are scheduled by a real-time operating system. Thus, the Giotto programmer's abstraction could be viewed as similar to the abstraction provided by a real-time operating system. However, standard real-time operating systems do not provide integrated schedulability analysis. It is up to the programmer to perform such analysis on her own. In addition, real-time operating systems commonly provide many styles of inter-task communication. Some of these, e.g. shared memory, can be tricky to program. In contrast, Giotto provides only a single communications semantics, but automates its implementation.

The synchronous programming languages are a family of programming languages that have been under development since the 1980s. Both Giotto and synchronous languages try to reduce the unpredictable effects of concurrency. The same general approach is taken by both: all activities of the program may be dated on a single timeline. Informally, a *timeline* is a sequential ordering of the activities of a program. In a traditional multithreaded application, each thread has its own timeline. These timelines may be interleaved in many pos-

sible ways depending on the operating system scheduler and the inputs from the environment. In contrast, a synchronous program or a Giotto program specifies exactly one way to interleave the timelines of the program's components. Thus, programs written in Giotto and the synchronous languages are more deterministic than those written for traditional multithreaded operating systems. Software implementations of synchronous programs are typically single-threaded; preemption is not a feature of these implementations. This makes for potentially inefficient CPU utilization. In contrast, since Giotto programs are multi-threaded, they can incorporate preemption, and thus fuller CPU utilization.

The time-triggered architecture (TTA) is a hardware and software system which provides fault-tolerant time-based services. The TTA consists of specialized boards which communicate using its own time-based communication protocols. In contrast, Giotto concentrates on providing an *abstract* programmer's interface. The TTA's time-based nature makes it particularly suitable for running Giotto. However, Giotto can also be run in other hardware and software environments. For example, Giotto is run on hardware and software designed at ETH Zürich, and is also run (without real-time guarantees) on Linux.

Chapter 6

Design of Helicopter UAV Embedded Software

In this section we discuss strategies for building a helicopter based UAV, with two main goals in mind.

1. The first goal is to incorporate both asynchronous input devices and a time-based controller. In Chapter 2 we saw that the sensors send data at their own, possibly drifting, rates. We also presented the advantages of using a time-based controller. However, in Chapter 5 we saw that our chosen time-based controller reads from input devices at its own fixed times. Thus, combining these components gives rise to a mismatch in timing behavior which needs to be addressed.

2. The second goal is to build a system that is modular enough to allow one suite of devices (e.g., a sensor suite) to be replaced by another.

To achieve these two goals we will use the principles of platform-based design presented in Chapter 4. We will show how the insertion of a layer of abstraction between the devices and the controller can be used to bridge the timing

mismatch and allow for the inclusion of different sensor suites.

In Section 6.1 the platform-based design principles are used to specify a functional description of the helicopter based UAV. In Section 6.2 we describe the process of implementing the functional description. Finally, in Section 6.3 we discuss how to compare implementation alternatives.

6.1 Building Functional Description using Platform-Based Design

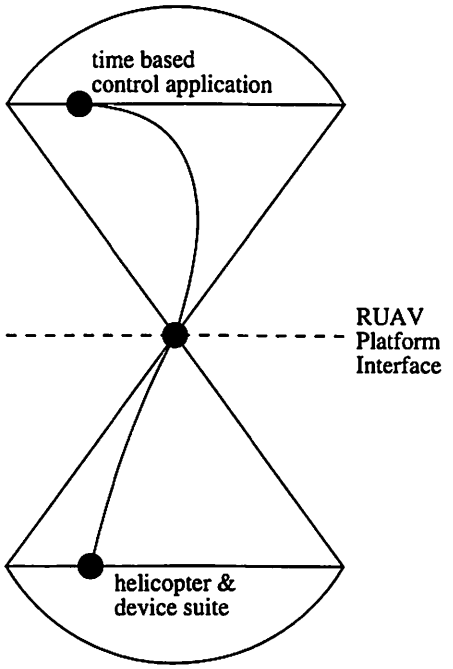


Figure 6.1: Platform-based design of helicopter based UAV

In Chapter 4 we explained how to begin the platform-based design process by separating the system into two views: the application and the architecture. Here we apply this separation to our helicopter based UAV, which is naturally seen from two views. From the top, a designer sees the time-based control

application. From the bottom, a designer sees the available physical devices, such as the helicopter, the sensors, and the actuators. Figure 6.1 situates these two views in the context of platform-based design: the time-based control application sits in the application space, while the physical devices make up the architecture space. Following the *meet-in-the-middle* approach of platform-based design, we include an intermediate abstraction layer, the UAV platform, whose top view is suitable for time-based control and whose bottom view is implementable using the available devices.

We next describe the functionality of the UAV platform.

Interaction with devices. The UAV platform should be able to receive transmissions from the sensors at their own rates and without loss of data. Similarly, the platform should be able to send commands to the actuators in the correct formats. It will also need to initialize the devices. Furthermore, the platform should be able to carry out these interactions with a variety of different sensor and actuator suites.

Interaction with control application. The UAV platform should provide measurement data to the control application in the format and at the frequency dictated by the controller. Similarly, the platform should receive the commands from the controller at times dictated by the controller, and immediately send them on to the actuators. The platform should also be able to support a variety of controllers.

One natural conclusion is that the platform should *buffer* incoming data from the sensors, *convert* sensor data into formats usable by controller applications, and convert control commands into formats usable by actuators. In Section 6.2 we describe in detail two ways to implement the functions of the platform.

6.2 Implementing Functional Description using Platform-Based Design

While the platform-based design methodology is a meet-in-the-middle approach, it suggests implementing the application first. In this section we begin by discussing the realization of the controller application. This implementation, as discussed above in Section 6.1, places constraints on the platform. Platform implementations which meet these constraints are presented next. Though the platform is constructed to work with a variety of available devices, we work with only one such architecture instance. Comparing the efficacy of alternate sensors and actuators is beyond the scope of this dissertation.

6.2.1 Implementing the Controller Application

To attain the benefits of time-based control, presented in Section 3.4, the controller application is realized using the Giotto programming language, detailed in Chapter 5. Section 5.1 presented a rough sketch of the Giotto implementation in Figure 5.2. The two essential tasks are `Fusion` and `Control`. `Fusion` combines the INS and GPS data using a Kalman filter and is run at a frequency of 100 Hz. `Control` uses the output from `Fusion` to compute the control law shown in Section 2.2 at a frequency of 50 Hz. The frequencies of these two tasks are chosen based on the expectations of the control law and on the limitations of the devices. Each task is written as a separate C function. These C functions are referenced inside of the Giotto program, which schedules and runs them as described in Section 5.2.

Unfortunately, the advantages of using such a time-based controller application —in particular, reduced jitter— trade off with the disadvantage of increased latency. For example, in Figure 5.2, consider the `Control` instance which executes from 0 to 20 ms. The incoming sensor data for this instance

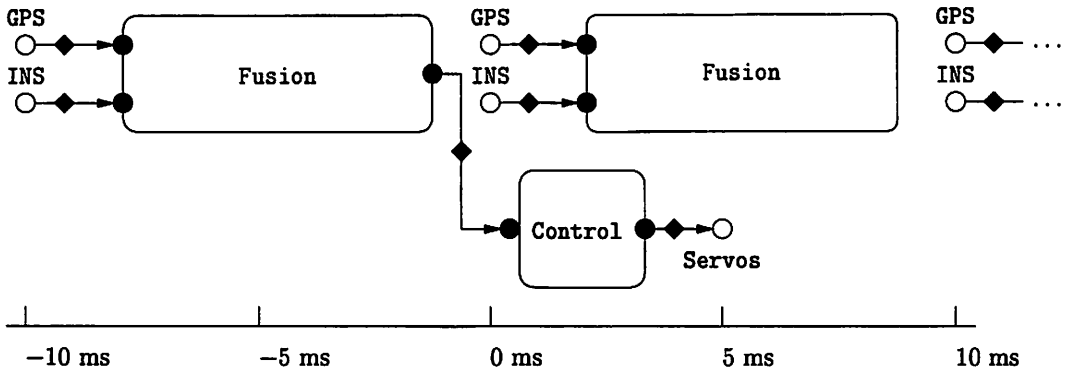


Figure 6.2: Refined Giotto program

was sampled at −10 ms. At 0 ms the data has been transformed by `Fusion` and is ready for use by `Control`. The output of `Control` is not written to the actuator until 20 ms, resulting in a total latency of 30 ms. This is unfortunate, since a new output of `Fusion` is available at 10 ms. In fact, the actual execution time of the `Control` task is less than 10 ms, so `Control` should ideally be scheduled *after* `Fusion` has made a new output available at 10 ms.

One way to reduce the latency of the program of Figure 5.2 is to increase the frequency of `Control` to 200 Hz, so that its deadline reduces to 5 ms. However, this results in `Control` being executed unnecessarily often. Instead, we wish to execute `Control` only once per 20 ms interval, but to retain the 5 ms deadline. Achieving this result in Giotto is possible, with a little extra effort. We first note that each Giotto driver is equipped with a *guard*, which is a condition on the driver's input ports. If the guard of a task driver evaluates to true, the task is executed, but if it evaluates to false, the task is not executed. To fix our problem, we add a counter which is incremented every 5 ms, and we add a guard to the driver of `Control` which evaluates to true when the counter equals 0 mod 4. `Control` thus executes from 0 to 5 ms, from 20 to 25 ms, and so on. The refined Giotto implementation with reduced latency is displayed in Figure 6.2.

6.2.2 Implementing the UAV Platform

Having considered a realization of the time-based controller, we now turn to the UAV platform. In Section 6.1, we discussed the requirements that our UAV platform needs to fulfill. We now present two possible implementations of the UAV platform, both of which fulfill these requirements. The first implementation uses one computer, effectively implementing in software the buffer discussed in Section 6.1. The second uses two computers, and implements the buffer in hardware.

First implementation: one computer.

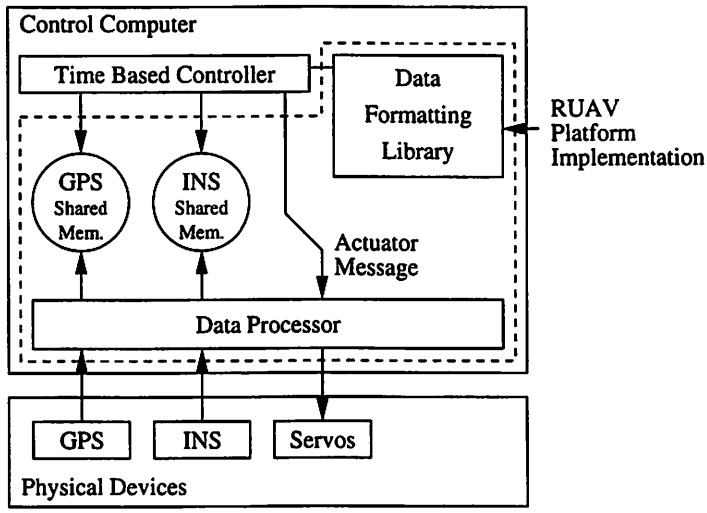


Figure 6.3: First implementation of UAV platform

The single computer implementation has three main elements, which are depicted in Figure 6.3.

Data processor. The data processor is an independent process, similar to a standard interrupt handler. In the sensing case, it responds to the new sensor data sent by the devices, and saves this data to a shared mem-

ory space with the sensor specific data format intact. In the actuation case, the data processor passes on to the servos the messages sent by the controller application.

Shared memory. The shared memory contains recent sensor readings, and is implemented as circular buffers. Data are placed into the circular buffers by the data processor, and can be accessed by the controller application. In this way the controller application can grab the sensor data without worrying about the timing capabilities of each sensor.

Data formatting library. Within the controller application, the sensor specific data format must be transferred to the format that the control computation expects. In the sensing case, the controller application uses the data formatting library to transform the buffered sensor readings. In the actuation case, the controller application uses the library to convert actuation commands into the format expected by the servos.

Recall from Section 5.2 that the controller application comes with guarantees about the deadlines of its own internal tasks. These guarantees, however, do not take into account the time that may be needed by other processes or interrupt handlers. If more than a "negligible" amount of time is spent in the other processes, then the timing guarantees of the controller application may cease to be valid. For this reason, the above design keeps the time needed by the data processor to a bare minimum. The data transformations necessary are instead written into the data formatting library and called from within the control tasks. The benefit of this approach is that the timing guarantees of the controller application are preserved, as much as possible.

Second implementation: two computers.

Though the single computer implementation results from a platform-based design methodology, one might well argue that it does not adhere to a strict

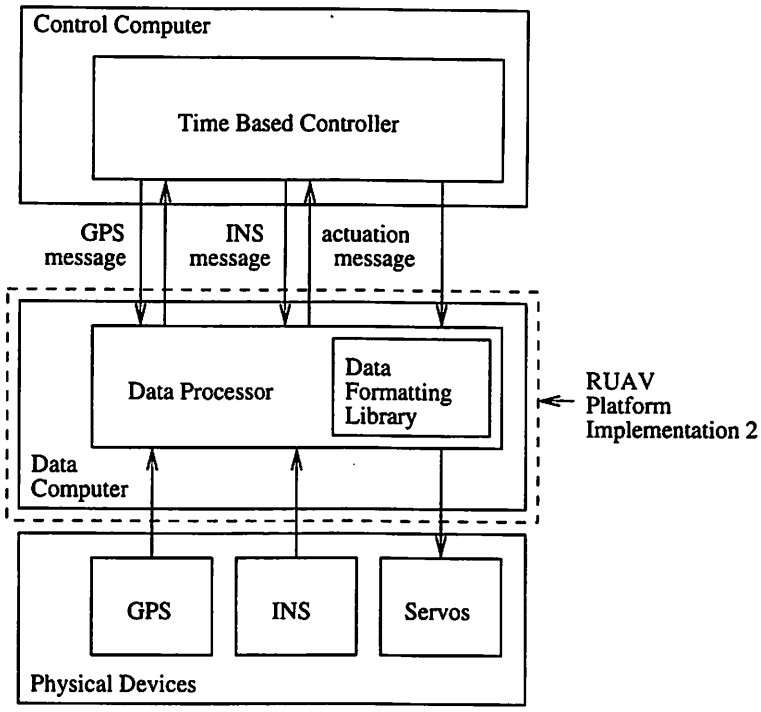


Figure 6.4: Second implementation of UAV platform

separation of the control from the sensor details. This problem results from the fact that the format conversion functions are run from within the controller. We have argued that this is needed to preserve the guarantees on the timing of the controller application. In a second implementation, both the timing guarantees and the separation of control from sensor details are maintained by including two computers on the helicopter. This alternative is depicted in Figure 6.4. The two computers perform distinct functions:

Control computer. The control computer runs the controller application. When the application needs the most recent sensor reading, it sends a request to the data computer. The application also forwards actuator commands to the data computer.

Data computer. The data computer performs the same functions as the data processor and data formatting library from the first implementation. It

receives readings from the sensors. When the control computer requests the most recent reading, the data computer replies with this reading in the correct format. When the control computer sends an actuator command, the data computer relays the command to the servos in the correct format.

In this implementation the separation of control from sensor details is strictly followed, and the timing guarantees of the controller application are maintained. There is a tradeoff, however. The downside to this second approach is an added amount of latency that is introduced between the time the sensor readings are taken and the time the control laws use the measurements. This latency is introduced by the communication between the control computer and the data computer. This increase in the staleness of the data is a common tradeoff with more structured designs. In the next section, we discuss methods for a quantitative comparison of the two designs.

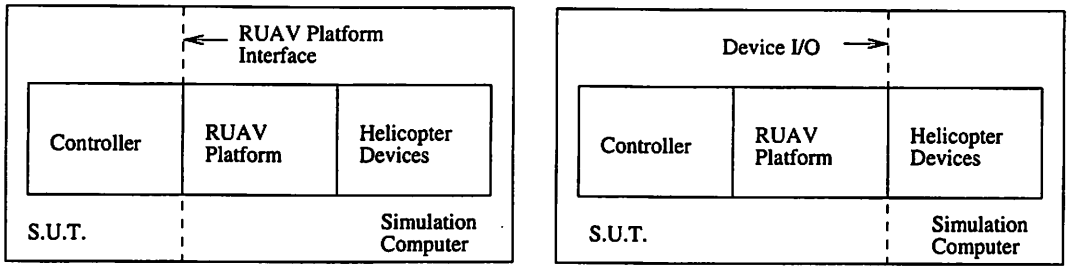
6.3 Comparison of Implementation Alternatives

Now that we have two platform implementations the next question is natural: which one is the best? Ideally, carefully controlled tests could be performed on the physical system. However the fact that we are working with an automation control system makes that a difficult proposition:

- Testing is expensive and potentially dangerous. For the helicopter, a safety pilot must be on hand for every test run in case a takeover is necessary.
- Tests are difficult to standardize. For example, the winds and GPS signal strength cannot be controlled.

To ameliorate this problem, we propose the use of a hardware-in-the-loop simulator, which allows for the direct testing of the entire control system. Instead

of mounting the control system onto the helicopter, the controller (often called the *system under test*) is connected to a simulation computer. The simulation computer uses a dynamic model to mimic the exact inputs and outputs of the sensors and actuators on the helicopter.



(a) HIL simulation for testing controller

(b) HIL simulation for testing RUAV Platform

Figure 6.5: Hardware-in-the-loop simulation: S.U.T. refers to the *system under test*

Hardware-in-the-loop simulators [SS01, Led99] are well suited to take advantage of the abstraction layers provided by platform-based design. The suitability arises from the capacity to slide back and forth the dividing line between the simulation computer and the system under test, as shown in Figure 6.5. To compare the controller applications, the simulator should act as the platform interface, and the controller applications should act as the system under test. To compare platform implementations, the simulator inputs and outputs should closely approximate those of the actual devices, and the controller application and platform implementation should be part of the system under test.

Due to the fact that the simulation computer must imitate a physical system, the simulator must meet two additional constraints:

- The simulator must run in real time. This greatly limits the choice of operating systems available to run the simulator. It also mandates a careful selection of the numerical methods used for solving the model's differential equations [Led99].

- The simulated helicopter should faithfully duplicate the dynamics of the real world helicopter. The parameters of the simulator should be set to values that have been measured on the helicopter. To check that the simulator software mathematically implements the behavior of the physical models, we propose the use of system identification techniques. The parameters of the mathematical model should be compared with those obtained using system identification on the input-output behavior of the hardware-in-the-loop simulator.

The proposed simulation framework, in combination with platform-based design, allow for the development of automation control systems that are modular and have guaranteed performance.

Chapter 7

Hardware-in-the-Loop Simulation System

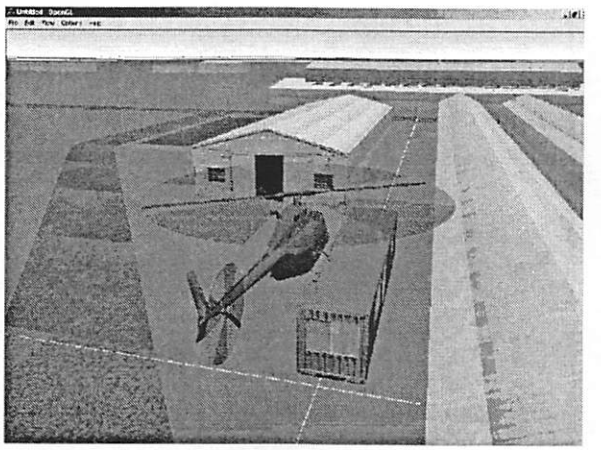


Figure 7.1: Graphical flight display

The work of building the hardware-in-the-loop simulation framework and the accompanying control computer was broken down into two main stages to aid development. For the first cut, the entire system was developed on a Linux operating system. This allowed for a simple means of inter-task communication and network communication, and a robust development backbone. Of course,

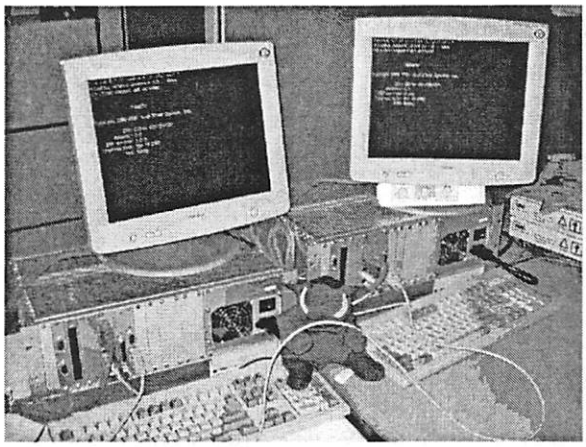


Figure 7.2: Flight capable hardware with real-time operating systems

Linux is not a real-time operating system and is therefore unsuitable for the final implementation. Furthermore, because this is a hardware-in-the-loop simulation, the hardware the controller runs on should be identical to the onboard computer. After this initial development stage, we ported the software to flight capable computers running VxWorks. The details of each build are provided in this chapter.

In Section 7.1 we present components that are pervasively used in both builds, such as the dynamic model and the simulation framework. Section 7.2 delves into a discussion of the implementation of the development system running on Linux, and Section 7.3 discusses the final hardware-in-the-loop simulation system implementation.

7.1 General Simulator Properties

7.1.1 The Dynamical Model

The simulator uses the practical dynamic model for the Yamaha R-50 presented in Section 2.2.1, equation(2.2). The presented numeric matrices **A** and **B** are the main constituents of this model. The state vector associated with this model (equation(2.3)) denotes the quantities in body coordinates, where the origin is tied to the center of mass of the rigid body. Since we would also like the simulator to compute the position of the helicopter in tangent plane coordinates, $\mathbf{X^{TP}}$, and the non-linearized orientation, $\mathbf{\Gamma}$, the linear model is supplemented by the first and third sections of equation(2.1). These two added nonlinear equations can compute $\mathbf{X^{TP}}$ and $\mathbf{\Gamma}$ from the state variables. They are shown here in their expanded form:

$$\frac{d}{dt}\begin{bmatrix}\Phi\\ \Theta\\ \Psi\end{bmatrix} = \begin{bmatrix}1 & \sin(\Phi)\tan(\Theta) & \cos(\Phi)\tan(\Theta)\\ 0 & \cos(\Phi) & -\sin(\Phi)\\ 0 & \sin(\Phi)/\cos(\Theta) & \cos(\Phi)/\cos(\Theta)\end{bmatrix}\begin{bmatrix}p\\ q\\ r\end{bmatrix} \tag{7.1}$$

$$\frac{d}{dt}\begin{bmatrix}x^{TP}\\ y^{TP}\\ z^{TP}\end{bmatrix} = \begin{bmatrix}\cos\Psi\cos\Theta & -\sin\Psi\cos\Phi+\cos\Psi\sin\Theta\sin\Phi & \sin\Psi\cos\Phi+\cos\Psi\sin\Theta\cos\Phi\\ \sin\Psi\cos\Theta & \cos\Psi\cos\Phi+\sin\Psi\sin\Theta\sin\Phi & -\cos\Psi\sin\Phi+\sin\Psi\sin\Theta\cos\Phi\\ -\sin\Theta & \cos\Theta\sin\Phi & \cos\Theta\cos\Phi\end{bmatrix}\begin{bmatrix}u\\ v\\ w\end{bmatrix} \tag{7.2}$$

The concatenation of these model equations make up the dynamic model that the simulator uses. With these additional state variables: the nonlinear orientation angles (Φ, Θ, Ψ), and the position (x^{TP}, y^{TP}, z^{TP}), the final dynamic model state contains 18 elements.

7.1.2 The Simulation Framework

The original idea was to use the Simulink toolbox from MATLAB to create the simulator, and to translate it to C code using Real-Time Workshop. In general terms, the Simulink simulator was made up of one feed-through S-function containing the differential equations of the above model, and one integration step. The Simulink block diagram is displayed in Figure 7.3.

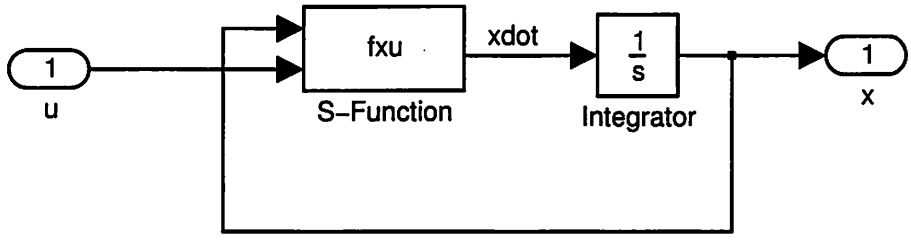


Figure 7.3: Diagram of simulator in Simulink blocks

The main problem with this approach was that we wanted the generated C code for the simulator to be able to interact, using the network, with a controller process. These input/output requirements ended up necessitating various modifications, or hacks, within the Real-Time Workshop framework. Even with the extensive I/O patches, the combination of Simulink with Real-Time Workshop proved to be too inflexible for our purposes.

After the failure of the tidy approach presented by Simulink and Real-Time Workshop, various simulation techniques were weighed. The most important characteristics necessary for our application were that the simulator framework be:

- straightforward
- flexible
- efficient

- portable to a real-time operating system
- convenient.

The choice that maximized the above characteristics was to custom build the simulator with hand written C code. Our C implementation makes use of a numerical recipes software package. The chosen package provides a suite of C functions that provide the programmer with a practical and efficient means of implementing mathematical tasks [num92]. This package aims to make the functions portable across different operating systems and different compilers, which makes the job of porting between Linux and VxWorks easily possible. In fact, the programs are "guaranteed" to run without modification on any compiler that implements the ANSI C standard. The chosen package does require a strict adherence to a rather cumbersome definition of arrays and 2-dimensional arrays, but this is a small price to pay in return for simplicity and efficiency.

7.1.3 ODE Solution Methods

The main use of the numerical recipes package is to solve the ordinary differential equations of the dynamic model so that the state vector can be updated based on the new inputs to the system at each time step. This objective boils down to an *initial value problem* where the starting point of the state variable $\mathbf{x_i}$ is given and it is desired to find some final point $\mathbf{x_f}$ or some discrete list of points (i.e. at tabulated intervals). All solutions to this pervasive initial value problem are based on the same idea: the $dx's$ and the $dt's$ are thought of as finite steps Δx and Δt. Now one can multiply by Δt, algebraically formulating the change in the function as a result of stepping by one "stepsize" Δt. A good approximation can be achieved with this methodology by making the stepsize very small. The simplest and most literal implementation of the above description is known as *Euler's Method*, which makes use of the following

equation:

$$x_{n+1} = x_n + \Delta t f(t_n, x_n) \tag{7.3}$$

This method, unfortunately, is too simple to achieve a practical approximation. The main problem is that the formula is asymmetrical in that it advances the solution through an interval in time, but only uses the derivative information dx recorded at the start point of the interval. Instead, other methods use the idea of a "trial" step to the midpoint of the time interval. The values of t and x are again evaluated at this midpoint and the combined set of information produces a more accurate total step across the whole interval. A very widely applicable and generally efficient algorithm by the name of *Runge-Kutta* builds upon the simple Euler framework in this manner. The Runge-Kutta method propagates a solution over an interval by combining the information from several Euler-style steps. The information attained is then used to match a Taylor series expansion up to some higher order. The classical fourth-order Runge-Kutta formula can be expressed as follows:

$$\begin{aligned}
k_1 &= \Delta t f(t_n, x_n) \\
k_2 &= \Delta t f(t_n + \frac{\Delta t}{2}, x_n + \frac{k_1}{2}) \\
k_3 &= \Delta t f(t_n + \frac{\Delta t}{2}, x_n + \frac{k_2}{2}) \\
k_4 &= \Delta t f(t_n + h, x_n + k_3) \\
x_{n+1} &= x_n + \frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6} + O(\Delta t^5)
\end{aligned} \tag{7.4}$$

It is apparent from the above equation that the fourth-order Runge-Kutta method requires four evaluations of the function per step size Δt.

Other popular solutions include the *Bulirsch-Stoer* method and predictor-corrector methods. These rely more heavily on past data and are therefore

suitable for a smaller class of problems. They are, however, more efficient for problems that fall within their domain.

For solving ordinary differential equations in real-time, Ledin recommends the use of algorithms which use a fixed step size and require inputs for derivative evaluation that are available at the current time or earlier [Led99]. The simplest algorithm satisfying these constraints is the aforementioned Euler method. A more accurate method that is apparently widely used within the real-time community is the *Adams-Bashforth* second order algorithm. This algorithm uses both the current frame's derivative, and the previous frame's derivative. Unfortunately, numerical C recipes for the Adams-Bashforth algorithm are not commercially available, and the Euler method is not suitable for the reasons discussed above.

The problem with using the Runge-Kutta algorithm for real-time work stems from the fact that the Runge-Kutta assumes that a derivative can be taken at any instant in between time steps. However, in a real-time environment, the data might not be ready for sub-time step evaluation. To deal with this complication, we implement a modification of the Runge-Kutta algorithm which is compatible with real-time inputs. In our implementation we approximate the sub-time step derivatives by using the most recent control data.

7.1.4 Numerical Recipes in C

The numerical recipes package provides a simple fourth-order Runge-Kutta function which carries out one classical Runge-Kutta step on a set of n differential equations. Therefore, we input the values of the independent state variables, and we get out the new values which are stepped by a stepsize of Δt. The function also requires as input a function `derivs` that states the differential equations in a particular format. The function declaration is shown in Figure 7.4. Given values for the variables `x[1..n]` and their deriva-

```
void rk4(float x[], float dxdt[], int n, float t, float Δt,
    float xout[], void (*derivs)(float, float[], float[]))
```

Figure 7.4: Numerical Recipes fourth-order Runge-Kutta function

tives `dxdt[1..n]` known at `t`, this function uses the fourth-order Runge-Kutta method to advance the solution over an interval Δt and return the incremented variables as `xout[1..n]`. The user supplies `derivs(x,t,dxdt)` which returns derivatives `dxdt` at `t`.

7.2 Development System

The first build of the hardware-in-the-loop simulation framework would be more accurately described as a software-in-the-loop simulation framework. The two halves of the framework: the controller vs. the simulator, both executed on a Linux operating system and communicated via network. The UDP socket interface was chosen as the network connection type because it implements a simple transfer of data without handshaking routines or re-send stipulations that would be absent on the actual helicopter.

7.2.1 Controller

The control law presented in Section 2.2.2 is the algorithm that is implemented in this build. This control law is realized using a simple Giotto program and the initial platform implementation. Since we are using the initial platform implementation, the controller is contained in one computer and a light weight data processor is included.

Giotto Program

As presented in Chapter 5, realizing a control law by writing a Giotto program requires writing the task functions, sensor and actuator ports, and drivers in a host language, here the C language. Each of these bits of code can then be combined by the Giotto semantics to formulate a whole program that executes as desired. We will now take a more in depth view of each component that makes up the Giotto program.

Sensor Port. The sensor port is where data enters the control process. Our implementation uses a static variable to allow for initialization procedures to be run only once. These initializations open for reading the circular buffer containing the sensor data which is sitting in shared memory space. Next, the sensor port code calls `read-sensor-buffer` and stores the received data at the local memory address provided to the sensor port function as an argument.

Drivers. The drivers connect the sensor ports to the input ports of tasks. They also connect the output ports of tasks to the input ports of later tasks, or to actuator ports. Therefore, a driver function is needed to join the sensor port described above, to the input port of the control task. The driver could perform some minimal data transformation, however since it logically executes in zero time, it is best to keep the driver responsibilities to a minimum. In our code, the drivers simply throughput data by copying it from their input to their output arguments.

Tasks. The tasks are the functions that Giotto will schedule. In the first build there was one Giotto task, the control task. The input ports of the control task were connected to the output from the sensor by the throughput driver. The control task first computes the current desired trajectory, $\mathbf{y}_{ref}(t)$, discussed in Section 2.2.2. The desired trajectory is based on the current iteration of the control loop since the maneuver we

demonstrate is an upward spiraling square motion. This trajectory is then used by the control law shown in equation(2.11) which outputs the control directive. This control command is stored at the output argument of the control function.

Actuator Port. The actuator port in our Giotto program is connected to the output of the control task through a throughput driver. The actuator port also uses a static variable to allow for singular initialization, this time of the network socket. Then the actuator port sends the control directive directly over the network to the simulation process.

The Giotto program code ties all of these C functions together, and allows for the description of the task schedule. The Giotto code for the first build is displayed in Figure 7.5.

UAV Platform

The missing piece of the controller process at this point, is how the sensor data ends up in the shared memory circular buffer where the sensor port of the Giotto program reads from. This function is accomplished through the use of the UAV platform. Recall from Section 6.2.2 that the data processor is designed to be as lean as possible so that it will not bias the Giotto compiler's count of available CPU time for scheduling Giotto tasks. For this reason, the data formatting is not done inside of the data processor. Instead separate data formatting C functions are provided as a library to the control programmer to call inside of the functions scheduled by Giotto.

Here, we construct the data processor described in Section 6.2.2 as a server process that simply waits for sensor data to be sent over the network and then immediately places that data in shared memory space. This data processor first initializes each of the network sockets (one for GPS data and one for INS

```
// Sensor ports
sensor
c_input sensor_inputs uses c_get_sensor_inputs;

// Actuator port
actuator
c_output actuator_outputs uses c_send_actuator_outputs;

// Task output ports
output
c_output control_outputs := c_zero;

// Task declarations
task control(c_input control_inputs) output (control_outputs) {
  schedule c_control_task(control_inputs, control_outputs)
}

// Driver declarations
// Input driver for control task
driver control_driver(sensor_inputs) output (c_input control_inputs) {
  if c_true() then c_inputs_to_inputs(sensor_inputs, control_inputs)
}

// Actuator driver
driver actuator_driver(control_outputs) output (c_output outputs) {
  if c_true() then c_outputs_to_outputs(control_outputs, outputs)
}

// Mode declarations
start normal {
  mode normal() period 100 {
    actfreq 1 do actuator_outputs(actuator_driver);
    taskfreq 1 do control(control_driver);
  }
}
```

Figure 7.5: Giotto code implementing the helicopter control laws

data) and each of the shared memory circular buffers. It then uses the `poll` function in order to read data from either socket as soon as it becomes ready. That data is then copied to the appropriate shared memory space.

7.2.2 Simulator

The first simulator build made use of the 18 state dynamical model covered in Section 7.1.1 and of the modified numerical recipes fourth-order Runge-Kutta function introduced in Section 7.1.4. The main elements of the simulator are five separate processes that communicate with each other via shared memory. These elements are:

- Plant process – computes solution to dynamical model ODE's.
- Server process – listens for incoming control directives.
- GPS process – outputs data to controller with format and rate of GPS card.
- INS process – outputs data to controller with format and rate of INS card.
- Display process – outputs data to separate display computer.

We also built a couple of supporting libraries. The astute reader may notice that these libraries are also used by the controller's C functions and data processor described in Section 7.2.1.

- Circular buffer library – implements shared memory as well.
- Network UDP sockets library.

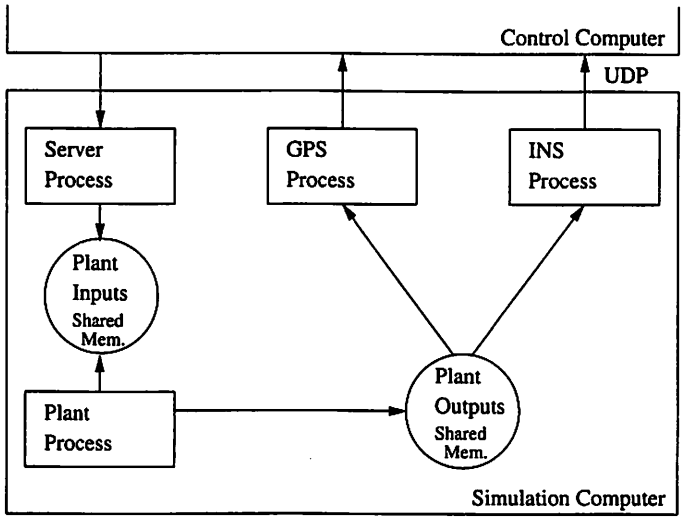


Figure 7.6: Block diagram of simulator processes

This section will focus only on the five main processes and leave the detailed implementation of the supporting libraries to the imagination. While reading through the remainder of this section, refer to Figure 7.6 for the road map of the interconnecting structure.

Plant Process

The plant process communicates only via shared memory and does not send or receive any data via the UDP network sockets. Figure 7.6 displays the plant process as sandwiched in between the server process and the INS and GPS processes, each of which communicates via network with the control computer as well as via shared memory with the plant process. Therefore, the first task of the plant process is to use the circular buffer library to initialize its connection to both its input buffer and its output buffer.

After the initialization phase, the plant process enters an infinite loop that repeats every `TIME-STEP` in which its main responsibilities are fulfilled:

1. The inputs from the controller are copied into local memory from the shared buffer space utilizing the circular buffer library. If there is no data in the shared memory buffer, then the control directives are set to 0 for this time step.

2. The main plant process now calls a supporting plant function that takes as input the control directives and updates the state variable using numerical recipes in C by one `TIME-STEP`.

   This supporting plant function keeps an internal static version of the state. The internal state is made up of the expected 18 elements, and concatenated with the 4 control inputs. The concatenated 22 element array then contains all of the information necessary to formulate the ODE: $d\mathbf{x}/dt = f(\mathbf{x}, t)$. The details of the exact ODE are written according to the numerical recipes standard in the function `derivs` as explained in Section 7.1.4. As covered in Section 7.1.3 the Runge-Kutta modified real-time solution uses the same 4 control inputs from the initial point in time throughout the calculation. The updated 18 element state is returned by the supporting plant function through its second argument.

3. The main plant process then writes the updated 18 element state into its output shared memory buffer using again the circular buffer library.

4. Finally, the process sleeps for one `TIME-STEP`, through the use of the POSIX compliant `nanosecond` command.

Notice that since the main calculations take place in a separate supporting function, the timing of this loop is fairly accurate. Of course, since this is not running on a real-time operating system, timing analysis is close to impossible.

Server Process

The plant server process receives data from the control computer over the network, using UDP sockets, and places that data into shared memory so that the plant process, described above, can retrieve the data. The data in question here are the four control directives that the control computer outputs. The program structure is as follows:

1. First the server's output buffer, the shared memory block between the plant's server and the plant process, is initialized using the circular buffer library.
2. Next, a UDP server type socket is initialized using the network UDP sockets library.
3. After initializations, the server process enters its infinite loop.
4. Within the loop, the first call is a read from the UDP socket. This read is a blocking call, so the server process timing is dictated by the control computer's outputs. This point is the main reason that the plant process and the server process are separate: the plant process must iterate every `TIME-STEP` while the server process iterates whenever data is available from the controller.
5. Finally, the loop ends with a call to write the inputs from the previous UDP read into the shared memory space.

GPS Process

Instead of sending the controller the raw state vector, we'd like to send an exact replica of the real GPS card's outputs. This desire incorporates two main goals:

1. Sending the data at the exact times that the GPS card would send data.

2. Sending the same types of data (i.e. velocity vs. position) with the same formats (i.e. coordinate frame choice) that the GPS card would send. The types and formats of this data might differ from the data provided within the state variable of the dynamical model.

The first goal above is the main motivator for providing a GPS process that is separate from the plant process itself. With the use of a separate process, the data transfer timing can be independent of the `TIME-STEP` used to solve the equations of the dynamical model. The correct GPS data types and formats can be calculated using the most recently updated state vector.

The GPS process first needs to initialize its input and output spaces. The input to the GPS process is the updated state vector that is waiting in shared memory after being calculated by the plant process. The output space is the UDP client socket that will send information to the control computer. After initializations, the main loop is entered:

1. First, the shared memory space is read to acquire the latest updated state vector.

2. Next the state vector is used to compute the exact data type and format of the GPS card that is being modeled.

3. The GPS output data is then send over the UDP sockets to the control computer.

4. Lastly, the GPS process uses **`nanosleep`** to delay before the next period of the loop begins. The delay chosen should correspond to the period of the actual GPS card in the configuration that it will be used, and not to the other simulator processes. In this step, a random but bounded additional time delay could be added to further emulate the function of the real GPS card.

INS Process

The INS process is functionally identical to the GPS process described in Section 7.2.2. The only difference is that in this case the data types, formats, and timing of the INS, not the GPS, are to be emulated.

Display Process

In order to properly view the working combination of the Giotto controller, platform realization, and simulation process, the motion of the simulated helicopter is communicated via network to a graphical flight display program where a 3-D helicopter performs on screen. Figure 7.1 displays a screen shot of this GUI.

The simulator needs a separate process to send the necessary data to the display computer at a rate bounded by the display computer. This display process is very similar to the GPS and INS processes. They all initialize the same type of input and output spaces: the shared memory containing the latest updated state vector and a UDP socket this time communicating with the display computer. Within the main loop the display process reads from the shared memory, sends the data needed by the display computer over the UDP socket, and sleeps until the begin time of its next period.

7.2.3 Using the Development System

The development system did go through many iterations, but was finalized as discussed throughout this section. To run the system, each process must be started individually:

Controller side On the control side the previously compiled Giotto program

is run on the e-machine. The data processor must also be started on the same computer with a call to its main.

Simulator side On the simulator side all of the mentioned processes: the server, GPS, INS, plant, and display, must be started on the same computer.

Display computer The display GUI on the display computer must be started before the simulator has begun for the most robust results.

7.3 Final Hardware-in-the-Loop System

After completing the development system, the next research effort was to port the controller process and platform implementation to the computer which will fly on the helicopter and run a real-time operating system. This control computer will then be the final embedded control system and is the hardware that is 'in-the-loop' within the simulation framework. As discussed in Section 6.3 the simulator process also needs to run on a real-time operating system. Therefore, both pieces of the hardware-in-the-loop simulation framework, the control system and the simulation system, have been ported to the flight capable hardware running a real-time operating system shown in Figure 7.2. The hardware-in-the-loop simulator can be used to examine the comparative performance of the second platform implementation as well as alternative control algorithms. In addition, the chosen Giotto controller and platform realization that reside on one or more flight computers can be flown on the physical helicopter with only sensor and actuator suite specific alterations.

In this section we discuss only the changes from the development system that were needed to re-build the hardware-in-the-loop simulator and control system on a real-time operating system. In fact most of the processes, tasks, and libraries we were able to reuse almost entirely. The reader should assume

the same basic structure and coding for this final system as for the development system, except for the changes mentioned in the sections below.

7.3.1 Real-Time Operating System Configuration

The real-time operating system VxWorks was chosen for use in this project due to its relatively wide-spread industrial use and competent customer support. VxWorks falls into the class of operating systems which require the use of separate development and target computers. The target computer is the computer that will boot the VxWorks image and run the real-time programs. The development computer is where the VxWorks image resides (and can be altered and recompiled) and where the real-time code is written. The target computer therefore boots the VxWorks image residing on the development computer over the network upon startup.

In our case there are two identical target computers (Figure 7.2) and one development computer housing one VxWorks image. One of the target computers is used as the simulator while the other is used as the controller. The development computer, meanwhile, stores the code used to run both the controller and the simulator, and through the use of a GUI, the user can selectively download that code. Also within the GUI there are shells which allow the user to run the code on the target computers.

Adding components, such as a high speed serial card, to the target computer requires a VxWorks driver. This driver is basically a set of patches for the VxWorks image. In this project we wish to utilize a high speed serial line between the control and simulation computers because this is the same mode of communication which will be used between the physical helicopter UAV and the embedded control computer. This addition of the high speed serial card and driver proved to be difficult but ultimately not impossible. The serial connections discussed in the rest of this section refer to these high speed serial

lines.

If VxWorks is the real time operating system that runs on the target computers, then what is Tornado? Tornado is the software system that runs on the development computer and allows the user to write programs, download them, and run them on the target computer. Within tornado there is a very specific file system setup that must be correctly configured [Win99]:

- A *project* consists of a group of source code and include code files and binaries that make up one cohesive application. A project can be compiled to a single object file and downloaded to a target computer.
- A *toolchain* is a set of cross-development tools used for building for a specific target processor. Each project is associated with one toolchain. The toolchains are based on GNU.
- A *workspace* is a grouping of one or more projects that allows for associating related applications. Every project must be built and maintained inside of a workspace.

In our case, two projects are needed. One will be compiled and downloaded onto the control computer target, the other onto the simulation computer target. However, some files are used by both sides: the communication libraries (UDP and serial), the include files that specify the types of the data that is sent from one computer to the other, and the circular buffer shared memory library. In these cases, we have set up the projects so that they both point to the same files. However, the simulator specific and controller specific code is kept separate in the two different projects.

7.3.2 Changes Needed for Real-Time System

Communication

Besides the difference in operating systems and hardware, the most major change between the development system described in Section 7.2 and the final real-time system was the communication scheme between the control computer and the simulation computer. Instead of using a UDP connection written for Linux, we wanted the two computers to interact through the high speed serial lines that the physical actuators on the helicopter would use. Configuring these serial cards and the VxWorks compatible software to run them was more difficult than expected.

A new library for serial communications was written to replace the UDP socket library used on the Linux development system. Unfortunately, VxWorks does not support a `read` command that blocks until the specified number of bytes has been read. It also, by default, translates new line characters sent over the serial into two characters: new line and carriage return. These two obstructions were overcome in the serial communication library.

Another major change necessary was the communication between tasks internally on each computer. In the Linux development system, this communication was handled by setting up shared memory in the circular buffer library. With VxWorks, any address can be accessed by any task (which is not the case in Linux or Unix), so a global variable inherently sits in 'shared memory'. This fact simplified the circular buffer library's creation of shared memory quite a bit.

Finally, the UDP socket library remained relatively intact. Instead of being used for communication between the simulator and the controller, in this version the socket library is only used for communication between the simulator and the display program.

Timing

The Giotto language would ideally run identically on any computing platform, therefore making the switch between Linux and VxWorks transparent. Unfortunately the current Giotto compiler and e-machine has only been developed for Linux at this point in time. Therefore, we needed to write a program that would function identically to the Linux Giotto program.

Timing in tasks other than in the Giotto program was altered. In the development system we used fairly inexact methods such as embedding a `nanosleep` statement within a loop so that the function of that loop would run at *about* the correct frequency. For example, the GPS function discussed in Section 7.2.2 completed several tasks requiring variable amounts of time, and then called `nanosleep` for the exact amount of time desired to create the correct frequency. This does not accurately create the correct frequency due to the time needed to complete the proceeding tasks. In the real-time system we correct these inaccuracies by using both a GPS-main function and a GPS function:

GPS-main This function handles initializing the input and output spaces for the GSP function. It then spawns the GPS function. Finally it enters an infinite loop where it sends a semaphore message to the GPS function and then uses `nanosleep` to delay for the correct amount of time. Since no other computations are done within this loop, the semaphore is sent at regular intervals and the delay time is therefore much more accurate. Furthermore, this task can be assigned a high priority so that its timing will not be altered by other task's preempting it.

GPS This function enters a loop and then waits on the semaphore sent by GPS-main. Upon receiving the semaphore it completes the same steps one, two, and three as the GPS function in the development system, utilizing serial instead of UDP sockets.

A similar method was also used for the INS and plant processes.

7.3.3 Sensors

In the real-time system we incorporated realistic data types. This included changes to both halves of the system, the HIL simulator and the Giotto controller.

Sensors on Simulator Side

The sensor processes on the simulator, GPS and INS, were discussed as part of the development system in Section 7.2.2. In that description we concentrated on the timing of the sensor processes. In the real-time system we incorporated the correct data types to be output from the sensor processes. Therefore we achieved both goals for the GPS and INS processes, independent and realistic timing, and correct output data.

The GPS outputs the positions: x_s, y_s, and z_s in spacial coordinates, both in the flight setup and in our real-time HIL simulator.

The DQI-INS offers many modes of operation. In fact this instrument incorporates a Kalman filter so the outputs of the DQI-INS are already updated with GPS data. Therefore, the first generation flight code could take the velocity data from the DQI-INS and use it without updating with another Kalman filter. The Kalman filter used in the first generation controller system was only to obtain and update position estimates. Again, we use the same data types for the simulated INS to output as the DQI-INS output in the flight setup. Those are: the pre-updated velocities in spacial coordinates, u_s, v_s, w_s; roll rate, pitch rate, and yaw rate, p, q, and r; and roll, pitch, and yaw, Φ, Θ, Ψ.

Sensors on Controller Side

The *position* controller shown in equation(2.11) uses the velocities in body coordinates, u_b, v_b, w_b, the roll, pitch, and yaw, Φ, Θ, Ψ, and the positions in body coordinates, x_b, y_b, z_b. For this controller we need to implement a Kalman filter on the controller side in order to accurately obtain the positions. To run the *velocity* controller shown in equation(8.2), the positions are not needed for control. Therefore, to simply run this controller we do not need an additional Kalman filter, since its inputs are updated by the DQI-INS, and similarly by our HIL simulator. However, in most cases we wish to do position control and need to know accurately what position the helicopter is at.

There are two main stages of the Kalman filter used for position estimation. The first is the estimation step, when the DQI-INS data is available but the GPS data is not available. The second stage is the update step, when the GPS data becomes available it is used to update the position estimate since it is a more accurate sensor.

Estimation Step In this step the position is estimated from the data output by the DQI-INS and several of the matrices for the update step are computed. The task starts by reading in the spacial velocities from the DQI-INS, $\mathbf{v_s} = [u_s\ v_s\ w_s,]$. Then the new position estimate is computed:

$$\mathbf{x_{new}} = \mathbf{x_{old}} + \mathbf{v_s} \cdot \mathtt{time} - \mathtt{step}$$

Also $\mathbf{P_{pre}}$ is updated:

$$\mathbf{P_{pre}} = \mathbf{P_{pos}} + \mathbf{Q_d}$$

Update Step In this step the main Kalman equations are calculated and the updated position is calculated using both the predicted position estimates and the position data output from the GPS.

First, we calculate $\mathbf{K}$:

$$\mathbf{K} = \mathbf{P_{pref}} \cdot (\mathbf{P_{pre}} + \mathbf{R_d})^{-1}$$

Next we calculate $\mathbf{P_{pos}}$:

$$\mathbf{P_{pos}} = \mathbf{P_{pre}} \cdot (\mathbf{I} - \mathbf{K})$$

Finally we calculate the updated position:

$$\mathbf{x_{new}} = \mathbf{x_{old}} + \mathbf{K} \cdot (\mathbf{x_{GPS}} - \mathbf{x_{old}})$$

These two steps are computed inside the fusion Giotto task.

7.3.4 Final Simulation System and Future Work

The final real-time control and simulation system is described in this chapter. It strives to achieves the goals for a second generation helicopter UAV software system stated in Section 3.4. The use of Giotto for the controller establishes the time-based structure we chose to utilize. The use of the UAV platform outlined in Section 6.2.2 and described in detail in Section 7.2.1 helps to achieve the second goal of modularity. Meanwhile the controller software runs on a real-time operating system and on flight ready hardware. Therefore, as per the reason for using a hardware-in-the-loop simulator, the new embedded software system is only a few steps away from flight.

In addition to readying the controller software for flight, the hardware-in-the-loop simulator can now be used to evaluate and explore embedded software implementations. For example, how much does jitter in a standard controller really degrade control performance? These types of questions cannot be answered with uncertain and dangerous flight tests. Another example would be to compare the alternative UAV platform designs presented in this dissertation with respect to their control performances. Multi-modal and multi-vehicle flight can also begin to be explored using this framework.

Adding Complex Control Laws

In the future, the low level control laws that were used in this work may be used as a basis for more complex, higher level controllers. Of greatest interest would be the addition of a regression learning controller. The draw of this controller is that it allows for aggressive maneuvers without requiring an all encompassing, highly accurate nonlinear model. Instead this controller learns the control parameters by using flight test data and a method that relies on the Bellman equation and Bayesian regression. Adding this controller into the Giotto control framework should be a simple operation since it was developed to work with the low level control law currently in use.

Chapter 8

Experimental Results

Numerous test flights have been performed by 'flying' our Giotto and UAV platform embedded control system on the hardware-in-the-loop simulator.

8.1 Velocity Controller Performance

In one series of test flights we configured the control task, which runs from within the Giotto program (Section 7.2.1), to instruct the triangular flight pattern shown in Figure 8.1. The control laws used for this test run require the reference, or desired states, be the velocities in body coordinates (using the North, East, Down configuration) and the heading. This control law is identical to the one shown in equation(2.11) except that it commands velocity and leaves out the position portions:

$$\mathbf{y}_{ref}(t) = (u_{ref}, v_{ref}, w_{ref}, \Psi_{ref}) \tag{8.1}$$

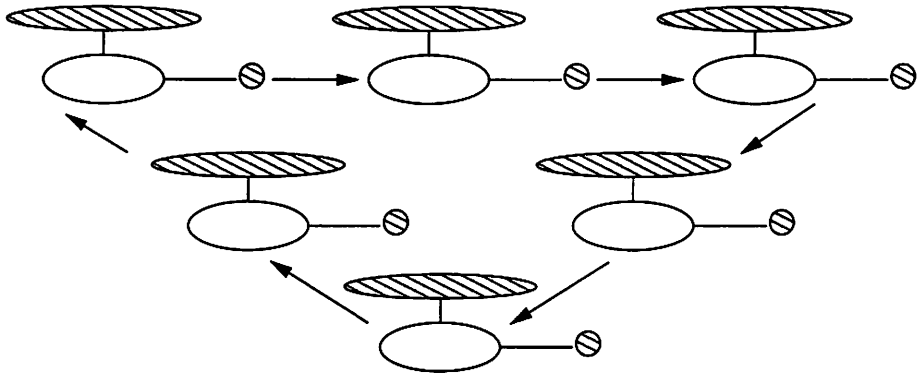


Figure 8.1: Velocity controller flight pattern

$$
\begin{aligned}
u_{a_1} &= -K_\Phi \Phi - K_v \Delta_v \\
u_{b_1} &= -K_\Theta \Theta - K_u \Delta_u \\
u_{\Theta_M} &= -K_w \Delta_w \\
u_{r_{ref}} &= -K_\Psi \Delta_\Psi
\end{aligned}
$$

The triangular pattern consisted of a repetition of the following modes of flight:

Takeoff : Flight in the forward and upwards directions at a 45 degree angle. (recall that upwards velocity is negative in this coordinate system).

Flight : Flight in the backwards direction with no heading change.

Landing : Flight in the forwards and downwards directions at a 45 degree angle.

The desired velocities and heading references are displayed in Figure 8.2. The next figure(8.3), displays the actual velocities and heading output by the hardware-in-the-loop simulator during a real-time test flight. By comparing the figures, one can conclude that the simulated helicopter follows very closely the desired

references. This result attests to the accuracy of our embedded control system in implementing the control laws and to the usefulness of the hardware-in-the-loop simulator.

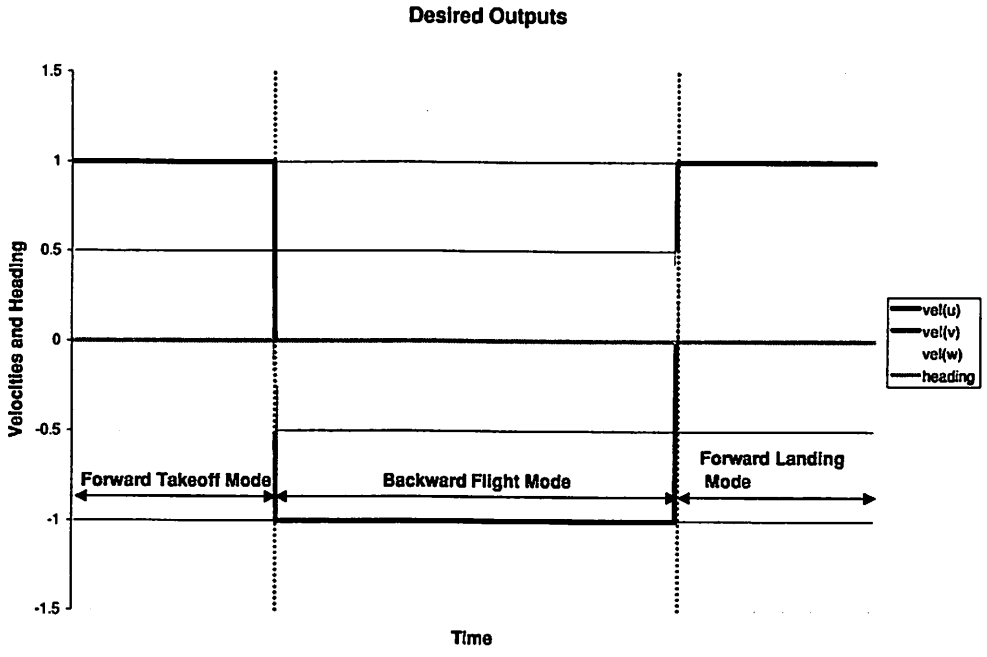


Figure 8.2: Desired velocities and heading

The next four figures take a closer look at the triangular test flight results. This flight pattern calls for the helicopter to always have a desired sideways velocity and heading of zero. On the other hand, the desired velocity downwards is first negative, then zero, then positive as the helicopter lands. The forward velocity also undergoes change, starting positive, then turning to negative during the backwards flight, and resuming positive during the landing.

Figure 8.5 displays both this changing desired forward velocity, and the

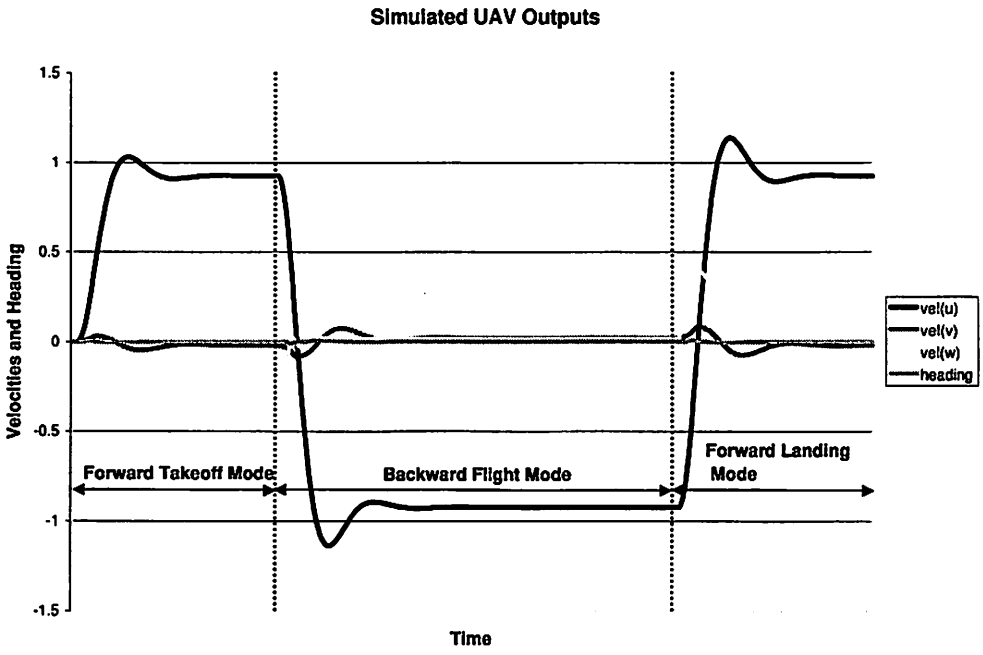


Figure 8.3: Output velocities and heading from HIL simulator

actual forward velocity output by the HIL simulator. Here we can see that the system response is fairly accurate in terms of the final value and the damping. Figure 8.6 displays the changing desired downwards velocity as compared to the actual downwards velocity. This response is less accurate with respect to the final value, but exhibits a shorter rise time.

The two remaining Figures, 8.4and 8.7 display the actual sideways velocity or heading with a desired velocity and heading of zero. Both of these variables exhibit peaking directly after the helicopter switches flight modes. However, the changes in heading are of a much smaller magnitude than the changes in sideways velocity.

These data only begin to illuminate the possible utility of the hardware-in-the-loop simulator. For starters, flight mode switching anomalies can easily be compared between competing controllers. Furthermore, without too much difficulty, this simulation framework could be put to work simulating multiple helicopters flying in formation.

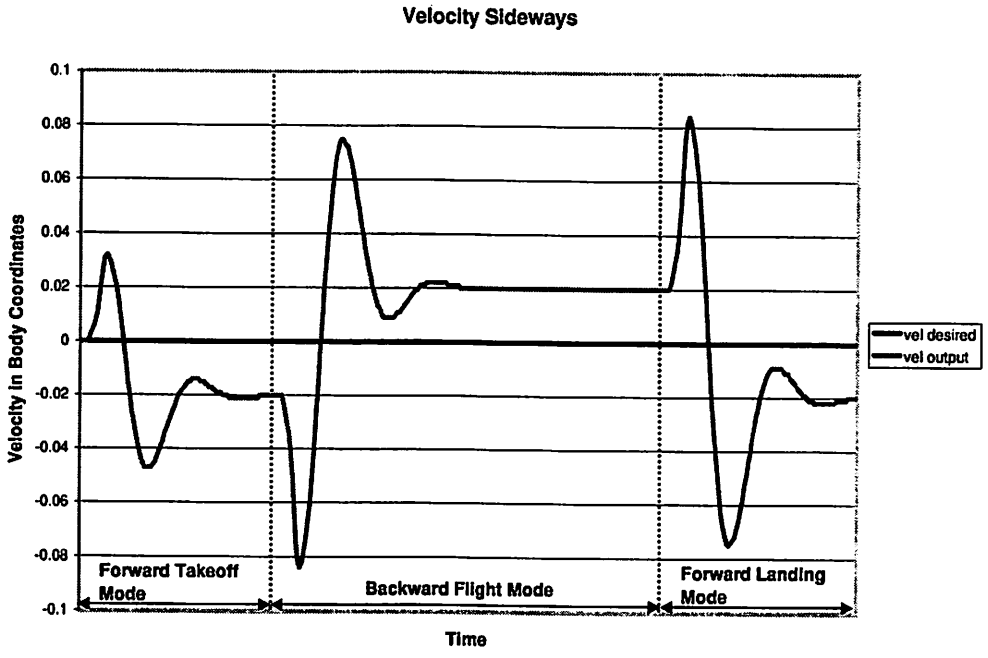


Figure 8.4: Sideways velocity: desired vs. HIL simulator output

8.2 Kalman Filter Performance

Figures 8.8, 8.9, 8.10, display in detail the performance of our Kalman Filtering of the position data. As described in Section 7.3.3 the Kalman filter is used to

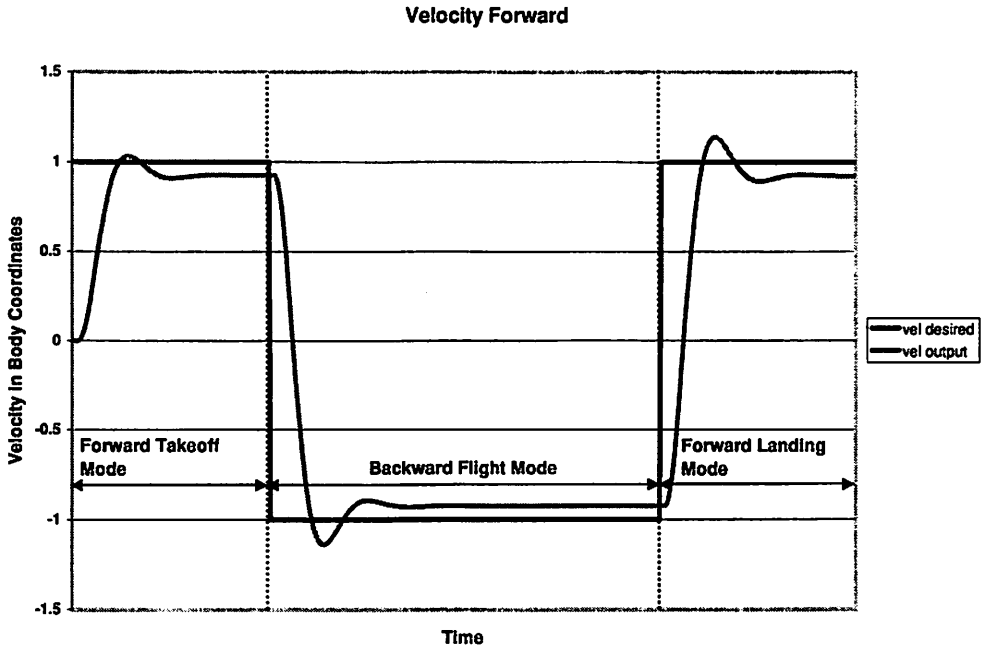


Figure 8.5: Forward velocity: desired vs. HIL simulator output

predict the position data from the INS every time step and update the position data with the GPS measurement when available. Each of these figures clearly displays the drifting of the INS position estimation and the GPS correction points at which the estimated position returns to the accurate plant position.

8.3 Position Controller Performance

We also tested the full position controller shown in equation(2.11). This test run therefore uses the Kalman filter functions as part of the Giotto style control

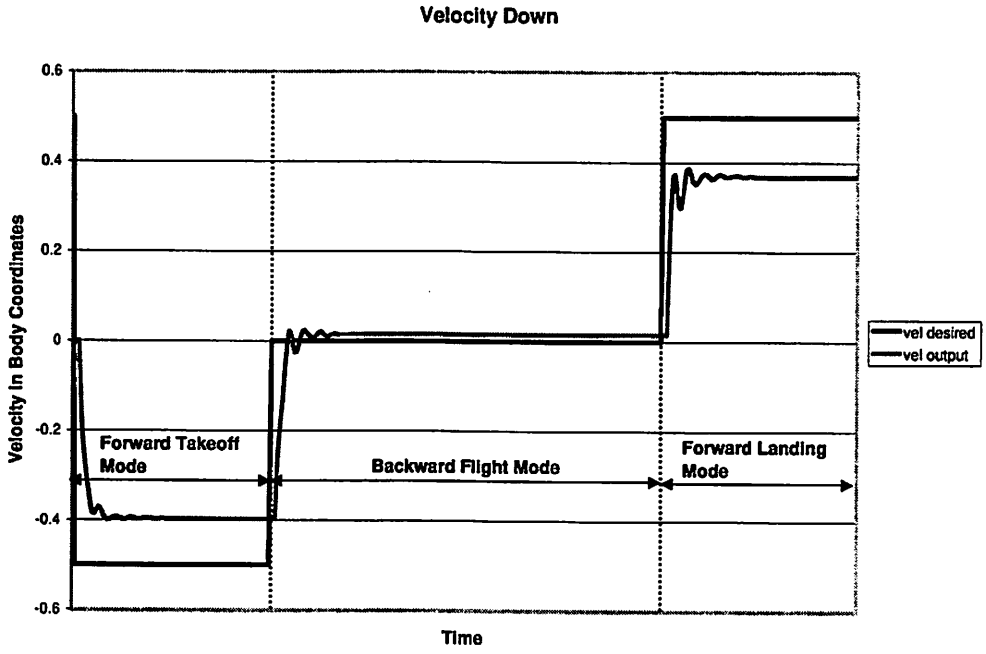


Figure 8.6: Downwards velocity: desired vs. HIL simulator output

program. In this testing scenario the desired trajectory was a square pattern as shown in Figure 8.11:

1. Take off: Upwards movement, negative change in z.
2. Forward flight: positive change in x.
3. 90 degree Turn: positive change in heading.
4. Forward flight: positive change in y.

This pattern, minus the take-off, was repeated to form a square flight formation. The test results shown in Figures 8.12 8.13 display close position following and

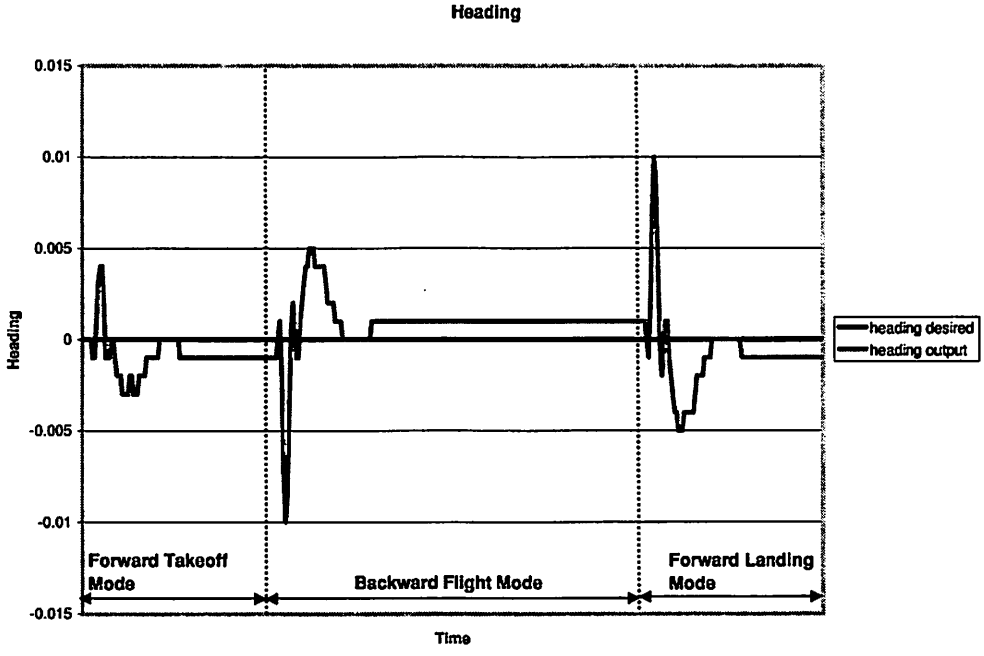


Figure 8.7: Heading: desired vs. HIL simulator output

stability.

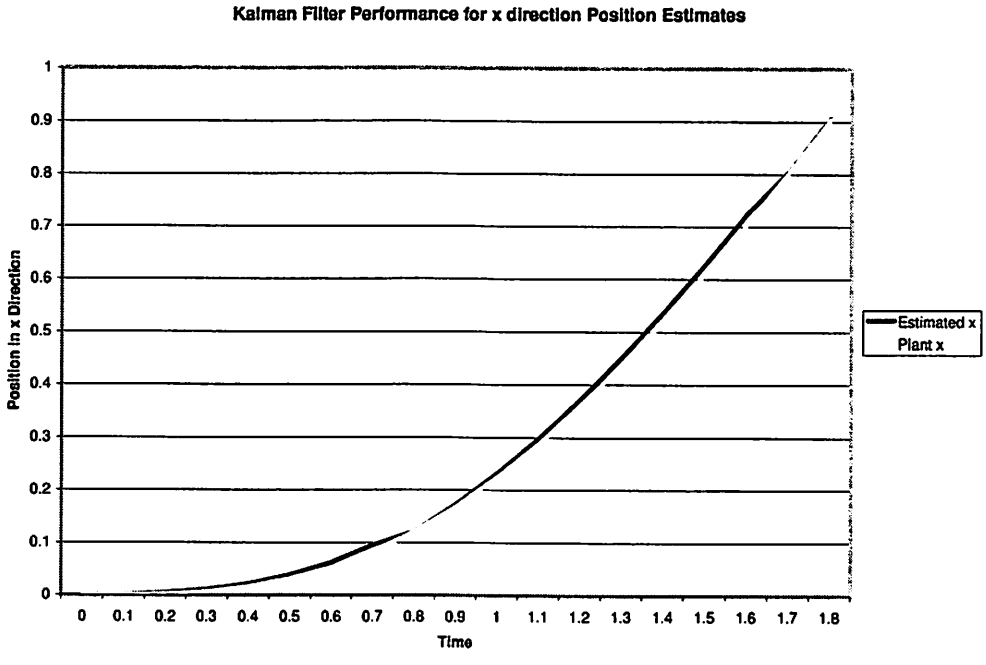


Figure 8.8: True x position vs. estimated x position

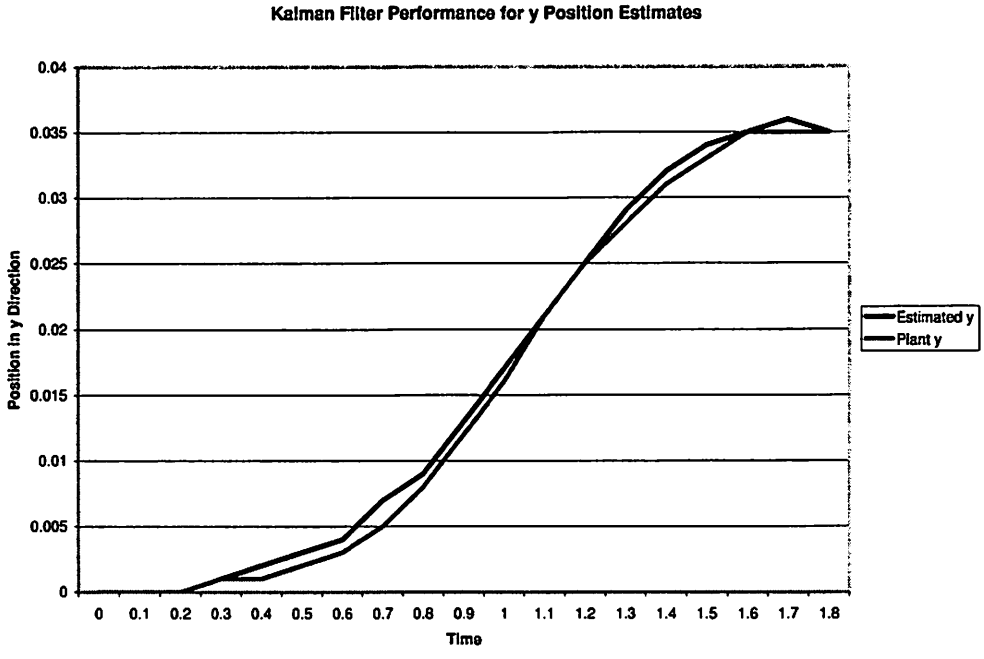


Figure 8.9: True y position vs. estimated y position

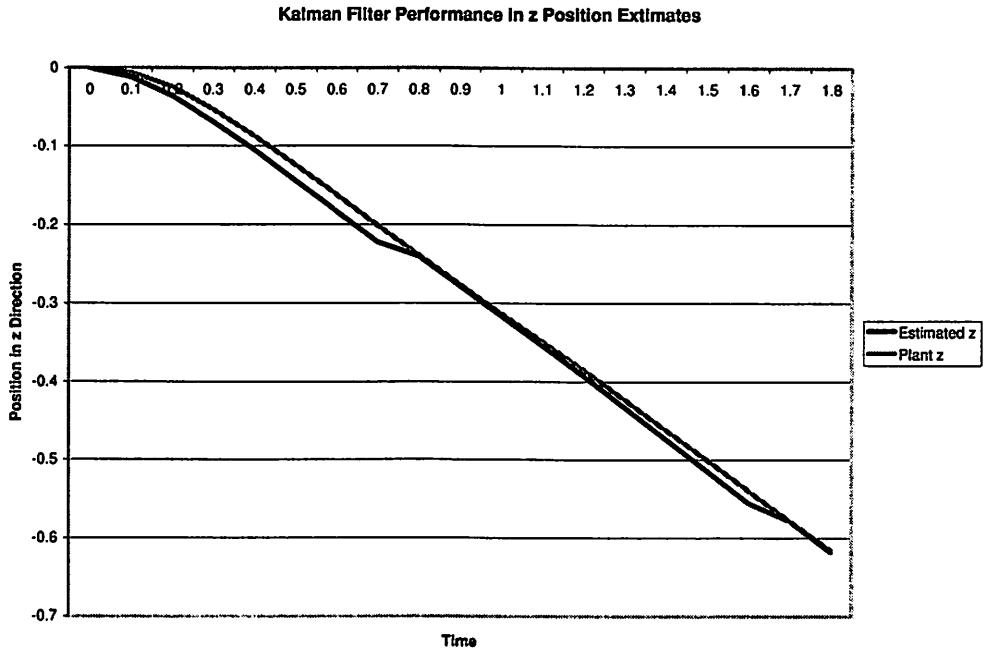


Figure 8.10: True z position vs. estimated z position

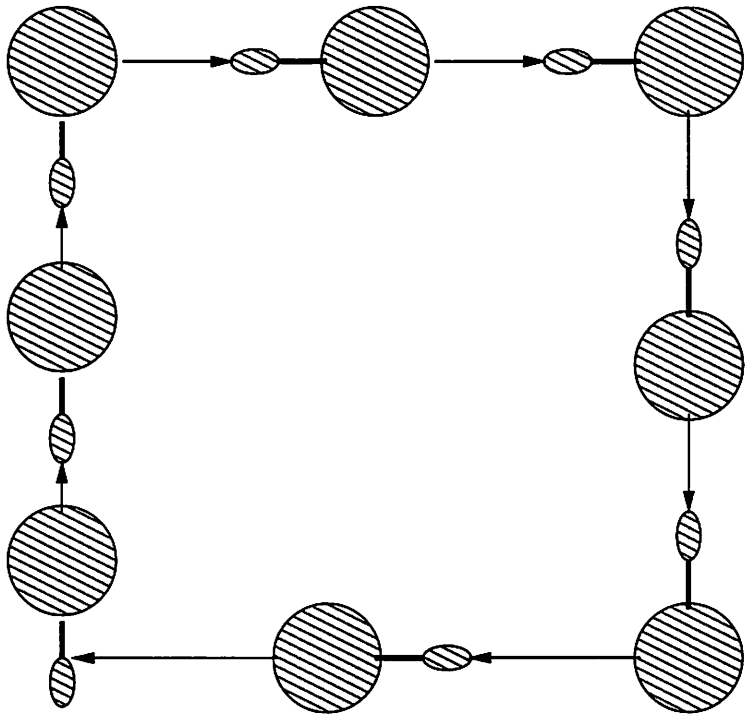


Figure 8.11: Position controller flight pattern

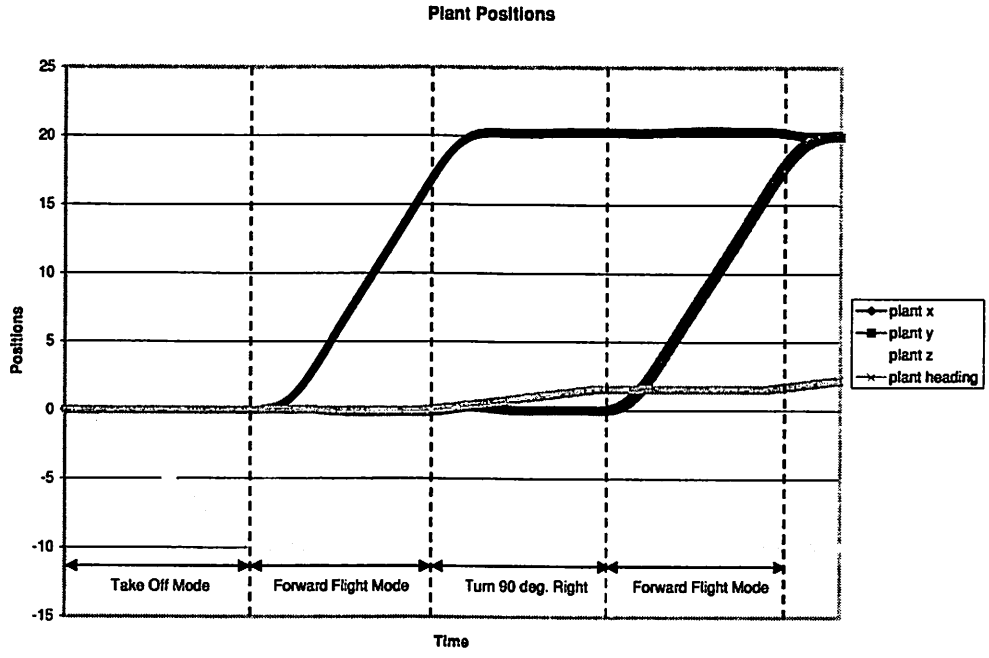


Figure 8.12: Positions and heading of the HILS helicopter

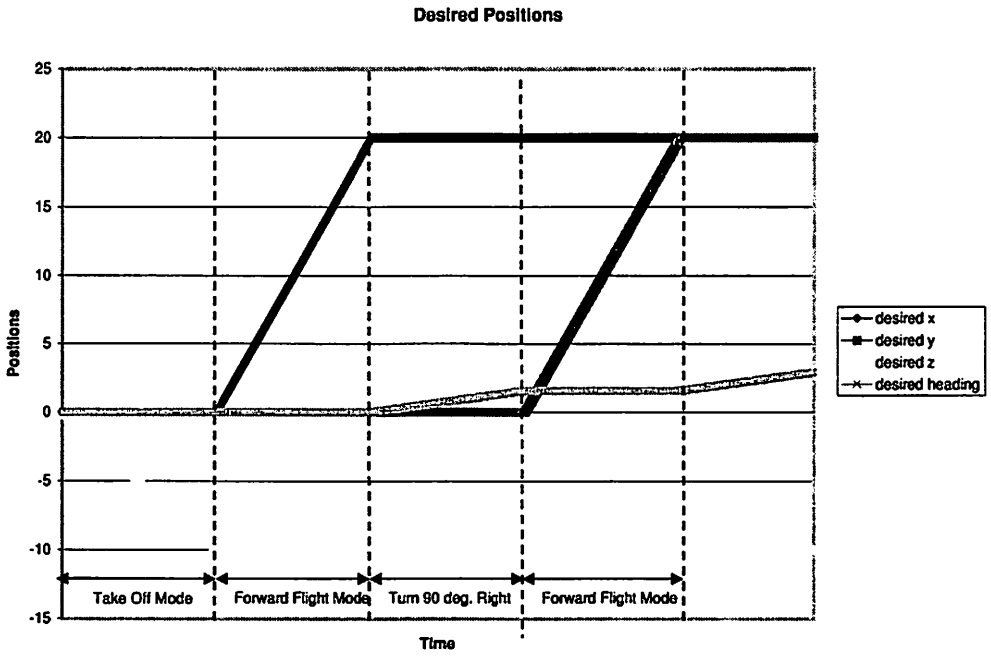


Figure 8.13: Desired positions and heading

Chapter 9

Conclusion

In this dissertation, we presented a methodology for the design of embedded controllers, and a hardware-in-the-loop simulation framework to test the finished product. These two proposals, the controller design and the simulator, both work to explore the integration of control laws and their software implementations.

As discussed, the controller design's use of time-based control provides a relatively simple and verifiable connection between the control algorithms and their realizations. The high level programming language, Giotto, then guarantees the time-based subsystem interactions. We also used the tools of platform-based design to allow the Giotto controller application to function isolated from the lower level system details and yet receive enough information about the important parameters of the lower layers of abstraction to prevent costly redesigns. In addition to providing the appropriate layers of abstraction, this methodology allows integration of legacy code and "foreign" subsystems.

The hardware-in-the-loop simulator helps to explore the connection between the control laws and their implementations by providing an environment to test the final embedded controller implementations. Instead of simply testing

the control algorithms, a hardware-in-the-loop simulator tests the embedded control system by running it in conjunction with a simulated environment. It can provide safe, reliable, and most importantly, repeatable and controlled testing for embedded control systems that cannot be achieved by using the 'real' environment.

To present how our design methodology can be applied, we have discussed two re-designs of the control system of a helicopter based UAV. We have built one of these designs and described that build in detail along with the detailed description of our hardware-in-the-loop simulator for the helicopter UAV. We also presented the experimental results from 'flying' the embedded controller on the simulator. This design goes a long way towards meeting the goals for our second generation helicopter control system:

1. The use of platform-based design allows us to build a bridge between the time-based controller application and the non-time-based sensors and actuators.

2. A time-based controller eliminates the timing irregularities present in first generation system. Further, the Giotto compiler ensures that the controller application meets its timing requirements.

3. Our platform-based design achieves a high degree of modularity. For example, to substitute a different sensor suite in our first redesign requires only changes to the data processor and the data formatting library. The data processor would require a different sensor initialization routine and a new circular buffer; the formatting library would need a new format conversion routine. However, no part of the controller application would need to be changed.

Though our case study contains many details that are specific to our helicopter system, our methodology is widely applicable. We believe that the

combination of time-based control and platform-based design can be generally applied to automation control systems, for which legacy software, independently engineered subsystems, and strict reliability and timing requirements all play a crucial role.

Bibliography

[ÅBE+99] K.-E. Årzén, B. Bernhardsson, J. Eker, A. Cervin, P. Persson, K. Nilsson, and L. Sha. Integrated control and scheduling. Internal report TFRT-7582, Department of Automatic Control, Lund Institute of Technology, August 1999.

[Bła76] J. Błażewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In E. Gelembe and H. Beilner, editors, *Proceedings of the International Workshop on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, October 1976.

[BLMSV98] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-Vincentelli. Scheduling for embedded real-time systems. *IEEE Design and Test of Computers*, January-March 1998.

[CSB90] H. Chetto, M. Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, September 1990.

[HHK01a] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Embedded control systems development with Giotto. In *Proc. of the Intl. Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '01)*, pages 64–72, August 2001.

[HHK01b] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto: a time-triggered language for embedded programming. In *Proc. of the*

1st Intl. Workshop on Embedded Software (EMSOFT '01), LNCS 2211, pages 166–184. Springer-Verlag, October 2001.

[KH01] C.M. Kirsch and T.A. Henzinger. The embedded machine. Technical report, University of California, Berkeley, 2001.

[KLMS01] T.J. Koo, J. Liebman, C. Ma, and S. Sastry. Hierarchical approach for design of multi-vehicle multi-modal embedded software. In *Proceedings of the First International Workshop on Embedded Software*, October 2001.

[Koo00] T. J. Koo. *Hybrid System Design and Embedded Controller Synthesis for Multi-Modal Control.* PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 2000.

[Kop97] H. Kopetz. *Real-time systems: design principles for distributed embedded applications.* Kluwer, 1997.

[Kop98] H. Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *The Fourth International Symposium on Autonomous Decentralized Systems*, 1998.

[KPS01] T.J. Koo, G. Pappas, and S. Sastry. Mode switching synthesis for reachability specifications. In *Lecture Notes In Computer Science*, Hybrid Systems: Computation and Control. Springer Verlag, 2001.

[KS98] T.J. Koo and S. Sastry. Output tracking control design of a helicopter model based on approximate linearization. In *Proc. 37th Conference on Decision and Control*, pages 3635–3640, December 1998.

[KSHP02] C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree. A Giotto-based helicopter control system (draft), 2002.

[KSS+98] T.J. Koo, D.H. Shim, O. Shakernia, B. Sinopoli, F. Hoffmann, and S. Sastry. Hierarchical hybrid system design on berkeley uav. Submitted to the International Aerial Robotics Competition, University of California at Berkeley, August 1998.

[Led99] J.A. Ledin. Hardware-in-the-loop simulation. *Embedded Systems Programming*, 12(2):42–60, February 1999.

[LH95] J.W.S. Liu and R. Ha. Methods for validating real-time constraints. *Journal of Systems and Software*, 30(1–2):85–98, July–August 1995.

[LMW95] Y.T.S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th RTSS*, pages 97–108. IEEE Press, 1995.

[LTS99] J. Lygeros, C. Tomlin, and S. Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, 35(3), March 1999.

[MLS94] R. M. Murray, Z. Li, and S. Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation.* CRC Press, 1994.

[MTK99] B. Mettler, M.B. Tischler, and T. Kanade. System identification of small-size unmanned helicopter dynmaics. *American Helicopter Society* 55^{th} *Forum*, May 1999.

[num92] Numerical recipes in C: The art of scientific computing. Cambridge University Press, 1992.

[San02] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[Shi00] D.H. Shim. *Hierarchical Flight Control System Synthesis for Rotorcraft-based UAVs.* PhD thesis, UC Berkeley, December 2000.

[SKHS98] D.H. Shim, T.J. Koo, F. Hoffmann, and S. Sastry. A comprehensive study of control design for an autonomous helicopter. In

Proc. 37th Conference on Decision and Control, pages 3653–3658, December 1998.

[SS01] M.A.A. Sanvido and W. Schaufelberger. Design of a framework for hardware-in-the-loop simulation and its application to a model helicopter. In *Proc. of the 4th Intl. Eurosim Congress*, June 2001.

[Win99] Tornado getting started guide, 2.0. WindRiver Systems, April 1999.