6

Real-Time Embedded Sensor Processing for an Autonomous Helicopter

by

Chin San Han

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering
in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

May 26, 1998

Author [illegible]
Department of Electrical Engineering and Computer Science
May 26, 1998

Approved by [illegible]
Paul A. DeBitetto
Senior Technical Staff, Charles Stark Draper Laboratory
Thesis Supervisor

Certified by [illegible]
Stephen K. Burns
Senior Research Scientist, Massachusetts Institute of Technology
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

Real-Time Embedded Sensor Processing for an Autonomous Helicopter

by

Chin San Han

Submitted to the
Department of Electrical Engineering and Computer Science

May 26, 1998

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering
in Electrical Engineering and Computer Science

Abstract

In 1997, the Charles Stark Draper Laboratory in Cambridge, Massachusetts continued its research in autonomous agents by designing and constructing the second-generation Draper Small Autonomous Aerial Vehicle (DSAAV). This aircraft, based on a commercially produced model helicopter, has been outfitted with numerous navigational systems to achieve autonomous flight between waypoints established by a ground control station. One such system, known as the Sensor Processing Unit (SPU), was developed as part of the 1997 effort to redesign the DSAAV. This device consolidates and integrates three electronic sensors, an accelerometer, compass, and sonar altimeter, along with other related components into a single device that administers both sensor control and data processing, thus relieving the on-board navigational computer of these tasks. Unfortunately, the 1997 SPU revealed an inability to achieve its objectives satisfactorily due to flaws in both hardware and software that resulted in part from the lack of an appropriate design methodology. Therefore, this thesis establishes a framework for understanding and realizing sensor integration in autonomous vehicles, and formulates a procedural basis for both evaluating the 1997 SPU and reengineering its architecture from foundation through implementation to achieve real-time embedded sensor processing for an autonomous helicopter.

Acknowledgment

This thesis was prepared at the Charles Stark Draper Laboratory, Inc.

Publication of this thesis does not constitute approval by The Charles Stark Draper Laboratory, Inc. of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

[illegible]

Forward

This thesis, undertaken at the Charles Stark Draper Laboratory, began in the fall of 1997 during my sixth academic year at the Massachusetts Institute of Technology. I have many to thank for not only their assistance during this project but also my development as an engineer, researcher, and student over the course of my undergraduate and graduate experience. My gratitude and sincerest thanks:

To Paul Debitetto, for your guidance and constructive criticism which kept this thesis on track, and for your encouraging words which lifted my spirits;

To Stephen Burns, for your patient reading, analysis, and supervision of my last thesis drafts;

To Robert Faiz, for your experience and sense of humor which always knew how to handle things when they broke, and when they did not;

To my lab companions, for your helpful, friendly, and knowledgeable presence in the lab;

To my teachers, academic and artistic, intellectual and spiritual, for your advice and direction that helped me to learn what I love;

To my friends, for your constant encouragement and care throughout and especially during a term full of uncertainty and questions about the future;

To my family, for your ever-present, ever-surprising love for a young man who is trying to find his way in life;

To my Abba, for teaching me to choose Life at every moment, for always leading me in the Way, for walking right beside me no matter what the cost.

Contents

Chapter 1 - Autonomy and Sense: Machines that Feel 15

1.1 Introduction. 15

1.2 Autonomous Devices. 15

1.1.1 Autonomous Agents 15

1.1.2 Autonomous Vehicles 16

1.3 Sensors. 17

1.4 Sensor Control and Integration in Autonomous Vehicles 18

1.5 Summary 18

Chapter 2 - Distributed Processing: Teamwork 19

2.1 Introduction. 19

2.2 Overview of Distributed Processing 19

2.3 Design of Distributed Systems 20

2.3.1 Modularity 20

2.3.2 Autonomy 21

2.3.3 Cooperativeness 21

2.4 Role of Distributed Processing in Autonomous Vehicles 22

2.5 Summary 23

Chapter 3 - 1997 DSSAV and SPU: Integration by Parts 24

3.1 Introduction 24

3.2 Overview of the 1997 DSAAV Distributed Systems 24

3.2.1 On-Board Navigation 25

3.2.2 Ground Guidance 25

3.2.3 Vision Processing 26

3.2.4 Sensor Processing 26

3.3 DSSAV Sensor Integration and Control 27

3.3.1 Overview of Sensor Processing Unit 27
3.3.2 SPU Objectives 28
3.4 1997 SPU Hardware Design and Implementation 29
3.4.1 Component Selection 29
3.4.1a Microprocessor Selection 29
3.4.1b A/D Converter Selection 30
3.4.1c Anti-Aliasing Filter Selection 30
3.4.1d Other Considerations 31
3.4.2 System Design 32
3.4.3 Circuit Design 32
3.4.3a Analog Design 32
3.4.3b Digital Design 32
3.5 1997 SPU Software Design and Implementation 36
3.5.1 Compass-Sonar Subsystem Software 36
3.5.2 IMU Subsystem Software 38
3.6 Summary 41

Chapter 4 - Evaluating the 1997 SPU: Cracks in the Wall 42
4.1 Introduction 42
4.2 Evaluating 1997 SPU 42
4.2.1 Consolidation and Integration of Sensors 42
4.2.2 Distribution of Sensor Processing Load 43
4.2.3 Augmentation of System Accuracy and Bandwidth 44
4.2.3a Modeling Inertial Errors 45
4.2.3b Applying the Error Model to the 1997 SPU 48
4.3 Summary 50

Chapter 5 - The 1998 Sensor Processing Unit: Building Blueprints 51
5.1 Introduction 51
5.2 Objectives 52
5.3 System Proposal for the 1998 SPU 53
5.3.1 1997 SPU System 53
5.3.2 1998 SPU System 54
5.4 Implementation Considerations 55
5.4.1 Hardware Considerations 55
5.4.1a Microprocessor vs. Microcontroller 55
5.4.1b A/D Converter 56
5.4.1c Anti-Aliasing Filters 57
5.4.2 Software Considerations 57

5.4.2a Software Architecture 57
5.4.2b Software Characteristics 58
5.4.3 Hardware Component and Software Implementation Guidelines . . 59
5.5 Hardware Design 59
5.5.1 Microcontroller Options 60
5.5.1a PC-104 Compliant Microcontroller 60
5.5.1b Enhanced Microcontroller 61
5.5.1c PIC17C756 Microcontroller 61
5.5.2 The PIC17C756 and the SPU 62
5.5.2a Overview of PIC17C756 Capabilities 62
5.5.2b Comparison of PIC17C756 to 1997 SPU 62
5.5.3 Design and Implementation 64
5.6 Software Design. 67
5.6.1 Sensor Protocols 67
5.6.1a Flux Magnetometer (Electronic Compass) 67
5.6.1b Sonar Altimeter 68
5.6.1c Inertial Measuring Unit 69
5.6.1d Optimal Sensor Sampling Rates 69
5.6.2 Software Event Categorizations 70
5.6.2a Period-Based Events 70
5.6.2b Occurrence-Based Events 71
5.6.3 Software Implementation Concerns 71
5.6.3a Communication with Navigational Computer 72
5.6.3b Interrupt Handlers versus Linear Code 72
5.6.4 Software Control Schemes and Schematics 73
5.6.4a Overall Implementation 73
5.6.4b Technical Details 74
5.7 Reviewing the 1998 SPU. 76
5.7.1 Comparing the 1998 SPU to the 1997 SPU 76
5.7.1a Hardware Comparison 76
5.7.1b Software Comparison 77
5.7.2 1998 SPU Advantages Over 1997 SPU 77
5.7.3 1998 SPU Disadvantages Over 1997 SPU 77
5.8 Summary 78

Chapter 6 - 1998 SPU Design Realization: Pipe Dreams 79

6.1 Introduction 79
6.2 SPU Hardware Prototyping Considerations 79
6.3 Software Simulation and Analysis 80

6.4 Summary 80

Chapter 7 - Conclusion: Matters to Mind 81

Appendix A: 1998 SPU PIC17C756 Software: 17c756.h 84

Appendix B: 1998 SPU PIC17C756 Software: spu-p17.asm 93

References 123

List of Figures and Tables

Figures

3.4.2-1: 1997 SPU System Diagram 33
3.4.3a-1: 1997 SPU Analog Circuit Schematic 34
3.4.3b-1: 1997 SPU Digital Circuit Schematic 35
3.5.1-1: 1997 SPU Compass-Sonar Service Control Diagram 37
3.5.1-2: 1997 SPU Compass-Sonar Interrupt Control Diagram. 39
3.5.2-1: 1997 SPU IMU Service and Interrupt Control Diagram. 40
4.2.2-1: 1997 SPU Transmitted IMU Data. 45
4.3.2b-1: 1997 SPU Transmitted IMU Data w/ Errors 49
5.3.1-1: 1997 SPU System Diagram (same as Figure 3.4.2-1) 53
5.3.2-1: Generalized SPU System Diagram 55
5.5.3-1: 1998 SPU System Architecture 65
5.5.3-2 1998 SPU Proposed Circuit Schematic 66
5.6.1a-1: Compass Signal Timing Schedule 67
5.6.1b-1: Sonar Signal Timing Schedule 68
5.6.4-1: 1998 SPU Software Control Diagram. 75

Tables

5,5,2a-1: PIC17C756 Microcontroller and Peripheral Features 63
5.6.1d-1: 1998 SPU Optimal Sensor and Status Sampling Rates 70

Chapter 1

Autonomy and Sense: Machines that Feel

1.1 Introduction

Autonomous devices play an integral role in modern times. Often responsible for tasks once relegated to human operators, autonomous machines have demonstrated an aptness for replacing a human's actions. But, by implication, an "autonomous" agent does more than mimic movement, but is also capable of some form of intelligence. Granted that the level of intelligence is relative to the application, but the assumption is that a machine fundamentally replaces not only human manipulation but also motive. This idea has direct implication for not only the way a machine "thinks", but also the way it interacts with its surrounding environment. Since complete autonomy requires an ability to perceive the world in which it operates, autonomous devices should be capable of some form of sensory perception. This chapter outlines autonomous devices from a very generalized perspective and narrows to a specific focus on autonomous vehicles, and discusses how sensors fit into their makeup.

1.2 Autonomous Devices

1.1.1 Autonomous Agents

An "autonomous agent" is any entity capable of existing independently, carrying on without outside control, having self-government. In essence, an autonomous agent acts without human direction and behaves only in response to its purpose. This definition immediately implies two component features of autonomous devices. First, autonomous agents must have some form of intelligence in order to carry out its mission. While "intelligence" is a relative term, in the context of autonomous agents it refers to a capacity to apprehend the information necessary to execute its mission and respond accordingly. The second feature of autonomous agents is an ability to evaluate itself in light of its purpose not simply in an intellectual sense, but also in a physical one as well. Simply put, the device must be able to observe its state and its environment space. Of course, a rather large arena of possible autonomous agent candidates present themselves given the two aforementioned qualifications but the issue at heart lies in machine autonomy that replaces human intelligence and intervention. And while this can be interpreted in many ways and applied to a number of possible autonomous agents, the field can be narrowed further given the understanding that replacement of human control and action, and not necessarily human form, is of concern, as in autonomous vehicles for example.

1.1.2 Autonomous Vehicles

Autonomous vehicles are a specialized breed of autonomous agents that seek to mimic the intelligent motion control of living creatures. While their purposes might vary depending on the vehicle, all distinguish themselves from the general class of autonomous agents by two additional characteristics, namely independent mobility and interactiveness. These two qualities attributed to intelligent vehicle motion are ones which strike at the core dilemma in the development of autonomous vehicles: Replacing human mind and sight with a machine. Truly, the autonomous mobility and interaction of a vehicle encapsulate parts of the basic notion of a robot: "An automatic apparatus or device that performs functions ordinarily ascribed to human beings or operates with what appears to be almost human intelligence" [1].

The mobility of autonomous vehicles implies that these devices are capable of physical movement. In fact, motion of autonomous vehicles is key in defining their autonomy since intelligent mechanical motion control is normally ascribed solely to living creatures. The modern view of a mechanical vehicle is one in which a human intercedes in some capacity to control or assist the device's actions. For instance, automobiles, aircraft, and bicycles typically have a person somewhere in the control scheme and in the case of bicycles, humans also provide locomotive power. In these scenarios, the vehicle often acts as an amplification or extension of the human pilot, providing a means of travel which is directed by the human mind. Consider the automobile: A person controls the movement of the vehicle, just as he controls his own two-footed steps; a person supplies the automobile's fuel, just as he directs his own food intake; a person monitors the car's gauges to verify proper functioning of all systems, just as he monitors his own senses to determine good health. But, the automobile alone is inanimate and ill-equipped to direct its movement intelligently within a given environment. Therefore, the complete automation of vehicle motion control ultimately attempts to replace human control and as such to achieve true autonomy, and quite possibly intelligence.

Interactiveness, the second characteristic of autonomous vehicles, is crucial to the evolution of autonomy and comes almost as a prerequisite for independent mobility. In order for an autonomous vehicle to move with purpose, it must be able to interpret and respond to its environment. A truly autonomous vehicle must be equipped to observe its surroundings in as many dimensions as necessary to not only ensure survival but also carry out its intended mission. Otherwise, intelligent motion control is impossible. This sensory perception lies at the heart of autonomy since intelligent observation of environment by autonomous vehicles seeks to imitate that typically demonstrated only by living beings. For instance, while automobiles, aircraft, and even some bicycles may be equipped with instrumentation that monitor both internal and external environmental status, a person must ultimately process and interpret such data. Consequently, these machines can hardly be labelled as autonomous. Further, a human often acts as the sole source of sensory reception and processing. For example, an automobile has no way of detecting roads. The driver is totally responsible for recognition of the path and directing an appropriate course of action. This indirectly articulates a major barrier to the development of autonomous vehicles: Devising devices and methods for sensing environmental conditions. Without this machinery, more commonly known as sensors, environmental interaction, both in observation and response, could not be possible and autonomous motion control is left out of reach. Consequently, sensors in autonomous vehicles are of paramount importance for while they cannot replace human sensory processing, they can provide sensory perception. Sensors place the lofty goal of autonomous motion control within technology's grasp and as such are the focus of much research and development.

1.3 Sensors

Sensors, or devices that respond to physical stimulus such as heat, light, and motion by transmitting a resulting impulse [1], typically fill the role of being an electronic or mechanical counterpart to biological sensory nerves. Just as these nerves are a fundamental building block of biological sensory perception, sensors play the equivalent role for electromechanical sensing. They act as an integral part of systems that must interact with the physical world, providing the electronic or mechanical "nerve" response which allow devices to observe and react to their environments. Without sensors, man-made systems cannot hope to respond to their surroundings without human intervention.

In this electronic age, sensors have seen great technological innovation and innumerable creative applications. In certain applications, sensors diminish or even remove the need for human intervention in systems requiring real-world feedback and control loops. Indeed, sensors often exceed the capabilities of human perception! For example, in quality control of ultra-high precision machining, a position probe provides feedback to a controller indicating cutting errors in machined parts that can be compensated for in the next cutting cycle. This sensor removes the need for a human to measure the cut part and provide the necessary error compensation. Further, the probe resolution of micrometers greatly exceeds that detectable by the human eye and as such can provide more detailed and accurate part information. In this situation and many others, even though a human operator commands the device, a sensor replaces a human in providing sensory data for control. Similarly, in the context of autonomous vehicles, sensors afford these machines with information about their internal and external environments. Everything from compass-heading to temperature can be observed and relayed to a sensory processing unit. But, while in the case of precision machining a human acted as the "processing unit," who or what plays the equivalent part in machines and, more specifically, in autonomous vehicles? Furthermore, how does it function? These questions lie at the heart of sensor control technique, an integral design consideration in the development of autonomous vehicles.

1.4 Sensor Control and Integration in Autonomous Vehicles

Typically limited to the observation of one type of physical phenomena i.e., temperature, velocity, electromagnetic field strength, sensors are usually designed with a high degree of autonomy in and of themselves. Since a single sensor typically focuses on the monitoring of one environmental aspect, its control lends itself to being completely contained as an inherent part of the sensor itself. Due to this high degree of autonomy, sensors can often be transported between applications requiring their functionality virtually at will, the major limiting factor being differences in interface protocol. Nevertheless, while sensors operate with much independence, they are typically used in conjunction with other sensors to increase the ways in which the surrounding environment can be observed. Using multiple sensors of similar and different types creates pathways for more diverse modes of observation and can consequently improve data variety, redundancy, and resolution. But, due to the autonomous nature of individual sensors, a synthesis of sensory information cannot simply occur as a matter of fact. A cooperative network of various sensors requires an appropriate control and integration methodology that will harness both the functionality and diversity of these devices. In applications involving high degrees of autonomy, distributed processing stands out as such a design methodology.

1.5 Summary

Indeed, autonomous agents and autonomous vehicles aspire to be machines that feel. These devices are not only intelligent, capable of interpreting and acting upon given circumstances, but are also able to sense those conditions whether internal or external. Yet, development of such devices and their sensory perception are no easy feats and call for a research methodology suited to attacking the inherent problems involved in creating autonomous instruments and, more specifically, autonomous vehicles.

Chapter 2

Distributed Processing: Teamwork

2.1 Introduction

Distributed processing techniques take advantage of autonomy as a tool for organizational structure and development. By dividing larger systems into smaller, autonomous sub-systems, each component can be researched and implemented separately thus reducing design time and overall complexity, and increasing robustness. For example, since autonomous vehicles require a variety of sensors in order to achieve a multi-faceted view of its surrounding environment, these sensors can be organized into a single, centralized unit with the intent of simplifying both the structure and control of the entire vehicle. While this may not be immediately obvious, distributed processing applied to the organization of autonomous vehicles presents unique advantages which clearly demonstrate the benefit of centralized sensor processing, and should be investigated.

2.2 Overview of Distributed Processing

Distributed processing is a model for the handling of a complex system by dividing it into smaller component sub-systems. Typically, each component is specified and designed such that its operation is not dependent on any other sub-system and can therefore stand alone. Of course, the sub-systems's independence in carrying out its assigned duties does not preclude an ability to interact with other components. Indeed, this interaction between sub-systems is the basis for distributed processing: that the resultant machinery would be greater than the mere sum of its parts.

Ideally in distributed processing, each of the component systems work in tandem as parallel processes. This obliterates the traditional linear view of systems which requires each step of a complex operation to flow from one to another. Under a process-oriented rubric, complex systems cannot be divided solely by operation types as in traditional programming styles. Such programming methods identify operations in the overall program and formalize them as subroutines. While this does provide space savings and increases modularity, the code is fundamentally structured upon a linear progression of commands and subroutines. Of course, identification and formalization of subroutines within a distributed processing architecture has value, bringing the same efficiency in space and modularity that traditional linear programming offers. Nevertheless, a process not only involves commands and subroutines in its programming but also must be capable of handling events in real time as the occur. Since events may or may not be predictable or repeatable, processes cannot be a mere collection of linear operations since this presumes a completely predictable universe. In essence, distributed processing is about individual, self-sufficient real-time processes that cooperate in parallel with one another to accomplish a larger, more complex task.

2.3 Design of Distributed Systems

Rather than dealing with all functions simultaneously both in implementation and operation, a distributed processing approach simplifies a complicated system by dividing it into multiple, function-focused sub-systems. While the identification and implementation of such sub-systems is often unobvious and requires careful thought, several sub-system characteristics clearly present themselves and must be considered during design, namely modularity, autonomy, and cooperativeness.

2.3.1 Modularity

Modularity refers the quality of being standardized to allow flexibility and variety in use. As applied to distributed processing, a modular sub-system would be clearly defined in role, having a specified function or operation that is not context dependent. Ideally, this implies that sub-systems are easily transferred between applications which require similar functionality. For example, the navigational guidance system of a autonomous land rover should be easily transferred to an autonomous aerial helicopter. Of course, this notion seems ridiculous due to the extreme difference in both operating environments and navigational considerations of autonomous land and aerial vehicles. But, fundamentally, the example ignores a key assumption in modular theory: the degree to which a sub-system is independent of context is a function of its interpreted application realm. In other words, the identification of subsystems and thus the extent of their modularity is bounded by the determination of its application species. For instance, autonomous land rovers could plausibly be derived from the generalization of "autonomous terrestrial vehicles." Therefore, a navigational guidance system designed with the terrestrial vehicle model in mind could ideally be transferred between a four-wheeled land rover and a quadruped robot. This navigational system could be considered modular within the domain of autonomous terrestrial vehicles, but not necessarily enough to traverse the gap between ground and aerial agents. So, in short, modular design of sub-systems requires standardization of system functionality within the scope of its application species, thus allowing both flexibility and variety in use within that application sphere.

2.3.2 Autonomy

Self-government or self-rule is the crux of autonomy, and autonomy is a key feature of distributed process sub-systems. An autonomous sub-system should be able to operate independently of external systems with all requisite operations for proper functionality lying within its control. This characteristic of sub-systems derives itself in part from the idea of modularity. Ideally, modular systems are extremely flexible and reusable from one application to the next. Therefore, the functionality of the sub-system must be completely transportable between agents. If a sub-system is dependent on other devices for proper function, it could not be exchanged between applications without altering either the system or the application itself. Furthermore, such a transport could prove impossible for sub-systems relying on other systems since there is no guarantee that the necessary components will be available from one application to the next. Consequently, by formalizing system autonomy as a design requirement in distributed processing, a sub-system should be independent in carrying out its functions and can therefore be transparently exchanged in or between applications requiring it. Futhermore, due to its autonomy, the sub-system carries with it all necessary components for its own processing needs. This distributes the overall computational load over the various sub-systems of a larger application and as such becomes the feature from which the term "distributed processing" is largely derived.

2.3.3 Cooperativeness

While autonomy is an established design principle, this cannot be equated with autocracy which refers to unlimited and complete government by a single entity. While an autonomous sub-system is indeed self-ruled and independent, this does not imply a qualification for governing the entire operation of a given application arena. Granted that this may be obvious, but the intent in distinguishing "autonomy" from "autocracy" in sub-systems is to not only acknowledge the difference but also clarify the need for cooperation. Distributed systems function on a basic "division-of-labor" principle: that each sub-system tackles a certain portion of the larger task at hand, and that together they would accomplish a greater feat that no single one could accomplish alone. But, of course, this synthesis of labor does not simply occur. A plan for cooperation must be detailed as an integral part of the infrastructure of the larger application. This might encompass everything from determining a proper communication protocol to choosing an appropriate hardware interface. Regardless, the pathways for coordination of sub-system operation cannot be ignored in design since distributed processing methodology requires that each sub-system be autonomous in operation but cooperative in application.

2.4 Role of Distributed Processing in Autonomous Vehicles

In many electromechanical applications, numerous functions, internal and external interactions, and computations are necessary to achieve intricate, coordinated, and highly complex behavior. Without an appropriate plan of attack, the application's development can require long design times and extremely involved research due to its complicatedness. But, if an application exhibits an aptness for being divided into modular and autonomous sub-systems, a distributed processing methodology can be utilized to simplify the overall design and implementation, as in the case of autonomous vehicles.

Autonomous vehicles, falling under the regime of autonomous agents, have several defining characteristics previously identified as intelligent, self-evaluative, mobile, and interactive. Each of these characteristics imply possible sub-system implementations that realize these properties, and together generate an intelligent, self-controlled vehicle. For example, an intelligence hints at some form of central processing or directing device dedicated to interpreting, evaluating, and commanding the vehicle for its intended purpose. Of course, the level of intelligence is relative to the design specifications, and in the case of autonomous vehicles is limited to an ability to independently direct movement in response to a given mission. This central processing unit is both autonomous as a device in itself and modular since its functionality can be transported from one autonomous vehicle to another. Similarly, since autonomous vehicles must be able to evaluate themselves with respect to and interact with the environment, the means of detecting surrounding conditions could be consolidated into a single sensory perception device. Even the mechanical apparatus can be seen as a system component of the larger application. In each case, sub-systems implicitly carry some part of the larger computational burden required by the entire application thus preventing the load from being placed entirely on the CPU as would occur in a completely centralized structure. Consequently, all of the aforementioned sub-systems, while autonomous and modular, must cooperate in order to generate an intelligent, self-governed vehicle. Therefore, autonomous vehicles are immediately suited to applying distributed processing techniques for the simplification, organization, and distribution of electromechanical and computational structure. Furthermore, these very generally outlined sub-systems themselves may be subdivided into component parts in a recursive application of distributed processing, as in the case of sensor processing. This multilayered view of distributed processing in autonomous vehicles becomes a basis for evaluating the effectiveness of such devices and their design.

2.5 Summary

In short, distributed processing applied to autonomous vehicles is about teamwork. It involves the identification of players and their roles not only as individual components but also as members cooperating for the accomplishment of the same purpose. This design methodology can be a powerful tool not only for developing autonomous vehicles, but also for evaluating the effectiveness of other designs such as the Draper Small Autonomous Aerial Vehicle.

Chapter 3

1997 DSAAV and SPU: Integration by Parts

3.1 Introduction

The first-generation Draper Small Autonomous Aerial Vehicle (DSAAV) was a preliminary attempt by the Charles Stark Draper Laboratory to develop a autonomous helicopter flight system. Unfortunately, while functional, the original DSAAV posed numerous dilemmas which made the vehicle unsuitable for long-term use. Plagued by a confused electromechanical and organizational architecture, the first aircraft served well as a proof-of-concept but not as a model for production. As a result, the Charles Stark Draper Laboratory began construction of a second-generation aircraft in 1997. The 1997 DSSAV, based on a commercially manufactured model helicopter, was designed to fly, hover, and maneuver autonomously between waypoints specified by a programmed flight plan. In the development of this system, distributed processing methodology was applied in order to facilitate its implementation, improve organization, and increase robustness. This led to the construction of the 1997 Sensor Processing Unit.

3.2 Overview of the 1997 DSAAV Distributed Systems

An aircraft is a complicated machine with a variety of interactions both mechanical, electrical, and directional. Under normal circumstances, the vehicle provides the mechanical and electrical structure while a human pilot acts as the directional head. In developing an autonomous aircraft, and especially helicopters, the machinery that replaces human guidance adds tremendous complexity that touches every aspect of the aircraft. By applying distributed processing methodology to these vehicles, a tangled web of interactions throughout the vehicle can be separated into subsystems that consolidate components and simplify the overall framework. This is key in autonomous vehicles since tighter, stronger electromechanical and computational architectures can better survive the hostile environments afforded by the aircraft and its surroundings that challenge both mechanical integrity as well as computational robustness in these machines. As a result, the 1997 DSAAV attempts to achieve a more cohesive and robust design by formalizing and implementing four major navigational sub-systems: on-board navigation, ground guidance, vision processing, and sensor processing.

3.2.1 On-Board Navigation

The 1997 DSAAV, like virtually all autonomous vehicles, has a centralized processing core dedicated as its directional head. Ideally modular in function, autonomous in operation, and cooperative with other vehicle control elements, the central processing unit (CPU) becomes a naturally defined sub-system. In terms of the DSAAV, the on-board CPU runs navigational algorithms for determining the most appropriate course of action within the vehicle's means to traverse from one point to another as specified by a flight plan. In addition, the on-board navigational CPU must issue the commands necessary to implement the desired actions. In the case of the 1997 DSAAV, a PC-104 compliant computer stack is the vehicle's navigational engine. With its Intel-type 486, 100 megahertz processor, the stack is adequate for the timely execution of all necessary navigational algorithms and commands. More quantitatively, optimal operation of the autonomous helicopter requires navigational algorithms be performed and commands issued at rate of 50 hertz. Since navigational commands only change with new environmental information, the optimal system frequency is tied directly to the optimal sensor data rate, and in fact are equivalent. Because vehicle hardware limitations dictate sensor data rates greater than 50 hertz as excessive[1], both the optimal data update and system control frequencies are 50 hertz.

3.2.2 Ground Guidance

In most autonomous vehicles, autonomy does not imply self-established purpose or self-awareness in the sense of artificial intelligence. These machine are autonomous only to the extent that the actual navigation is without human control. As such, the vehicle's mission or flight plan must be determined by human intelligence and as a result, requires some form of user interface. This interface becomes another readily established sub-system due to its conceptual modularity, autonomous operation, and cooperative interaction with the vehicle itself.

The DSAAV utilizes the Ground Control Station (GCS) to instruct the vehicle with

1. "Electronics Design for an Autonomous Helicopter", p.86 [2].

guidance commands based upon waypoints established by a human user. Beyond instituting these flight markers, the user has no part in controlling the actual flight of the vehicle. The ground computer solely determines the vehicle's course based on the user-specified destinations and transmits guidance information to the autonomous helicopter. Having received this instruction, the vehicle independently responds to these commands by adjusting its flight accordingly. The GCS also receives flight telemetry from the DSSAV and presents it for monitoring purposes. Everything from vehicle battery voltage to actual flight heading is relayed from the vehicle to the GCS for graphical presentation, thus facilitating and expediting interpretation of the vehicle's status for the human user.

3.2.3 Vision Processing

Vision processing is an advanced form of sensory perception that many autonomous vehicles do not incorporate due to its complexity and high production and computational cost. But, properly implemented, vision processing is a powerful method of observing the surrounding environment and, with the appropriate analytical tools, can be used for refining motion control. Due to the intensive computations typically required of image processing algorithms, the vision unit is established as a sub-system itself, separate from other sensors, in order to localize its calculational load, thus preventing an exorbitant centralized processing overhead. At bare minimum, a vision system incorporates a camera for capturing visual images, but should also be equipped the necessary electronics for image preprocessing. Without a self-contained computational engine, the visual processing must be conducted by another unit (most likely the main navigational computer), defeating the attempt to distribute the overall application load and circumventing distributed processing methodology. Unfortunately, though demonstrated with hardware in loop simulation, the 1997 DSAAV did not incorporate vision processing even though it had been planned to be included in the final design.

3.2.4 Sensor Processing

Without the environmental perception afforded by sensors, the DSAAV, nor any other autonomous vehicle, could hope to achieve flight independent of human control. Even people themselves could not function properly without some form of sensing ability. Indeed, a lack of sense is a key barrier to overcome in the development of any self-intelligent agent capable of motion. Consequently, sensory perception becomes central to sustaining autonomous movement for without this ability there would be no basis for evaluating motive actions with respect to the environment or, for that matter, any external frame of reference. In short, sensors allow machines to interact with their surroundings, and this interactive ability is an essential trait that must be possessed by all autonomous vehicles.

3.3 DSAAV Sensor Integration and Control

While all sub-systems play an integral part of the 1997 DSAAV, the Sensor Processing Unit (SPU) became the focus of much attention due to its integral role as the electromechanical "nervous system" for the vehicle. For the most part, the first generation aircraft subsystems follow a distributed architecture to a limited degree except for sensor processing. These functions are centralized within the on-board navigational computer thus burdening the CPU with sensor administration and taking away computational resources that could be used for evaluating complex flight algorithms. Furthermore, sensor errors are not easily localized since the aircraft's framework buries them within a sea of navigational code and twisted wires. But, as one of the most important evolutionary changes in the DSSAV design from the first- to second-generation models, the SPU has the potential to address these problems.

Basically, the Sensor Processing Unit isolates all necessary components for sensory perception into one device thereby creating a highly modular and autonomous machine dedicated to the delivery of environmental data to other navigational systems. By integrating three separate sensors, namely sonar altimeter, flux magnetometer, and inertial measurement unit, the SPU consolidates sensor functions providing the overall navigational system with a three-fold benefit: increased organizational order, reduced central processing load, and more robust design.

3.3.1 Overview of Sensor Processing Unit

Originally created by Christian A. Trott, the Sensor Processing Unit (SPU) results more as a natural consequence of an inertial system revision of the first-generation DSAAV than from a desire to achieve efficient distribution of systems [2]. Nevertheless, the SPU's effects and potential benefits are clear. While the first vehicle had centralized control of all sensors, the SPU of the second-generation aircraft takes on two major functions which remove sensor processing overhead from the on-board CPU, namely taking data from sensors and preparing that information for transmission to the central processor.

The SPU administers three sensors essential for the autonomous navigation of the DSAAV, the inertial measurement unit (or accelerometer), sonar altimeter, and flux magnetometer (or electronic compass). Manufactured by Systron-Donner, the IMU provides the system with linear acceleration data for the x, y, and z directions as well as angular acceleration data for rotation about the x-, y-, and z-axes. The sonar altimeter produced by Precision Navigation, Inc. projects ultrasonic pulses and captures the echoed signal to determine the aircraft's altitude. And lastly, the electronic compass, based upon the Polaroid Corporation's sonar ranging module, is a 2-axis magnetometer that senses the earth's magnetic field to calculate vehicle heading. These devices communicate their data to the SPU before being packaged for transmission to the on-board CPU via an RS-232 serial link. The SPU is designed to forward these packets at a rate of 50 hertz with new inertial data in each transmission (all other sensors are too slow). Assumed to be stable and accurate, this data rate is the central synchronizing mechanism as well as the limiting system rate for the entire navigational system as discussed previously in Section 3.2.1.

3.3.2 SPU Objectives

Convoluted architecture ran amuck on the original DSSAV and presented several difficulties that the next-generation 1997 vehicle sought to overcome. Since necessity directed much of the design process and available space along with weight distribution dictated component placement, the consolidation of devices into systems became a key issue to address in redesign, and whose proposed solution lead to the objectives for the Sensor Processing Unit (SPU).

By consolidating sensors and formalizing a sensor system, the SPU seeks to overcome the barriers presented by sensor processing on the first-generation aircraft. First and most obvious, greater electronic organizational order should immediately result thus simplifying the overall design. As was the case with the first-generation DSAAV, sensors would no longer be wantonly connected to both the aircraft and a central processor dedicated to handling all computational and directional tasks. Instead, these components would be drawn together into a single dedicated and cohesive sensing unit. This theoretically increases electronic robustness by disassociating sensors from the on-board CPU. Further, since the SPU hypothetically isolates all requisite functionality for proper sensor operation to itself, this localization should facilitate the identification and debugging of sensor errors since only the SPU need be examined, as opposed to the entire navigational system. By controlling and preprocessing sensor data autonomously, the SPU would unlink the devices from both dependency to and failures of a central processor. Finally, an autonomous SPU also has the advantage of distributing computational load away from the on-board CPU, the benefits of which come immediately from following a distributed processing methodology.

As part of an inertial system revision of the first-generation DSAAV, the SPU would also improve navigational performance by increasing both accuracy and bandwidth of sensor measurements. While the aircraft relies heavily on the Global Positioning System (GPS) for location data, GPS has also shown itself to be periodically unreliable due to environmental conditions or interference from the helicopter avionics. During such GPS failures, the inertial system readings become the main source of inertial data (in fact, vision processing was pursued since it could also be used to augment the navigational system by providing velocity data during

GPS faults). Consequently, any improvements to the inertial system readings would be a boon to the DSAAV especially amid GPS faults. Since inertial system measurements have drift associated with them, attempts at increasing accuracy could involve reducing drift error to improve the vehicle's positioning precision. Furthermore, any reduction of electromagnetic or mechanical noise in accelerometer readings would refine inertial system readings. In addition, boosting sensor bandwidth would provide obvious gains by allowing for an increased control system bandwidth thus improving the aircraft's maneuverability.

Thus, in summary, the 1997 DSAAV Senor Processing Unit (1997 SPU) objectives are:

- To consolidate and integrate sensors in order to simplify design, increase robustness, and facilitate debugging,
- To effectively distribute computational load away from the main navigational processor, and
- To increase both overall system accuracy and bandwidth.

3.4 1997 SPU Hardware Design and Implementation

3.4.1 Component Selection

3.4.1a Microprocessor Selection

The Sensor Processing Unit attempts to supplant the first-generation central computer in part by transporting the sensor processing load away from the central computer to microprocessors on the SPU. These microprocessors would execute numerous functions associated with sensor control and data handling formerly assigned to a single on-board CPU, as on the original DSAAV. Consequently, both microprocessor selection and implementation become a paramount consideration in SPU design since the microprocessors directly affect the distribution of sensor processing. While these devices can govern sensor administration, their limitations can also present restrictions on how effectively the SPU tackles sensor processing that has been distributed away from the on-board computer.

With this in mind, the PIC16C73 microcontroller manufactured by Microchip Technology, Inc. presents itself as a candidate component for the 1997 SPU. This integrated circuit is a low-cost, high-speed processor having a plethora of features including but not limited to a 5 Mhz instruction rate, 22 digital I/O ports, 5-channel analog-to-digital (A/D) converter, serial I/O, 192x8 bytes of data RAM, and reprogrammability [3]. Nevertheless, even with this multitude of options, an examination of the hardware requirements for the microprocessor clearly demonstrates how a single PIC16C73 proves insufficient for managing all SPU sensor functions.

Each of the three sensors that the SPU must control, the inertial measurement unit (IMU), flux magnetometer or compass, and sonar altimeter, have I/O pins which must be routed to the SPU processor in order to establish sensor control. The compass, with five control pins, and the sonar, with three control pins, require a total of eight digital I/O lines for operation, which a single PIC16C73 can easily accommodate. The IMU calls for six analog inputs, and the temperature and battery voltage measurements require an additional two. But unfortunately, a single PIC16C73 falls short of this need since each microprocessor only carries five analog-to-digital conversion channels. While this dilemma could be circumvented by using two microprocessors in tandem, the PIC16C73 A/D converter only has a resolution of 8-bits. Since a minimum resolution of 10-bits is necessary for reasonably accurate conversions, no number of PIC16C73 microprocessors would prove adequate. As a result, the SPU employs a separate analog-to-digital converter for translating IMU signals into discrete values.

3.4.1b A/D Converter Selection

Manufactured by Analog Devices, the AD7891 is an 8-channel, 12-bit A/D converter. While in a relatively large 44-pin PLCC package, the converter's other features, including a 1.6 microsecond conversion time and -5 to +5 voltage range, make it desirable for converting IMU signals. But, though the six IMU analog inputs can be easily accommodated by the AD7891, the SPU microprocessor must accept the converter's twelve digital output lines (one for each bit of resolution) as well as an additional four for control purposes. These sixteen inputs and outputs, in addition to the eight required by the compass and altimeter, saturate the available I/O resources of a single PIC16C73. Nevertheless, in this case, two such microprocessors working together would prove adequate for sensor management. This two-processor solution was implemented to create the directional and computational core of the 1997 SPU.

3.4.1c Anti-Aliasing Filter Selection

The inertial measurement unit (IMU) generates high-bandwidth data which can be problematic for A/D converters sampling at a relatively low rate. Specifically, unless removed from the IMU readings, high-frequency signal content typically produced by mechanical or electromagnetic noise will be aliased into lower frequencies leading to significant errors in the discrete IMU signal issued by the A/D converter. Consequently, IMU analog outputs require low-pass filtering prior to A/D conversion in order to prevent aliasing. To that end, a single-pole, low-pass filters with 3dB points well below that of the IMU sampling frequency prove more than adequate. But since the AD7891 A/D converter has low-impedance inputs, the IMU filters must accordingly be low impedance. Operational amplifiers meet both this qualification and the capability requirements for implementation of IMU anti-aliasing filtering. While most any op-amp could prove acceptable for the task, the LM614 stands as an ideal candidate due to its availability and accessibility.

The frequency cutoff of the anti-aliasing filters derives itself from the desired navigational rate of the DSAAV. While the original aircraft limited this rate to 5 hertz, the 1997 redesign set a 50 hertz navigational frequency goal. Consequently, any spectral content above 50 hertz from the IMU should be removed with filters whose cutoffs prevent aliasing of higher frequency signals during sampling. This seems valid since, in most cases, the choice of cutoff would simply be that of the highest rate of interest - 50 hertz in the case of the IMU. Of course, to further increase the navigational update rate and thus the inertial accuracy, a higher cutoff frequency could be chosen since the IMU provides data at up to ~1700 hertz. But, there is a trade off between navigational performance and IMU noise rejection in deciding upon filter breakpoints that is revealed with an analysis of the three main sources of inertial signal noise: internal IMU noise, vibration, and electromagnetic interference. As indicated by device specifications, the internal IMU noise has frequency content spanning a range of 10 hertz to 100 hertz [4]. Further, experiments involving the placement of the 1997 DSAAV on a shaker table revealed the second source of noise, vibration, occurs most strongly at a fundamental mode of 20 hertz and harmonics of that frequency. Finally, most appreciable electromagnetic interference is assumed to occur in relatively high frequency ranges, above and beyond both the spectrum for internal IMU noise and the fundamental vibrational mode. As a result, the only "noiseless" information in the 1997 DSAAV inertial system exists in the frequency band below 10 hertz. Therefore, a 50 hertz cutoff filter unavoidably passes noise whose sources include the internal IMU fluctuations and mechanical vibration. Note that the fundamental vibrational mode of the aircraft of 20 hertz is passed by single-pole 50 hertz filters, but by mechanically isolating the IMU from the chassis, the degenerative effects are diminished. So, while a 50 hertz breakpoint frequency does pass some noise, the relatively faster rate of sampling provides more inertial information since useful IMU data occurs in frequency bands up to ~1700 hertz.

3.4.1d Other Considerations

While not especially notable in form or function, connectors play an integral role in the proper functioning of the SPU and the DSAAV in general. Due to helicopter vibration, any component attached to the aircraft has the potential for being exposed to extremely high mechanical stresses. These forces not only burden the linkages and machinery associated with producing flight but also strain electrical connections between components. Therefore, due to their integral role as electronic interconnections, connectors must be very trustworthy being highly tolerant to vibration and resistant to breakage.

3.4.2 System Design

With these components in mind, the SPU can be broken down into roughly two sections that define its system layout and ultimately direct circuit design: the IMU, and the compass-sonar subsystems. As previously mentioned, two PIC16C73 microprocessors are needed for controlling sensors due to their digital I/O requirements and the limitations posed by a single PIC16C73. Each microprocessor is assigned to the control of one subsystem. Together, these processors cooperatively manage processing and transmission of sensory information to the on-board navigational CPU. Figure 3.4.2-1 depicts the organization of each subsystem and clearly indicates both the flow of control and the flow of data as well as illustrates the subsystem hierarchy. While both subsystems function autonomously to read sensor data, the IMU subsystem must transmit its data to the compass-sonar subsystem. The compass-sonar subsystem then packages that data along with the other sensor data for transmission to the on-board navigational computer. Finally, as an aside, the system battery voltage and IMU temperature are states that must be monitored by the SPU. Since they are for the detection of internal SPU conditions not directly related to navigation, they are identified with dotted lines and one-way arrows. Though the decision is arbitrary, monitoring of battery voltage and IMU temperature is accomplished by the compass-sonar subsystem and IMU subsystem respectively.

3.4.3 Circuit Design

3.4.3a Analog Design

The analog design for the 1997 SPU is relatively straight forward. Much of it is concerned with the anti-aliasing filters necessary for the IMU output signals. Figure 3.4.3a-1 shows the analog circuit schematic and clearly indicates the six IMU filters, three linear acceleration channels and three gyro channels. All filters are identical with approximately 50 hertz cutoff frequencies (as discussed in Section 3.4.1c) except that the accelerations channels include a 450 ohm resistor to convert the IMU linear acceleration current signals into voltage signals. While this resistor does affect the filter cutoff frequency, the change is only slight and therefore negligible.

3.4.3b Digital Design

In many ways, the digital design derives itself from the system design shown in Figure 3.4.2-1. Granted that other necessary components are included in the final circuitry that have not been discussed, but the major parts of the design including interconnect clearly present themselves in the final layout as shown in Figure 3.4.3b-1.

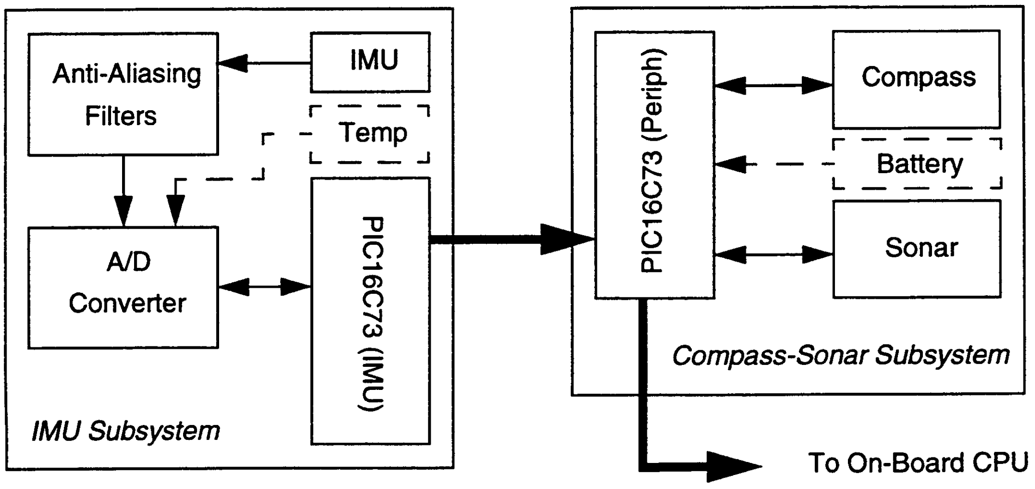
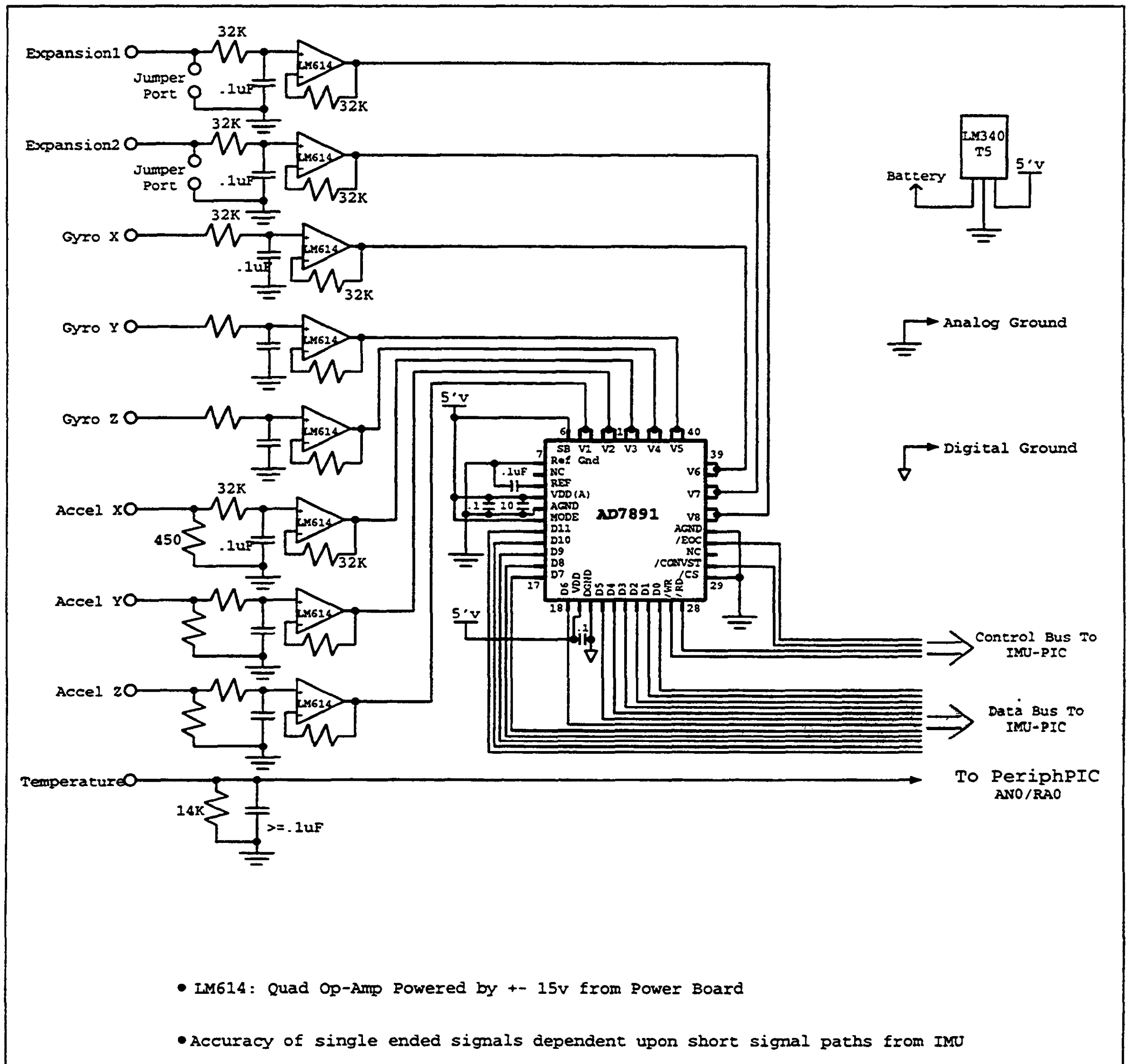


Figure 3.4.2-1: 1997 SPU System Diagram



- LM614: Quad Op-Amp Powered by +- 15v from Power Board
- Accuracy of single ended signals dependent upon short signal paths from IMU
- Offset Errors and Temperature Coefficients to be nulled in software

Figure 3.4.3a-1: 1997 SPU Analog Circuit Schematic [2]

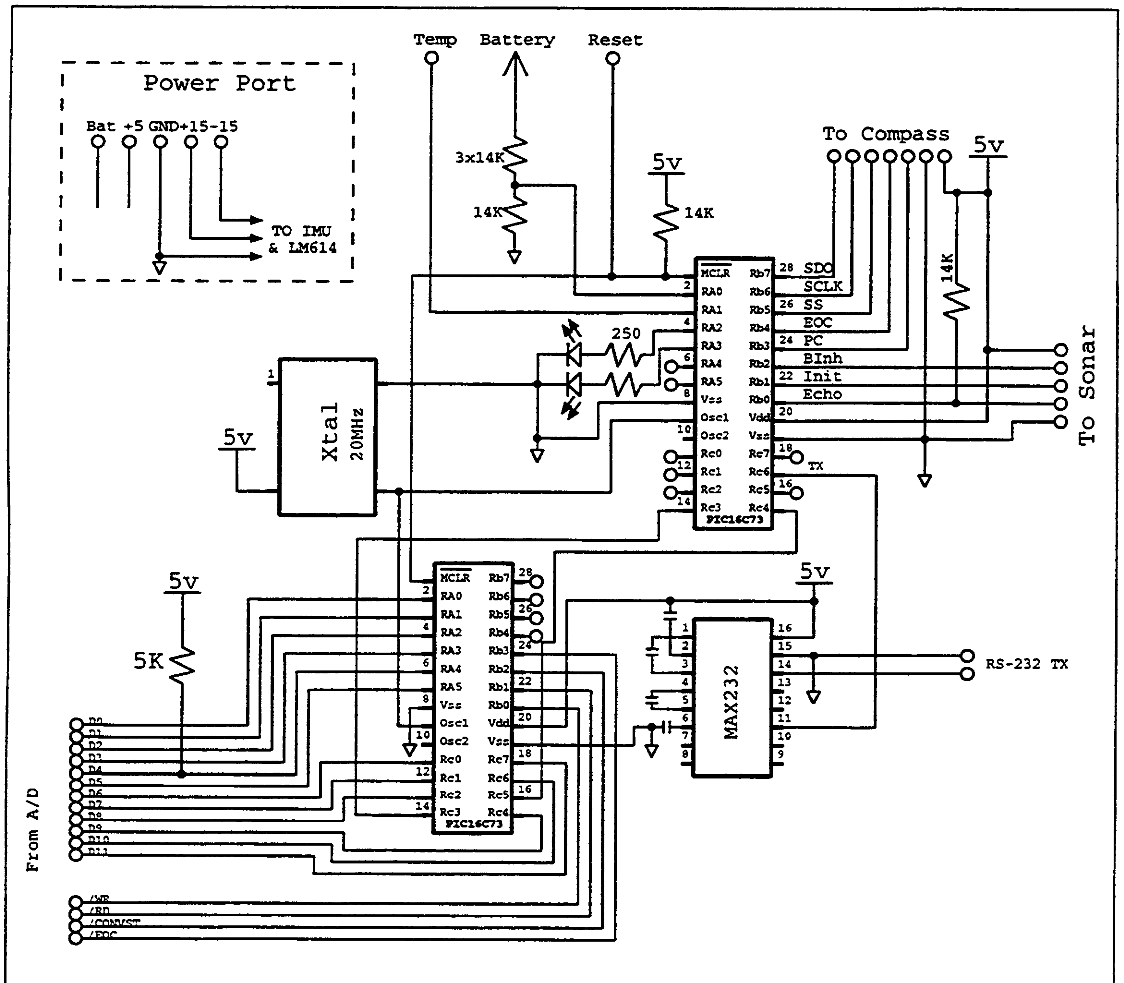


Figure 3.4.3b-1: 1997 SPU Digital Circuit Schematic [2]

3.5 1997 SPU Software Design and Implementation

While the SPU hardware is important, the software cannot be ignored since it is ultimately responsible for initiating control. The 1997 SPU uses PIC16C73 microcontrollers that can be programmed with a proprietary pseudo-assembly language developed by Microchip Technology, Inc. Using this language, appropriate initializations and functions can be executed on the microprocessors as dictated by the user. Since the 1997 SPU employs two PIC16C73 microcontrollers, each must be programmed according to its assigned function.

3.5.1 Compass-Sonar Subsystem Software

The compass-sonar subsystem microprocessor, or peripheral PIC, has code that can be broken into two major divisions. The first division is formed of the peripheral PIC interrupt routines. These interrupts occur as a result of the global millisecond timer, a sonar echo reception, a completed on-chip analog-to-digital conversion, a data transmission received from the IMU PIC, and a successful serial transaction with the on-board navigational computer. The second software division is the linear program which runs from line to line as directed by the main loop. This main loop conducts four different servicing features: compass polling, sonar polling, A/D conversion for the on-board battery voltage, and data transmission to the navigational computer.

At start-up, the peripheral PIC software runs the appropriate initializations and variable declarations before entering the main loop. In this loop, each service routine is called when allowed by its service flag. During a millisecond interrupt, all flags are set to request servicing, and counters keeping track of elapsed time since the beginning of each service cycle are incremented. Consequently, servicing is requested for each of the four main routines every millisecond, even if no service is to be done at that time. After a service is requested, the specific service phase to execute (if any) can be determined by its counter since the elapsed time within a given service period indicates which phase to conduct. Once completed, the routine cancels the service flag and lies in wait for the next interrupt to set the marker again. Upon the conclusion of the entire routine sequence (i.e., all phases have been discharged) and once the service period elapses, the next service cycle is initiated. This control scheme applies to all four service routines and can be seen in Figure 3.5.1-1.

The compass polling routine, named `Service_Compass` in the peripheral PIC software, is basically a chain of if-then statements. Each if clause compares the compass timer value with a phase event time to determine if that phase event has been reached and needs to be executed. For example, if the compass timer indicates five milliseconds have passed since the beginning of a compass service period, then the microcontroller would identify that as the appointed time to clear the compass PC line from the corresponding if-then conditional. But, if the compass timer is at six milliseconds, then all the if-then statements would return false since six milliseconds is not associated with any compass service phase and is therefore non-significant.

For the most part, the sonar polling routine, named `Service_Sonar`, works in the same manner as `Service_Compass`. Each service phase of the sonar cycle is executed at the proper time as indicated by the sonar timer. The one major difference is that the complete sonar service is divided between the `Service_Sonar` function and an interrupt handling subroutine called `Echo_Int`. When triggered by an ultrasonic echo received by the altimeter, this interrupt quickly begins the determination of the echo arrival time. Without this interrupt, the echo signal can only be recognized when the software bothers to check and this might occur well after the fact thus causing altitude reading errors.

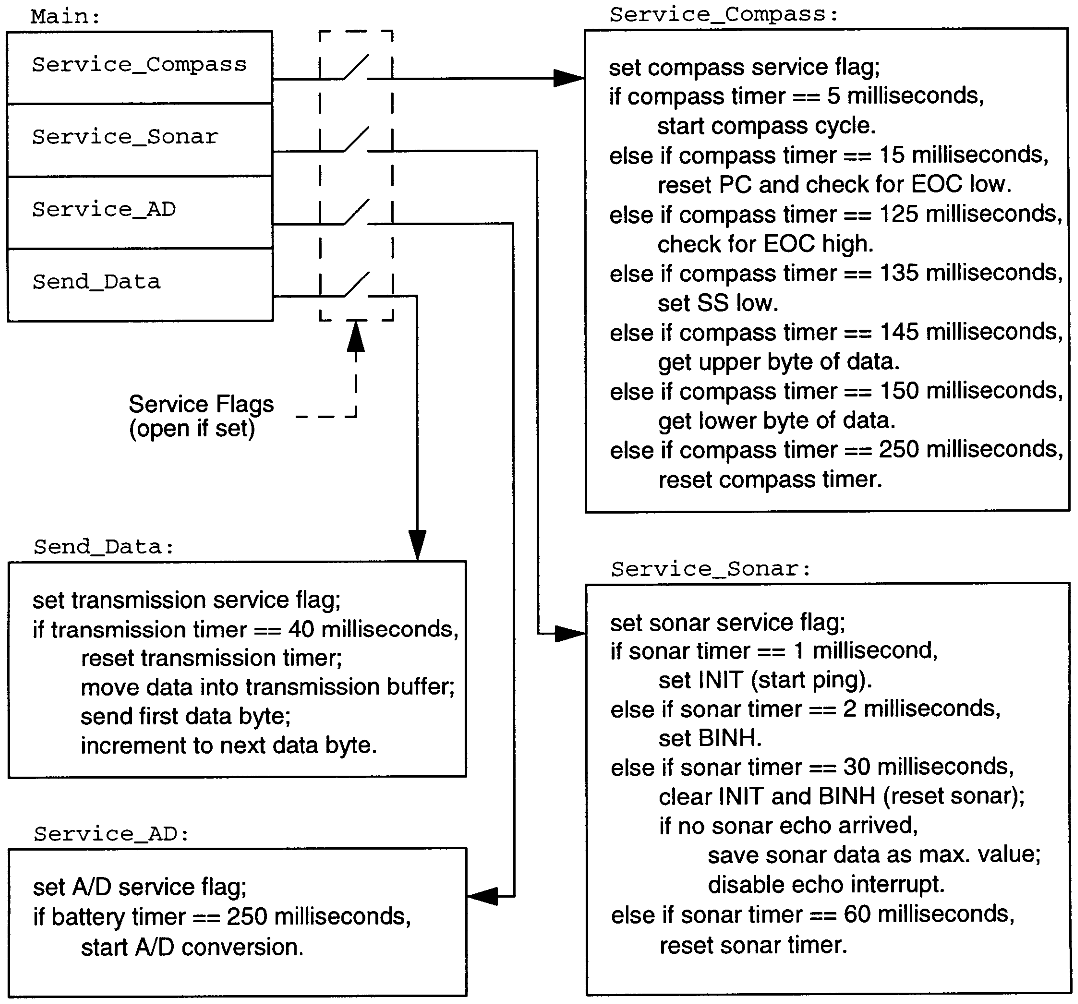

Main:
Service_Compass
Service_Sonar
Service_AD
Send_Data
Service Flags (open if set)
Service_Compass:
set compass service flag;
if compass timer == 5 milliseconds,
start compass cycle.
else if compass timer == 15 milliseconds,
reset PC and check for EOC low.
else if compass timer == 125 milliseconds,
check for EOC high.
else if compass timer == 135 milliseconds,
set SS low.
else if compass timer == 145 milliseconds,
get upper byte of data.
else if compass timer == 150 milliseconds,
get lower byte of data.
else if compass timer == 250 milliseconds,
reset compass timer.
Send_Data:
set transmission service flag;
if transmission timer == 40 milliseconds,
reset transmission timer;
move data into transmission buffer;
send first data byte;
increment to next data byte.
Service_AD:
set A/D service flag;
if battery timer == 250 milliseconds,
start A/D conversion.
Service_Sonar:
set sonar service flag;
if sonar timer == 1 millisecond,
set INIT (start ping).
else if sonar timer == 2 milliseconds,
set BINH.
else if sonar timer == 30 milliseconds,
clear INIT and BINH (reset sonar);
if no sonar echo arrived,
save sonar data as max. value;
disable echo interrupt.
else if sonar timer == 60 milliseconds,
reset sonar timer.

Figure 3.5.1-1: 1997 SPU Compass-Sonar Service Control Diagram

Battery voltage conversion and serial transmission to the navigational computer also function very similarly. In both cases, at the beginning of their respective service cycle periods, the service is simply initiated. For the battery voltage, this means that the on-chip analog-to-digital converter begins a sample and conversion of the battery voltage level. For the serial transmission, all data is copied into a transmission buffer and the first data byte is sent. Futhermore, both routines require the use of separate interrupts to handle completion of the service, similar to the sonar. The conclusion of an A/D conversion triggers an interrupt that saves the voltage data, and the successful communication of a data byte to the on-board computer begins the transmission of the next byte available. All this occurs as a result of interrupts and not by looped code.

Obviously, interrupts play an integral part in the proper functioning of the peripheral PIC. They act in the background to tackle much of the work involved in polling sensors and transmitting data, but are not necessarily part of any linear code progression or software loop. As Figure 3.5.1-2 illustrates, they are independent routines that respond to events. For example, the IMU received data interrupt is one function that the peripheral PIC directs, but it is not a part of the main service loop. Executed upon the reception of IMU data, the `IMU_int` interrupt simply stores this information upon its receipt from the IMU subsystem.

3.5.2 IMU Subsystem Software

The IMU subsystem microprocessor, or IMU PIC, follows the same control methodology as the peripheral PIC. The software is divided into two sections: the interrupt routines and the linear program. Of the two interrupts, one is a 30.2 microsecond global timer and the other is a serial transmission interrupt. The linear program contains a single servicing feature for collecting IMU data from the A/D converter. Both interrupt and service control diagrams appear in Figure 3.5.2-1.

As with the peripheral PIC, initializations and variable declarations are executed upon startup before beginning the main loop. Once in this loop, the `Read_Data` service routine is called if its corresponding service flag has been raised. This flag is set upon each 30.2 microsecond timer interrupt and, during the interrupt, a sampling counter is incremented. This sampling counter determines which event phase is to be executed in the `Read_Data` subroutine. In the case of the sampling counter, all significant, event-associated values initiate the sampling of a given A/D converter channel, and each channel is read in succession one after each timer interrupt. Once all the channels have been read, non-significant sampling counter values are traversed until the service period of 0.635 milliseconds has elapsed (equivalent to 20 timer interrupts) and the cycle begins anew. This sampling period implies that all converted IMU channels and temperature are read and stored consecutively at a rate of 1.6 kilohertz.

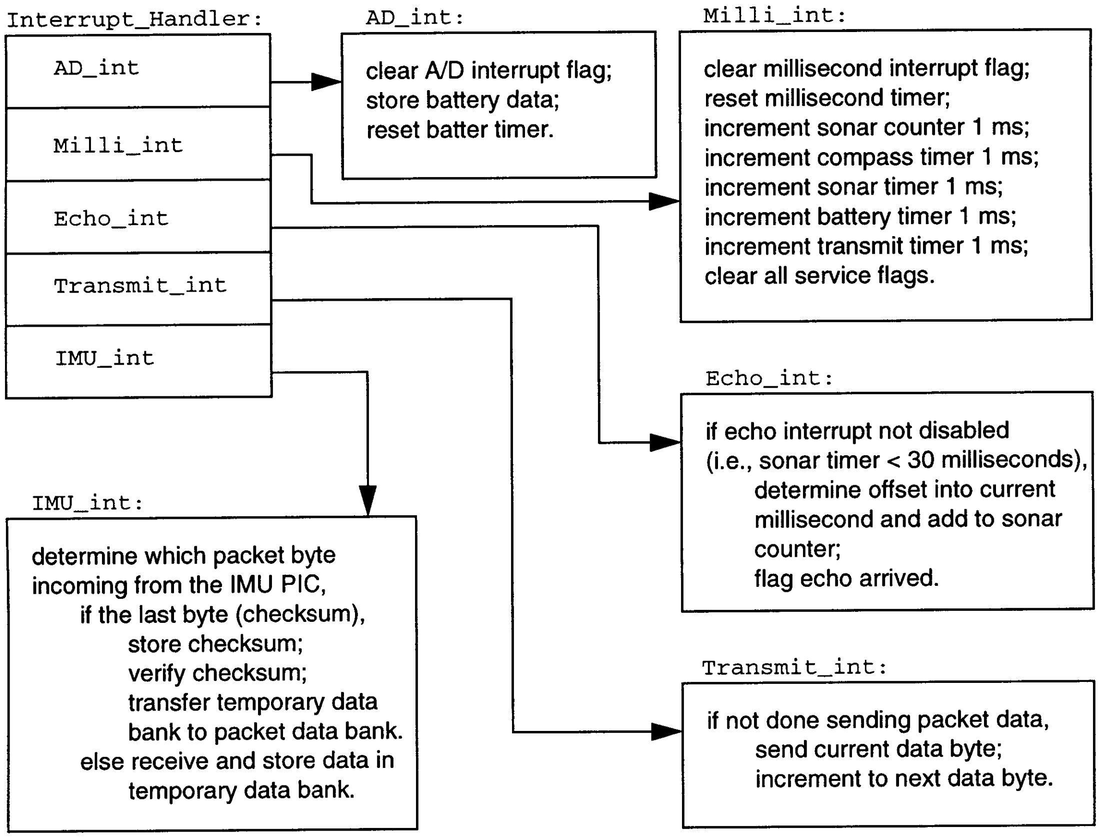

Interrupt_Handler:
AD_int
Milli_int
Echo_int
Transmit_int
IMU_int
AD_int:
clear A/D interrupt flag;
store battery data;
reset batter timer.
Milli_int:
clear millisecond interrupt flag;
reset millisecond timer;
increment sonar counter 1 ms;
increment compass timer 1 ms;
increment sonar timer 1 ms;
increment battery timer 1 ms;
increment transmit timer 1 ms;
clear all service flags.
Echo_int:
if echo interrupt not disabled
(i.e., sonar timer < 30 milliseconds),
determine offset into current
millisecond and add to sonar
counter;
flag echo arrived.
Transmit_int:
if not done sending packet data,
send current data byte;
increment to next data byte.
IMU_int:
determine which packet byte
incoming from the IMU PIC,
if the last byte (checksum),
store checksum;
verify checksum;
transfer temporary data
bank to packet data bank.
else receive and store data in
temporary data bank.

Figure 3.5.1-2: 1997 SPU Compass-Sonar Interrupt Control Diagram

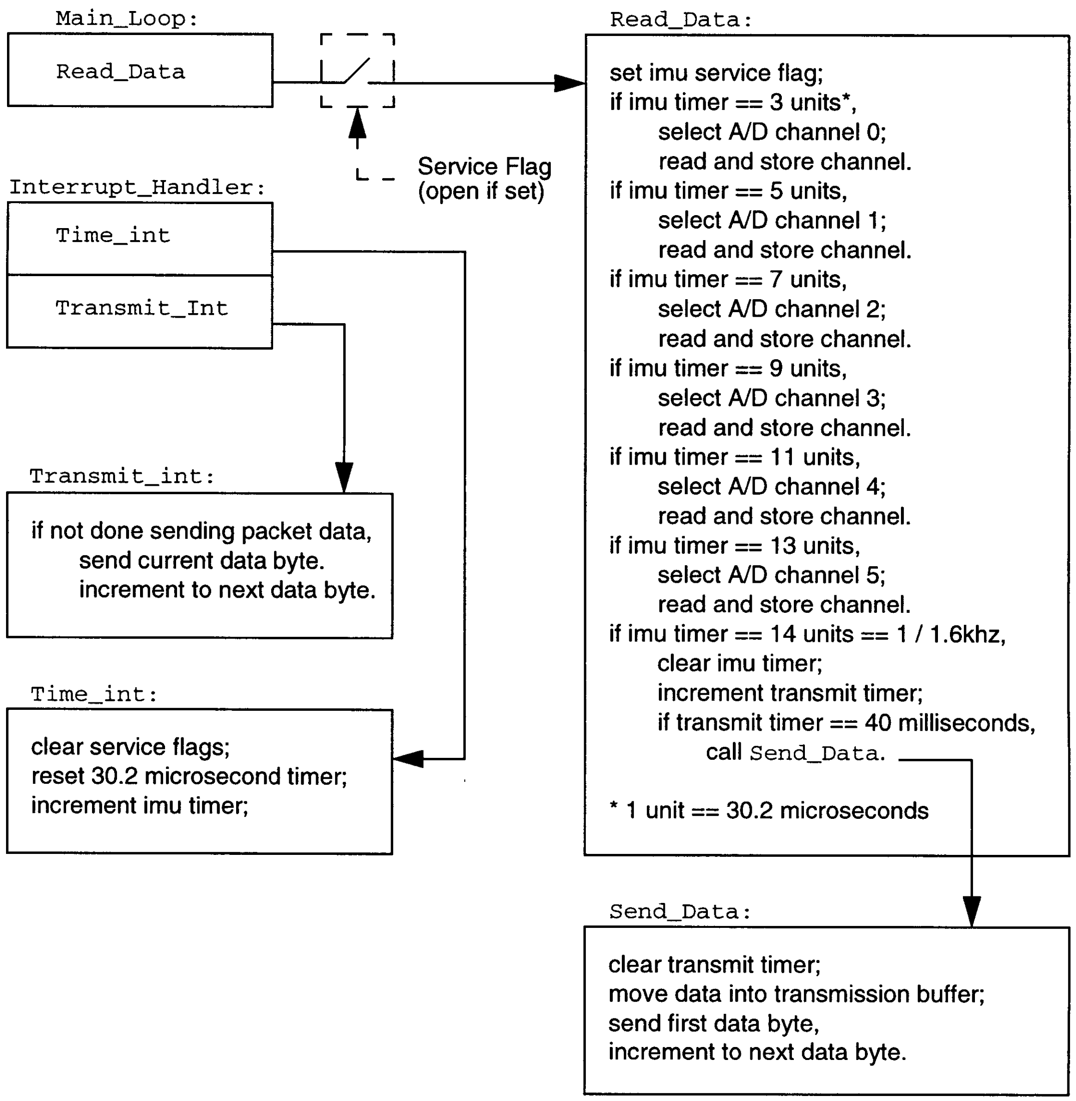


Figure 3.5.2-1: 1997 SPU IMU Service and Interrupt Control Diagram

Unfortunately, the Read_Data function effectively downsamples this 1.6 kilohertz rate by incrementing a transmission timer such that IMU data is only transmitted to the peripheral PIC every 40 milliseconds. Once the first byte of data has been transmitted, each successful serial transaction between the IMU and peripheral PICs triggers an interrupt that begins the transmission of the next available byte until all information has been sent. Of course, this occurs rapidly enough such that the time to send all data bytes is much less than a transmission period. Once completed, the next transmission commences after the 40 millisecond period has elapsed. Consequently, the IMU PIC transmits new inertial data to the peripheral PIC at a rate of 25 hertz. This transmission rate is different from the 1.6 kilohertz sampling frequency accomplished between reading successive IMU channels. The actual rate of new IMU data arrival for all A/D conversion channels from the IMU PIC to the peripheral PIC is 25 hertz. Of course, this is contrary to the specified 50 hertz rate mentioned in Section 3.4.1c and is a fundamental design flaw.

3.6 Summary

The 1997 Draper Small Autonomous Vehicle stands as an attempt at developing a truly autonomous vehicle. With its many functions, the aircraft applies distributed processing techniques to organize its various mechanical, electrical, and algorithmic components into coherent subsystems thus achieving "integration by parts" - the collection of hardware, software, and architectural ingredients to form an aerial vehicle capable of independent flight. The 1997 Sensor Processing Unit is such an ingredient that endeavors to integrate navigational sensors into a single device that removes the burden of sensor processing away from the on-board navigational computer and increases overall system accuracy and bandwidth. But, further examination of the 1997 SPU's organization and implementation reveals much to be desired.

Chapter 4

Evaluating the 1997 SPU: Cracks in the Wall

4.1 Introduction

The 1997 Sensor Processing Unit is a first attempt at integrating all the DSAAV navigational sensors and their administration into a single device. Unfortunately, the actual 1997 SPU falls short of meeting its objectives. These failures are revealed by evaluating the 1997 SPU design as well as its organization.

4.2 Evaluating 1997 SPU

The 1997 Sensor Processing Unit implements both the hardware and software plans discussed in the previous chapter. These schemes can be evaluated to determine the effectiveness of the final design by comparing the product and its performance against the stated objectives for the SPU which are restated below:

- To consolidate and integrate sensors in order to simplify design, increase robustness, and facilitate debugging,
- To effectively distribute computational load away from the main navigational processor, and
- To increase both overall system accuracy and bandwidth.

4.2.1 Consolidation and Integration of Sensors

The Sensor Processing Unit consolidates electronics by confining or attaching all sensor-related components to a single printed circuit board (PCB). Using commercial software, all SPU constituents (i.e., microprocessors, op-amp filters, connectors, etc.) can be mapped efficiently and effectively to minimize space consumption prior to construction as well as ease the location and identification of parts after construction. This reorganization simplifies vehicle layout and enhances robustness since sensors are no longer randomly connected to either the aircraft or the on-board navigational computer. Also, since PCBs reduce and confine interconnection wires, electromechanical robustness is further improved over that of the first-generation DSAAV. Additionally, as problems are discovered in testing and preparing the DSAAV for flight, sensor debugging is aided by the physical isolation of the SPU and its associated components from other on-board systems. Indeed, from a hardware perspective, the 1997 SPU achieves the goal of consolidating electronics thus simplifying design, increasing robustness, and facilitating debugging.

From a software perspective, the SPU misses the mark. While sensor hardware has been effectively integrated into one central apparatus, much of the software is considered unified simply by virtue of being on board the SPU. Granted that this might have been the case with a single microprocessor, but the SPU has its code divided between two separate PIC16C73s. This immediately adds a programming overhead for achieving interprocessor communications as well as unnecessarily confusing overall code structure. And while sharing software between microprocessors could provide legitimate benefits by further distributing sensor processing load, this cannot occur as a matter of fact, but rather requires thoughtful planning. In the case of the 1997 SPU, the software split cannot be justified as a conscious design decision since it comes only as a consequence of using two microprocessors to alleviate the I/O insufficiencies of one. Ultimately, the choice to use multiple PIC16C73s reflects the hardware advantages alone, ignoring both the software advantages and disadvantages. As a result, the SPU software acts as a fix or work-around solution, but not as an integrated part of the design process. The code is divided, not distributed, and fails to meet the objective of sensor consolidation and integration at the software level.

4.2.2 Distribution of Sensor Processing Load

From a purely hardware systems perspective, the 1997 SPU successfully distributes sensor processing load away from the navigational computer and stands in stark contrast to the completely centralized processing structure of the first-generation DSAAV. But, the 1997 SPU and the on-board navigational software together produce unfortunate limitations on the vehicle.

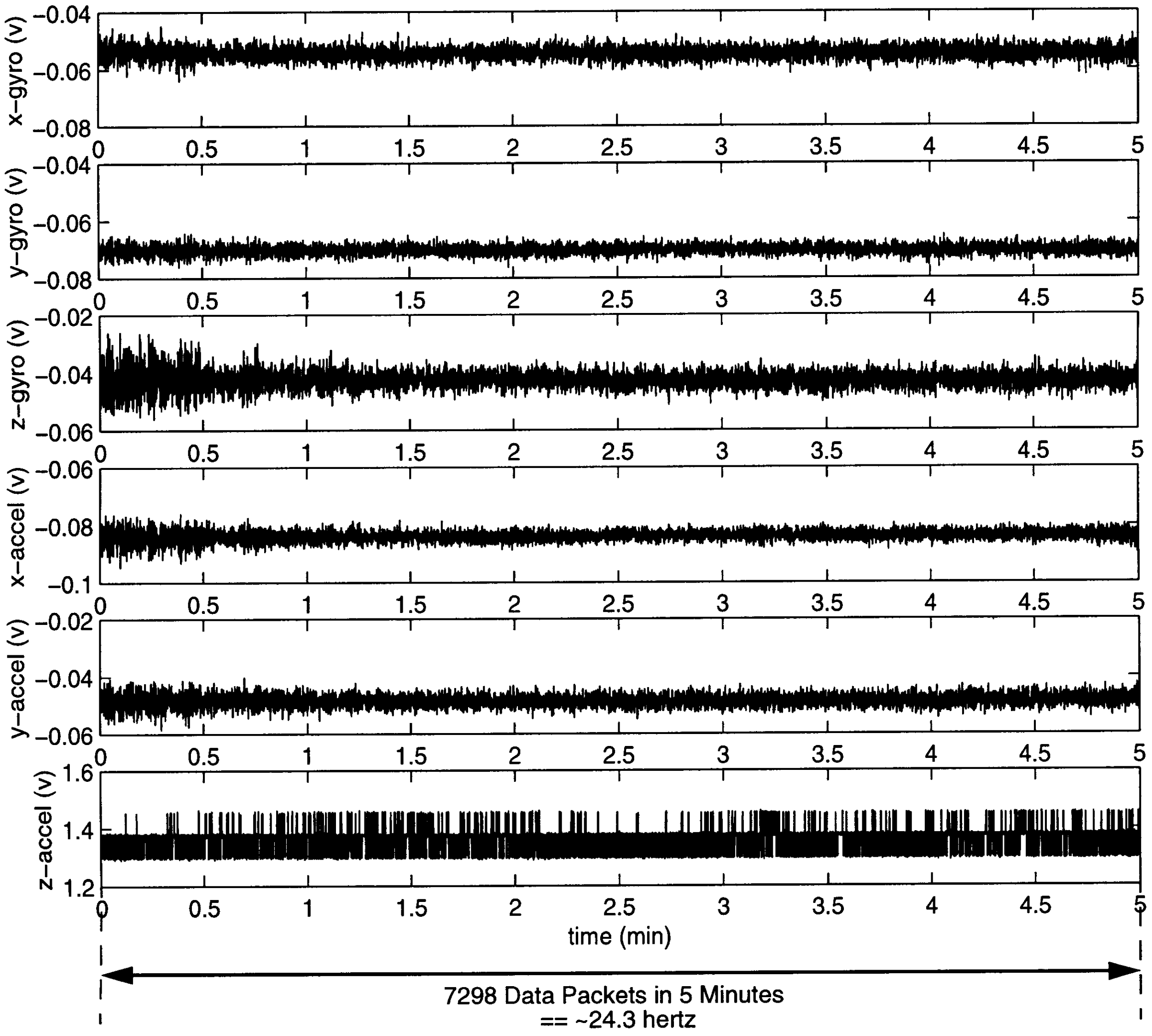
As mentioned in Section 3.3.1, the SPU is designed to transmit data packets to the navigational computer at an optimal rate of 50 hertz with new inertial data in each packet. This data rate is the central timing mechanism for the entire flight control software. Consequently, any adjustments, deviations, or errors in the SPU must be accounted for in the navigational code since the software relies on a pre-established data rate in calculating flight commands. There are fundamental problems with this avionics control approach, namely that both the modularity and autonomy of distributed systems are circumvented. The dependency of the navigational computer on a pre-determined data rate of the SPU produces problems if either the SPU is removed or if the SPU produces a data rate other than prescribed. Granted that the on-board computer can function without the SPU, but it can only do so by completely ignoring the SPU. In doing so, the on-board computer ceases to be navigational, and solely computational. Consequently, the on-board computer is neither modular nor autonomous as an avionics control computer. On the other hand, being able to function independent of a navigational CPU and any other system, the 1997 SPU is both modular and autonomous. Nevertheless, the unit fails to operate under the agreed system framework; it must output data at a rate of 50 hertz. In fact, testing reveals that the 1997 SPU only transmits information packets at rate of ~24.3 hertz, below half that deemed ideal (see Figure 4.2.2-1)! As a result, the on-board navigational computer must not only be recalibrated, but it also cannot run the flight algorithms at the optimal rate of 50 hertz since it is not receiving inertial updates at 50 hertz. Obviously, cooperativeness - the third characteristic of distributed systems - has been hindered by the 1997 SPU since the unit fails to abide by the operation terms established between itself and the on-board computer.

To further aggravate the situation, the downsampling of the IMU 1.6 kilohertz conversion rate by none other than the IMU PIC itself incurs an absolutely unnecessary run-time overhead caused by excessive sampling and converting, and then resampling. Simply put, the IMU PIC software could have read and transmitted the IMU channel data at 25 hertz instead of oversampling at 1.6 kilohertz and then downsampling by sending that information at 25 hertz.

In examining the distribution of sensor processing load, the 1997 SPU has both its successes and failures. While the device has succeeded in moving sensor control away from the navigational computer, the 1997 SPU as well as the on-board CPU itself have managed to defeat all three characteristics of distributed systems.

4.2.3 Augmentation of System Accuracy and Bandwidth

The SPU seeks to improve overall system accuracy and bandwidth by decreasing errors in inertial system measurements. In order to accomplish this, a model for inertial error can be developed to compare the differences in accuracy and bandwidth between the 1997 SPU implementation and a proposed 1998 SPU design.



All data transmitted at the same time belong to the same data packet transmitted by the SPU to the navigational computer.

Figure 4.2.2-1: 1997 SPU Transmitted IMU Data

4.2.3a Modeling Inertial Errors[1]

Sources for error which reduce inertial sensor accuracy include: inherent sensor noise (P_S), mechanical vibration (P_V), electromagnetic interference or EMI (P_{EMI}), sampling quantization (P_Q), integration errors (I_E), instrument non-linearity, sensor bias drift, and miscalibration. The last three errors, assuming proper calibration and software corrections, can be assumed negligible and ignored in quantifying accuracy. But, the remaining error types, P_S, P_V, P_{EMI}, P_Q,

1. "Electronics Design for an Autonomous Helicopter", pp. 73-76 [2].

and I_E, can be examined probabilistically to determine their behavior, and how best to reduce noise effects on inertial precision.

Inherent sensor noise, mechanical vibration, and EMI are assumed to be continuous, independent, and gaussian in distribution with zero mean. The accumulated drift error due to these noise sources is represented by a simple function:

$$T \cdot \left(\sum_{n=0}^{K-1} (P_S[n] + P_V[n] + P_{EMI}[n]) \right)$$

(T = Sample Period, K = Number of Samples)

Consequently, their sum can also be considered gaussian with zero mean whose standard deviation is defined as σ_C. Thus, the combined drift error of inherent sensor noise, mechanical vibration, and electromagnetic interference can be characterized as:

$$\sqrt{KT}\sigma_C$$

(T = Sample Period, K = Number of Samples)

where KT represents a period of time.

Sampling quantization errors are differences between analog signal values and their converted digital values that result from rounding losses inherent with any finite precision analog-to-digital conversion. Values for quantization error run between $-V_{MAX}/2^N$ and $+V_{MAX}/2^N$ where V_{MAX} is the maximum voltage output and N is the number of bits per discrete value. Since none of the infinite quantization error values is more likely than another, the probability density function (PDF) for this type of noise is uniformly distributed over a quanta with a value of $2^{N-1}/V_{MAX}$. Consequently, the quantization error PDF has zero mean and an a standard deviation of:

$$\sigma_{P_Q} = \frac{L \cdot V_{MAX}}{\sqrt{3} \cdot 2^N}$$

(L = Constant [relates sensor voltage to angular velocity or acceleration, V_{MAX} = Maximum Voltage Output, N = # of bits per discrete value)

Similar to the previous errors analyzed, accumulation of quantization noise leads to further drift error characterized by:

$$\sqrt{KT}\sigma_{P_Q}$$

(T = Sample Period, K = Number of Samples)

Finally, integration errors result from differences between the continuous-time inertial signal $(\tilde{P}(t))$ and its discrete-time representation. The discrete signal ignores the continuous-time values between samples, and this lost information accumulates to generate an integration error. Assuming that the slope of the continuous-time signal is a gaussian random variable, then the standard deviation of that slope can be defined as σ_δ. Granted, consecutive slope measurements are not statistically independent meaning positive and negative accumulation of errors cancel, but some σ_δ exists such that integration contributes to drift error and can be described as:

$$\frac{1}{2}T^2\sqrt{K}\sigma_\delta = \frac{1}{2}T^{\frac{3}{2}}\sqrt{KT}\sigma_\delta$$

(T = Sample Period, K = Number of Samples)

With these five error type and their contributions to drift error determined, the net effect can be computed by summing drift sources. The total drift error is modeled by:

$$\sigma^2_{\tilde{I}[k]} = KT\left(\sigma^2{}_C + \sigma^2{}_{P_Q} + \frac{1}{4}T^3\sigma^2{}_\delta\right)$$

Clearly, this equation identifies several possible methods for reducing drift errors. One of the most notable involves increasing the sampling frequency and thus decreasing the sampling period T. By raising the sampling rate, contributions to drift error by integration are minimized. Further, increasing the number of bits per discrete value generated by analog-to-digital conversion immediately reduces σ_{P_Q} resulting in reduced overall drift. And while internal sensor noise can only be diminished with difficulty, both mechanical vibration and electromagnetic interference contributions to drift error can be lessened by mechanical dampeners and electronic filters respectively.

In summary, the three possible measures for improving inertial system accuracy by decreasing drift error are:

- Increasing the sampling frequency of the inertial measurement unit,
- Increasing the number of bits in analog-to-digital conversion, and
- Utilizing filters to reduce mechanical and electromagnetic noise.

Each of these become evaluating guidelines for both the hardware and software of the 1997 SPU since they are directly concerned with improving sensor accuracy, and thus system accuracy as well.

4.2.3b Applying the Error Model to the 1997 SPU

In comparing the proposed SPU design against the actual 1997 realization, the 1997 SPU clearly does not meet the specified design objectives and, as a result, fails to achieve the best theoretical inertial system accuracy possible as can be determined by the error model previously developed. The proposed SPU design establishes a baseline data sampling rate performance of 50 hertz specifically intended to ensure inertial data updates of 50 hertz. To that end, the actual 1997 SPU presents two difficulties. The IMU PIC has been programmed in software to transmit inertial data at only 25 hertz, and the peripheral PIC reaches its maximum transmission rate at ~24.3 hertz. Since higher inertial sampling rates increase system accuracy and the 1997 SPU has clearly fallen below specifications, the 1997 SPU has yet to achieve the error reduction possible to attain better inertial accuracy.

In addition to flaws that reduce the IMU data update rate and peripheral PIC transmission frequency, the 1997 SPU anti-aliasing filters poles have been revealed to be at the wrong cutoff frequency for the IMU sample rate encoded in the IMU PIC. For any signal whose spectrum is band-limited to F, the Nyquist criterion specifies a sampling resolution of at least twice F in order to completely capture all frequency data contained in that signal. Consequently, since the IMU data rate has been verified both in software and empirically to be ~25 hertz, the filter cutoff points should consequently be at 12.5 hertz to prevent aliasing of frequencies higher than the breakpoint. But, since the anti-aliasing filter poles lie at 50 hertz, a frequency four times greater than the maximum 12.5 hertz allowed, the spectrum between 12.5 and 50 hertz is aliased into the sampled IMU signal thus generating significant inertial measurement errors. Further, this aliasing error is compound by the additional internal, electromechanical, and vibrational IMU noise that passes through due to filter poles which are unnecessarily and incorrectly placed at 50 hertz.

Finally, extended bench tests monitoring the transmission of 1997 SPU data packets revealed the presence of random impulses of bad data (see Figure 4.2.3b-1) even with SPU software safeguards i.e., data checksums. Unfortunately, during experiments both in the lab and some in flight, these bad packets have communicated sensory misinformation to the navigational computer causing erratic responses. In the field, such chance happenings can result in extremely dangerous, unpredictable, and wild flight behavior. So, as an extra precautionary measure, the navigational computer could scan incoming data and reject those with bad data bytes. Though slight, this solution unfortunately presents a two-fold disadvantage. First, the on-board flight computer must take some of the sensor processing burden, against the tenants of distributed processing methodology. Second, the navigational CPU loses an entire packet of data even if only one bit is in error and consequently, the data rate is reduced. This also decreases system accuracy since inertial information is lost.

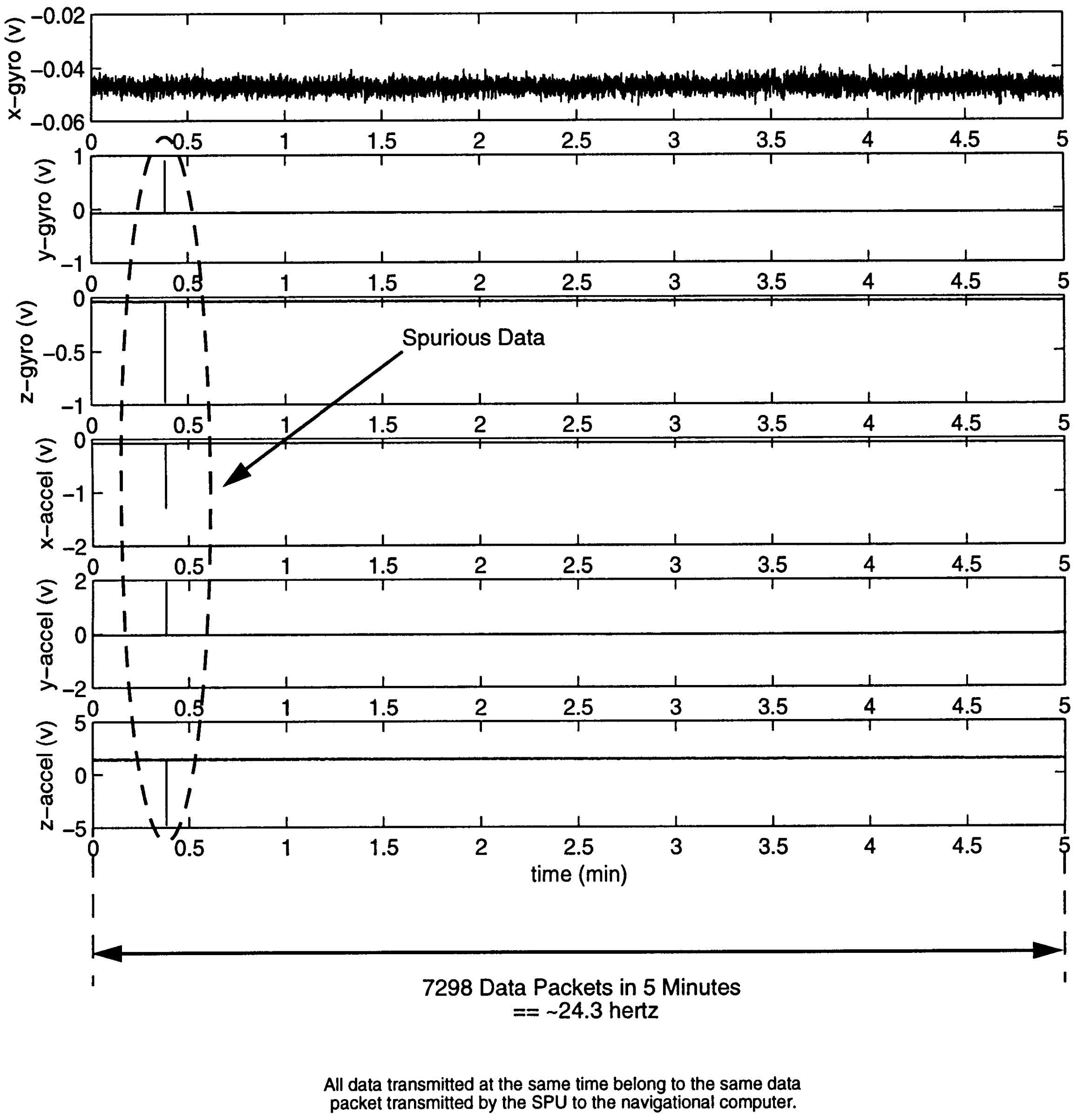


Figure 4.2.3b-1: 1997 SPU Transmitted IMU Data w/ Errors

As far as can be determined, the spurious data packets may be caused by out-of-phase communications between the IMU and the compass-sonar subsystem PICs. The 1997 SPU attempts to correct for these by sending extra, non-relevant bytes from the IMU PIC to prepare the compass-sonar PIC for data reception. Interestingly, these "empty" bytes do not appear at the compass-sonar subsystem even though they are indeed sent by the IMU subsystem, and are a mystery. But unfortunately, they still do not completely correct for the bad data problem.

4.3 Summary

In examining and evaluating the structure, performance, and implementation of the 1997 Sensor Processing Unit, cracks in the wall of its design clearly present themselves. Since much of the 1997 SPU architecture resulted as a mere a consequence of the inertial system revision, the SPUs development process unfortunately defaulted to one in which the immediate needs defined the goals for the next design step. This very short-sighted design philosophy does not approach the SPU with all its objectives and control nuances from a holistic perspective, and produces a product with excessive components in both hardware and software. The 1997 SPU design process clearly demonstrates a bias toward hardware advantages and blindness toward software concerns, and is further frustrated by a lack of adherence to proper distributed processing methodology by both the 1997 SPU and the on-board navigational software. Obviously, a definite need for SPU redesign exists.

Chapter 5

1998 Sensor Processing Unit: Building Blueprints

5.1 Introduction

Evaluation of the 1997 SPU reveals several design deficiencies which must be addressed in its revision. As discussed in Chapter 4, these shortcomings occur in three categories: consolidation and integration of sensors, distribution of sensor processing load, and augmentation of system accuracy bandwidth. In reviewing the 1997 SPU sensor integration, while the hardware implementation proves somewhat successful, this happens at the expense of dividing the SPU software thus adding interprocessor communication overhead and also confusing the overall code structure. Furthermore, distributed processing methodology is violated by both the 1997 SPU and the navigational computer. While the correction of the navigational computer's dependency upon the SPU transmitted data rate is beyond the scope of this thesis, the 1997 SPU's failure to sustain a 50 hertz data packet transmission rate make the device a prime target for refit. Finally, an insufficient IMU update rate, incorrect anti-aliasing sample rates, and spurious data packets have been recognized by an inertial error model and other criterion as hindrances to inertial system accuracy and only add to the growing list of reasons for the revision of the 1997 SPU.

But, the revision of electronic sensor integration for the 1997 DSAAV should ultimately lead to a proposal for the 1998 Sensor Processing Unit that seeks to accomplish more than simply fix the errors discovered in its predecessor. Throughout the 1997 SPU's development, plans for SPU realization came as a result of linear problem solving where one design decision led to another, but not necessarily a best solution. In fact, the 1997 SPU resulted as a side effect of the first-generation inertial system revision and clearly not as a direct fruit of a systematic approach to SPU design. Especially in software development, the final 1997 SPU implementation was dictated by the decision to use a dual-microprocessor computational engine which ultimately ignored the effects upon overall SPU architecture. Consequently, the 1998 Sensor Processing Unit establishes its objectives with the intent of not only correcting the errors of its forerunner but also attacking the SPU design by iteratively analyzing the systematic effects of design choices in both hardware and software, with the hope of ultimately arriving at a best, overall SPU system arrangement.

5.2 Objectives

At the most fundamental level, the 1998 Sensor Processing Unit aspires to fulfill nothing less than all of the original SPU objectives: the consolidation and integration of sensors, the effective distribution of sensor processing load away from the navigational computer, and the increase of system accuracy and bandwidth. Each of these goals represents a basic characteristic of any SPU or similar device, but the 1998 SPU revision aspires to reach these objectives with a holistic approach to system design contrary to the linear methodology followed for the 1997 SPU.

Though very different in nature, both the SPU hardware and software contribute towards the achievement of the same end, and should therefore be reflected by a development perspective that unifies their functionality, not treating one merely as a consequence of the other. This approach can reduce both the hardware and software overhead that can result from a linear procedure that produces bridges or patches to connect segments. These cumbersome and confusing add-on features that join linear design solutions are removed and superseded when a unified technique that brings about conciseness and compactness to design is applied. A more consolidated SPU design should result in a less complex system control flow which can be formalized and compared to the precursor model. Also as a natural consequence of revising the 1997 SPU architecture, the 1998 SPU should not only correct 1997 SPU implementation errors but also enhance the overall performance characteristics. And while this augmentation focuses on improving the SPU's abilities, it should also address the issue of expandability. A good design for any system normally includes room for upgradeability to extend the product's useful lifetime thus guaranteeing its longevity. Finally, in order to preserve downward compatibility with the 1997 SPU, the 1998 SPU should provide the exact same data output format and respond to the same control commands as its predecessor (though the 1997 SPU has no provision for receiving instruction).

All of the aforementioned considerations should directly affect the 1998 SPU design process. By doing so, the 1998 Sensor Processing Unit endeavors to be a potent system that not only meets the original SPU objectives but is also both superior in its capabilities and effectiveness and more cohesive in design than its ancestor, the 1997 SPU. By performing hardware and software redesign, the 1998 SPU should ultimately:

- Reduce hardware and software overhead by trimming excessive design components,
- Simplify and formalize system control flow,
- Correct and enhance performance characteristics,
- Allow for future expandability, and
- Maintain backward compatibility with 1997 SPU.

5.3 System Proposal for the 1998 SPU

In order to realize both hardware and software redesign, an abstracted systems model should be formed to aid in steering the implementation decisions that satisfy the 1998 SPU objectives. But, prior to doing so, revisiting the 1997 SPU system layout might provide useful insight which should not be ignored. Jumping headlong into reengineering could possibly result in a design which repeats the errors of the 1997 SPU.

5.3.1 1997 SPU System

As shown in Chapter 3, the 1997 SPU system diagram depicts the control hierarchy between various devices and is reprinted in Figure 5.3.1-1.

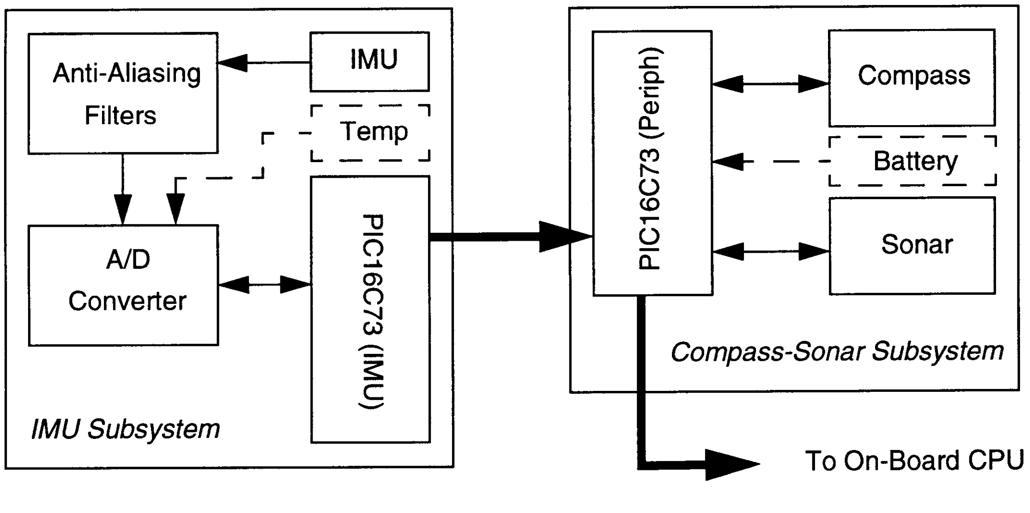


Figure 5.3.1-1: 1997 SPU System Diagram

This system diagram results directly from an analysis of the SPU hardware structure and clearly describes the control scheme of the 1997 implementation. Unfortunately, the design process did not establish this layout as a guide prior to the actual realization of the 1997 SPU; the schematic is merely a consequence of the hardware. This should be avoided in the 1998 development methodology by fabricating a generalized archetype for SPU systems. With such a model, all future modifications have a basis for comparing the evolution of the SPU architecture. The 1997 SPU's lack of such a model make refit and upgrade difficult since there is no reference for editing the composition. Lamentably, the 1997 SPU system schematic presented in Figure 5.3.1-1 is far too specific to act as a system abstraction. Changes in the 1997 SPU would be limited to the control hierarchy dictated by and limited to the major system components already in place.

5.3.2 1998 SPU System

In compliance with sound design methodology, the 1998 SPU development process begins by establishing a complete, albeit simple, blueprint for the system interactions of the SPU. In order to construct this map, all fundamental systems or components must be identified before a control network can be determined. Five components which immediately present themselves are the three sensors to be governed by the SPU, the inertial measurement unit, electronic compass, and sonar altimeter, and the two system states to be monitored such as flight battery voltage and IMU temperature. Each of these must be controlled or observed by a some device which has yet to be determined but, in the meantime, can be termed the "processor". Of course, it cannot be assumed that a direct link between the processor and sensors or battery will be satisfactory for directional, observational, or implementation purposes and consequently leads to the third and final component of the system layout, the interlink systems. These intermediary elements are responsible for preparing the sensor data or control signals for the processor as necessary and vice versa.

These five constituent parts or states of the SPU system design, sensors, battery level, IMU temperature, processor, and interlinks, comprise the full scheme for the SPU since the unit cannot be abstracted beyond these members until more information has been garnered. The 1997 SPU is able to specify its layout in more detail since it results from a post-analysis of the implemented system; no attempt was made at establishing a system skeleton, resulting in design choices with little foresight in their effects. Consequently, the 1998 SPU development process begins by laying a foundation for the project by depicting in Figure 5.3.2-1 a system interaction framework for determining the most appropriate hardware and software components and configurations.

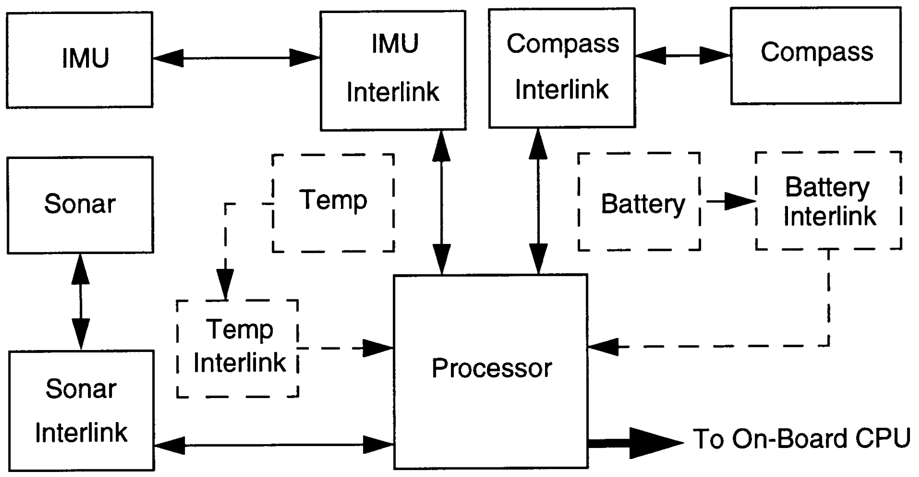


Figure 5.3.2-1: Generalized SPU System Diagram

5.4 Implementation Considerations

The Sensor Processing Unit is a complicated device both in terms of control and components. In developing such an apparatus, many factors arise which directly affect the formation of the instrument. As such, the hardware and software needs of the SPU should be identified in order to generate guidelines for laying out the next generation model. This allows for sober judgement in identifying both hardware and software components that effectively realize the generalized system model previously developed.

5.4.1 Hardware Considerations

5.4.1a Microprocessor vs. Microcontroller

Throughout much of this thesis, the terms "microprocessor" and "microcontroller" have been used interchangeably, but there is a definite distinction which must be explored in order to clarify the direction of the 1998 SPU revision. A microprocessor refers to an electronic device which is fundamentally computational in nature with its primary purpose being the evaluation of simple expressions. Using an internal archive of mathematical or logical operatives, the microprocessor manipulates mathematical or logical expressions to generate values for use by other electronic devices. On the other hand, the microcontroller goes a step further by including a library of utilities which have specific application to interaction with physical systems either natural or man-made. Nevertheless, while also able to manipulate values, the microcontroller does not typically carry the versatility in mathematical or logical evaluation that a microprocessor would have since its facilities are more focused to the control, monitoring, and interaction of systems. Yet, microcontrollers accomplish this without any of the additional hardware or software support that a microprocessor would need to achieve such capability. As a result, the microcontroller enjoys functionality that would have no place on a conventional microprocessor. For example, an analog-to-digital converter would be a natural part of a microcontroller's architecture, allowing for interaction between physical and digital systems, but would never be integrated into a microprocessor.

Applying this understanding to SPU design, a question arises as to whether a microprocessor or microcontroller is more suited to the task of sensor processing. This requires further insight which can be garnered by examining the intent of the SPU. The SPU's primary goal as a processing instrument is to distribute computational load away from the navigational computer. In this case, the word "computational" is something of a misnomer since the SPU is computational more for directional control than for mathematical manipulation ("directional control" refers to the polling and administration of sensor data and functions respectively). While data can be manipulated mathematically by the SPU, this does not precede the primary role of the SPU as a sensor integrating device which is more clearly suited to a microcontroller. The low-level abstraction of microprocessors would make them unsuitable as a viable alternative for direct interfacing with the navigational computer, let alone sensors, without significant additional hardware and software overhead. Consequently, a microcontroller is preferable over the microprocessor as the computational and directional engine for the SPU.

5.4.1b A/D Converter

The 1997 SPU utilizes a separate analog-to-digital converter in its design instead of the PIC16C73 on-chip A/D converter due to the microcontroller's insufficient resolution. Nevertheless, it is important to note that the PIC16C73 is indeed a microcontroller with its on-board A/D converter being a natural extension of its role as a device that interacts with physical systems. Consequently, the use of another converter in the 1997 SPU seems not only redundant, but also defeats the purpose of utilizing a microcontroller to reduce the overhead a microprocessor would require in accomplishing analog-to-digital conversion. Therefore, a microcontroller with an on-board A/D converter would prove more effective than employing an independent chip. As mentioned in Section 3.4.1, a minimum resolution of ten bits and eight channels are necessary for adequate precision in A/D conversion for all IMU channel, temperature, and battery voltage readings.

5.4.1c Anti-Aliasing Filters

Anti-aliasing filters on the inertial sensor outputs attempt to reduce aliasing errors due to analog-to-digital conversion. Unfortunately, they add to the hardware components necessary for proper operation. The only possible means of removing them would be to increase the sampling rate to twice that of the highest frequency component contained in the IMU output signals thus preventing aliasing. Once accomplished, digital filters could be applied to extract useful spectral data. However, while the IMU produces signals with a maximum frequency of approximately 1700 hertz, this is not the determining rate for the highest frequency in the IMU output signals. The lines carrying the IMU data are susceptible to electromagnetic interference from many sources including those with spectral content greater than a megahertz. Consequently, without an external filter, the sampling frequency for A/D conversion necessary to completely avert aliasing errors would be unreasonably high. Therefore, the anti-aliasing filters are necessary and justifiable components to the SPU design that must intercede between IMU outputs and A/D conversion inputs.

5.4.2 Software Considerations

As explained in Chapter 4, the 1997 SPU allows much of its software architecture to be dictated by hardware implementation. This places software design in a secondary role and thus belies its importance as an integral part of the entire SPU package. Studying the programming concerns of the SPU prior to implementation directly impacts the formation of the final product not only in software but also hardware. This notion is implicitly contradicted in the methodology followed in forging the 1997 SPU. Consequently, the 1998 SPU design process looks at SPU software considerations to assist in directing the actual drafting of the final device.

5.4.2a Software Architecture

Due to the various control and data interactions associated with the SPU, the software architecture can become rather cumbersome and complicated without a plan of action to aid its development. Consequently, software design objectives should be clearly defined to guide the actual structural implementation and coding. But, in order to do so, a technique for arriving at those objectives should be determined and utilized. Since the 1997 DSAAV attempts to follow distributed processing methodology, this is an appropriate place to begin to ultimately set the SPU code objectives.

As mentioned in Chapter 2, distributed processing concerns itself with the creation of systems formed of individual, self-sufficient real-time processes that cooperate in parallel with one another to accomplish a larger, more complex task. Applied to software, a distributed processing methodology involves the identification and formalization of processes. Such processes typically interact with and respond to events in real-time as opposed to being directed or called upon from within a linear sequence of software commands. The main advantage of process-based, event-driven software is that code is not limited to a predictable, looped structure dependent on a prespecified progression of commands. In such a linear framework, the software must have the appropriate values and inputs available at each step in the code. If these preconditions are not satisfied, then a program stall or crash could result thus yielding undesired and possibly chaotic results. But, a process and event based architecture does not stipulate any prerequisites for proper functioning. Instead, once a process is identified and associated with an event, the software need only be concerned with administering the appropriate response. Since the event itself is the triggering mechanism for the process handler, the immediate conditions surrounding the occurrence, such as available data, can be presumed to exist since the event is implicitly associated with the arrival of these conditions.

Since the SPU has many event associated processes, the unit becomes a prime target for the application of distributed processing methodology. For example, all of the sensors are sampled on a specific time schedule or rate, therefore the passage of a sampling period for any given sensor becomes an event to which the handling of that sensor can be associated. Also, since the sensors are not dependent on each other for proper functioning, their sampling can be run concurrently as independent processes. Furthermore, independent processes also facilitate the handling of multiple sensor transaction rates that would be more difficult to manage with purely linear programming. Consequently, the SPU software architecture seems ideally suited to using an multi-process, event-driven approach.

5.4.2b Software Characteristics

In addition to the design methodology, the SPU code (and most other forms of software) have general characteristics for efficacious coding namely efficiency, modularity, reusability, and robustness. Efficient software has several features including but not limited to usage of resources and speed. Program code that is efficient makes good use of all resources including data and program memory space. Such software has no redundant, useless, or dead code and minimizes the amount of storage space necessary for proper execution. Efficient programs also employ algorithms to encapsulate functions using as few commands as possible consequently affecting code speed, another trait for determining software effectiveness. Programs should not be so unwieldy as to be unable to manage its functions. For example, an SPU event process should run fast enough during its handling such that an unreasonable number of other events will not accumulate.

Though very similar, modularity and reusability are two other characteristics of good programming whose subtle connotations make them worthy of recognition as separate and distinctive characteristics. Modular code can be easily transported between applications requiring the software's functionality. But, reusable code, while also easily adapted for use within the same or different applications, is normally considered to be an elemental or fundamental construct. Modular code is not necessarily a foundational construct and can in fact be extremely complex and specific in nature. Consequently, reusable code may be viewed as a specific type or refinement of modular code.

Lastly, robustness alludes to a program's ability to manage, tolerate, and reject fault or unpredictable run-time conditions. Without an appreciable degree of robustness, programs in volatile, erratic, or unforeseeable environments can be subject to capricious behavior. While this is less often the case with linear code that operates with assumed prerequisite states, process-oriented code can be extremely susceptible to unforeseeable conditions that trigger events that are difficult to track and debug. Therefore, the 1998 SPU software should be especially conscious of robust design.

5.4.3 Hardware Component and Software Implementation Guidelines

As discussed, both hardware and software considerations directly affect the evolutionary process of the 1998 SPU. They suggest component selection or implementation guidelines for the next generation product and are summarized as follows:

Hardware Selection Guidelines:

- Microcontroller or microcontroller-based computational engine,
- Integrated or on-chip 10-bit, 8-channel A/D converter,
- Interceding anti-aliasing filters (between IMU and A/D converter),

Software Implementation Guidelines:

- Process-oriented, event-based architecture, and
- Efficient, modular, reusable, and robust code.

5.5 Hardware Design

Having achieved a generalized SPU system design, the parameters for hardware selection and software implementation may be applied to conclude what specific components will conform to and function effectively within the system framework as well as fulfill the 1998 SPU objectives.

5.5.1 Microcontroller Options

The 1998 SPU system diagram in Figure 5.3.2-1 has the "processor" as the computational and directional center for the SPU. From the discussion in Section 5.4.1a, the microcontroller clearly plays the role of "processor" in the 1998 SPU design. But, the wide variety of microcontrollers can make the final choice a difficult one. Nevertheless, a large search through the available microcontrollers revealed three options that seemed most promising for the application at hand: a PC-104 compliant microcontroller, an enhanced microcontroller, and a PIC17C756 microcontroller.

5.5.1a PC-104 Compliant Microcontroller

PC-104 is a standardized printed circuit format for small, compact computer stacks. CPUs that conform to this type of layout, though typically not as computationally powerful, are capable of all the normal functionality of conventional desktop machines including video output, serial communications, and ethernet hookup. The navigational computer for the DSAAV is PC-104 compliant and executes algorithms for flight control and directs much of interaction between systems. An interesting feature of PC-104 computers is that additional hardware can be added by simply stacking them on the device bus lines. As long as these add-ons comply with the PC-104 form factor and bus communication protocols, they can be easily integrated for control by the stack CPU.

A PC-104 compliant microcontroller is basically a microcontroller integrated circuit with additional external features merged together on a PC-104 printed circuit board. For instance, a PC-104 microcontroller running at a clock speed of 20 megahertz is relatively fast and can be programmed in C for supposedly handling SPU functions independently of the computer stack's main processor. Since this device is directly connected to the navigational computer bus lines, the on-board CPU can quickly and easily access data from the PC-104 microcontroller. Looking at the 1998 system diagram in Figure 5.3.2-1, this clearly facilitates and quite possibly removes the communication overhead between the SPU processor and the on-board CPU thus allowing for faster data access.

Unfortunately, the PC-104 microcontroller can have several limitations which override its potential as an SPU processor. For example, some models bound the total number of stackable circuit boards to two. Since the DSAAV on-board computer can have up to four circuit cards not including the microcontroller itself (i.e., CPU, memory, ethernet, and serial ports), this clearly precludes the microcontroller from being used as part of the 1998 SPU. Furthermore, even without the stack limit, it is not clear that a PC-104 microcontroller can coexist with a true microprocessor on the same bus line. And while the navigational algorithms and flight control command could possibly be ported to the microcontroller, it would be completely unsuited for evaluating the complex mathematical equations necessary for autonomous flight. Normally, this duty would be assigned to a full-function microprocessor that is more adapted for complex mathematical and logical evaluation. Consequently, the PC-104 microcontroller is not a viable option for the SPU processor.

5.5.1b Enhanced Microcontroller

An enhanced microcontroller is very similar to the PC-104 compliant microcontroller and can in fact be viewed as a generalization of the PC-104 device. Built around a RISC-based microprocessor and a microcontroller coprocessor, the enhanced version incorporates additional features which increase its usability especially as a mathematical engine. Also, since the device lacks a PC-104 bus, it is capable of stand-alone operation and implicitly has an ability to communicate with other devices. With serial ports, built in analog-to-digital conversion, C-programmability, and digital I/O, the enhanced microcontroller seems well suited to acting as the SPU processor.

Regrettably though, the enhanced microcontroller is an extremely proprietary device which includes much internal overhead in both hardware and software that might ultimately hinder the SPU (this is unlike the PC-104 microcontrollers that conform to a standard architecture that is proven and accepted). For instance, the enhanced microcontroller architecture is highly dependent on other components which ultimately reduces its mechanical robustness since added interconnects and parts increase the possibility of failure caused by the hostile operating environment produced by the DSAAV. Furthermore, as in the case of the 1997 SPU, extra hardware and software overhead adds complexity to the generalized system and devours possible resources. This should be avoided since it can ultimately foil the simplicity of design and make future modification or expansion difficult.

5.5.1c PIC17C756 Microcontroller

The PIC17C756 is an integrated circuit designed to be a microcontroller. Developed and manufactured by Microchip Technology, Inc., this device represents the next-generation superseding the PIC16C7X series. Equipped with a 12-channel, 10-bit on-chip analog-to-digital converter and 50 digital I/O lines, the PIC17C756 microcontroller easily satisfies the hardware selection guidelines. Furthermore, with its multiple interrupt system, various types of events, both internal and external, can be responded to allowing for a process-oriented software architecture thus partially satisfying the software implementation considerations. Nevertheless, these hardware and software features do not alone stand as convincing arguments for choosing the PIC17C756 over the enhanced microcontroller. But, the additional criterion that ultimately reject the enhanced microcontroller instead support the nomination of the PIC17C756.

The PIC17C756 is a single integrated circuit completely embedded within a ceramic leadless chip carrier. Therefore, unlike the enhanced microcontroller, the PIC microcontroller is a physically compact product that is also highly immune to vibrational damage since there are no attached parts. Furthermore, not only is hardware overhead reduced, but due to the simple, minimalistic assembly-like coding language, PIC operations carry no compiling or interpretation overhead. Instead, each operator is a fundamental part of the PIC17C756 function library and cannot be further abstracted. On the other hand, the enhanced microcontroller utilizes higher level languages such as C or BASIC whose functions are formed from a series of assembly-like operators. These functions compose a library of routines that constitute the basis for higher level languages. But, since underlying each function is assembly code, the enhanced microcontroller must utilize an interpreter or compiler to deconstruct the routines into their fundamental operations. Consequently, the efficiency of the code is highly dependent on the interpreter or compiler and cannot be predicted, compensated, or controlled as readily as PIC software that only contains elementary operators. Of course, the efficiency as well as modularity, reusability, and robustness of PIC software is therefore the programmer's burden.

Finally, since the PIC17C756 is a completely self-contained microprocessor capable of fulfilling all SPU hardware and software considerations as previously specified, it can act as the processor in the generalized SPU system architecture. As an added bonus, the use of a single microprocessor delinks the software architecture from the hardware implementation since the software can be completely contained within the device and not divided between multiple microprocessors as in the 1997 SPU. Consequently, in looking for a product that not only meets the capability requirements of the SPU and its revision goals, but also works well in the abstracted system architecture previously developed, the PIC17C756 stands out as an excellent selection.

5.5.2 The PIC17C756 and the SPU

5.5.2a Overview of PIC17C756 Capabilities

The PIC17C756 provides numerous microcontroller core and peripheral features which make it well suited for immediate application to the 1998 SPU and future enhancements. Table 5.5.2a-1 itemizes these characteristics and capabilities.

5.5.2b Comparison of PIC17C756 to 1997 SPU

As can be seen by its features themselves, the PIC17C756's functionality quite easily outmatches the 1997 SPU's capabilities. While this performance gain results in part from a faster instruction rate, a quicker clock frequency in tandem with new components and a more streamlined system architecture make for an even more potent and powerful 1998 SPU with a greater potential for future enhancement.

Microcontroller Core Features:	Peripheral Features:
• 33 Mhz Clock Input • 121 ns Instruction Cycle • 58 Single Word Instructions • All Single Cycle Instructions Except Program Branches and Table Read/Writes (2 Cycles) • Hardware Multiplier • Interrupt Capability • 16 Level Deep Hardware Stack • Direct, Indirect, and Relative Addressing Modes • Internal/External Program Memory Execution • Capable of Addressing 64K x 16 Program Memory Space • 902 x 8 Bytes Data RAM	• 50 Bidirectional I/O Pins • High Current Sink/Source for direct LED drive • Four 16-bit Capture Inputs • Three 10-bit Max PWM Outputs • TMR0: 16-bit Timer/Counter with 8-bit Programmable Prescaler • TMR1: 8-bit Timer/Counter • TMR2: 8-bit Timer/Counter • TMR3: 16-bit Timer/Counter • Two USARTs • 10-bit, 12 Channel A/D Converter • Synchronous Serial Port and I^2C Modes

Table 5.5.2a-1: PIC17C756 Microcontroller and Peripheral Features [5]

For instance, the PIC17C756 includes a new set of mathematical instructions which take advantage of its "hardware multiplier". This on-chip device is basically a math coprocessor that is specifically designed to decrease the number of instruction cycles per multiplication thus increasing code speed. This new feature opens the door to more efficient and powerful digital signal processing techniques not attainable with the older PIC16C73 microprocessor. Additionally, an increased number of independent hardware interrupts allows for greater flexibility and variety in developing interrupt-triggered event handlers. Unlike the PIC16C73 which has only one interrupt and therefore cannot implicitly distinguish between event types except via software, the PIC17C756's multiple interrupts can be associated to different triggering mechanisms. This simplifies code structure by localizing process handlers with a specific interrupt. This is an obvious boon to an process-oriented, event-based software architecture which the 1998 SPU embraces. Furthermore, using a single microcontroller eliminates the split-processor 1997 SPU architecture and reduces both hardware and software communications overhead thus increasing software execution and allowing faster sampling and transmission rates.

Finally, it should be noted that the PIC17C756 coding language is very similar to that of the PIC16C73 since both microcontrollers are manufactured by Microchip Technology, Inc. This has both an advantage and disadvantage. The assembly-like instruction set of the PIC-type microcontrollers can be very difficult to follow. Especially in light of higher level languages which have greater readability, PIC code is rather cryptic. Nevertheless, because the coding language is native to the 1997 SPU, this reduces the learning curve for the 1998 SPU implementation. Furthermore, some of the code in the 1997 SPU can be ported to the 1998 SPU thus taking advantage of reusability.

5.5.3 Design and Implementation

The circuit design for the 1998 SPU can be guided in part by the generalized system architecture developed earlier. Since the compass and sonar both utilize digital control and data lines and do not require any prefiltering, their interlink systems are simply wires connecting the sensors to the appropriate digital I/O ports on the PIC17C756. Similarly, being a passive source of relatively static data, the battery and IMU temperature detector voltages can be connected directly to the microcontroller A/D converter. On the other hand, with its high frequency analog signals, the IMU output signals must ultimately be guided to the PIC17C756 A/D converter inputs through anti-aliasing op-amp filters. Consequently, these filters form the IMU interlink system. In the 1998 SPU system architecture shown in Figure 5.5.3-1, the two-way arrows between the compass, sonar, and microcontroller indicate data flow to and command flow from the PIC17C756 respectively. In contrast, battery level, temperature, and inertial signals travel in only one direction, either directly to or through the anti-aliasing filters toward the microcontroller, since they all continuously dispense information without the need for control signals.

Figure 5.5.3-2 illustrates the circuit implementation for the 1998 SPU based the system architecture just introduced. Several features in the 1998 SPU schematic are worthy of mention but three requiring special attention are the linear acceleration IMU channels, the anti-aliasing filters, and the battery voltage scaler. All linear IMU outputs are tied to ground with a 450 ohm resistor as in the 1997 SPU to convert current into voltage. While this resistor can alter the ~50 hertz filter poles, the shift in breakpoint frequency would be slight and therefore negligible. Also, while the anti-aliasing filters are passive, the op-amp buffers are necessary to ensure low input impedance to the PIC17C756 A/D converter (maximum analog source impedance of 10kohms). Again, as in the 1997 SPU, the 32k ohm resistors acting as the negative feedback path for the op-amp buffers compensate for bias current in the amplifiers but may be removed if compensated for in software. Finally, the battery voltage scaler dimensions the battery voltage range to that of the A/D converter. The 1998 SPU achieves a negative 0.25 scale factor using an inverting amplifier rather than a voltage divider as in the 1997 SPU. This ensures a low input impedance as required by the PIC17C756 A/D converter. Also, as a word of caution, in order to maintain downward compatibility with the 1997 SPU data format, the scaled battery voltage must be multiplied by a factor of -1 in order to remove the inverting effect caused by the amplifier.

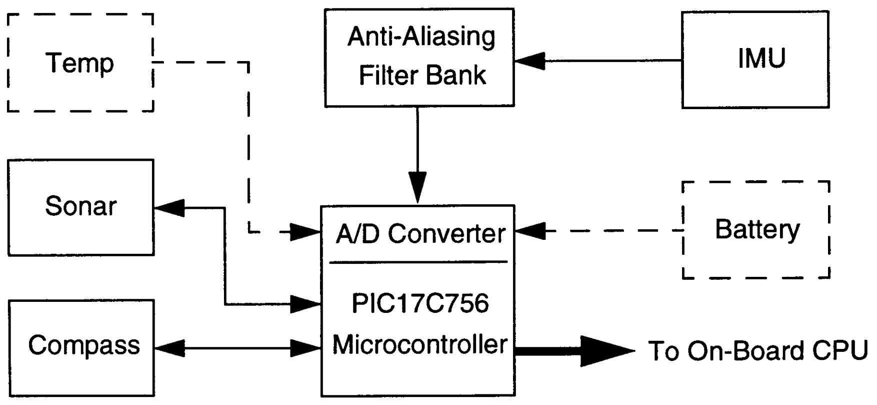
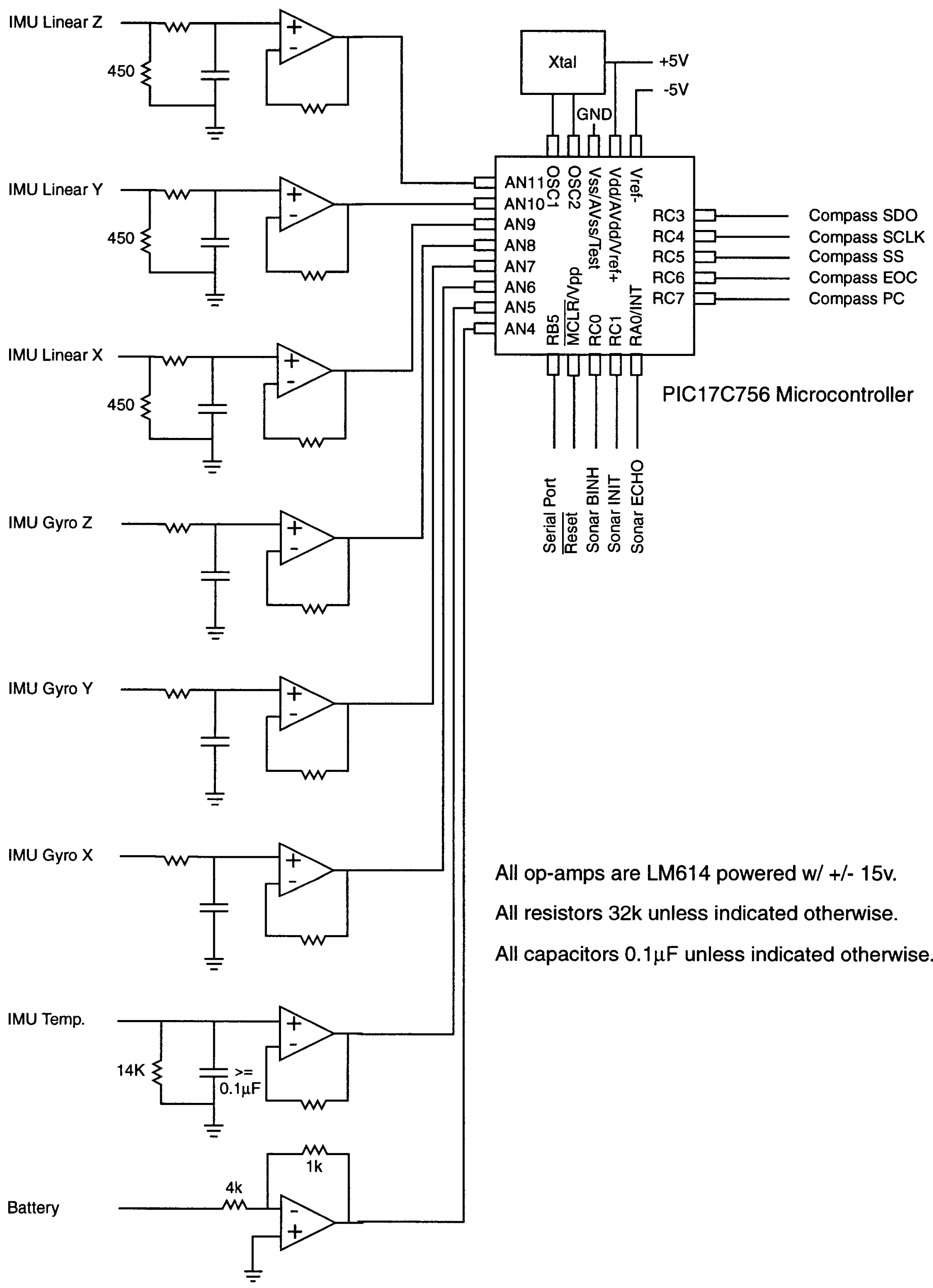


Figure 5.5.3-1: 1998 SPU System Architecture



All op-amps are LM614 powered w/ +/- 15v.

All resistors 32k unless indicated otherwise.

All capacitors 0.1μF unless indicated otherwise.

Figure 5.5.3-2: 1998 SPU Proposed Circuit Schematic

5.6 Software Design

5.6.1 Sensor Protocols

Each sensor controlled by the 1998 SPU is an electronic device with specific control and communication protocols that must be adhered to in order to guarantee proper operation. These procedural rules for interacting with the sensors are encoded in the SPU software and enable the handshaking and coordinating necessary to properly extract and store sensory data. Consequently, each of the three SPU sensors should be briefly reviewed.

5.6.1a Flux Magnetometer (Electronic Compass)

Produced by Precision Navigation, Inc., the electronic compass uses two orthogonal coils to detect the earth's magnetic field and determine its heading. While having a relatively slow update rate of 5 hertz, the compass has a 2 degree accuracy under reasonable measurement conditions i.e., proper calibration, no ferrous materials nearby, approximately level with respect to earth's surface.

Correct sampling of the electronic compass necessitates conforming to a rather rigid timing schedule for transmission and reception of both commands and data. The electronic compass has five digital I/O lines used for control and data reception namely PC, EOC, $\overline{SS}$, C-CLK, and C-SDO. Figure 5.6.1a-1 exhibits a timing schedule for these I/O lines.

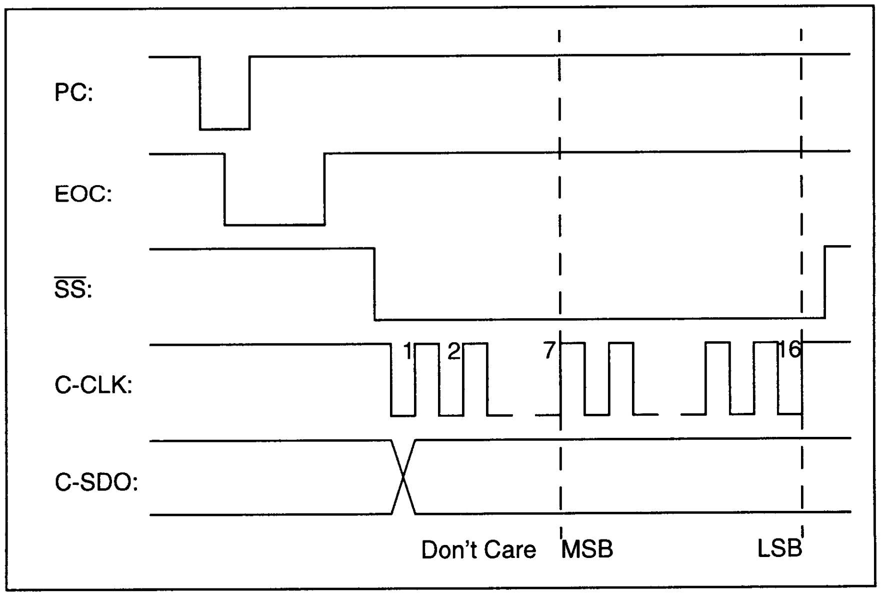


Figure 5.6.1a-1: Compass Signal Timing Schedule [6]

To summarize a single operation cycle, the PC begins a compass sampling cycle by being set low. While in this state, the EOC, or End Of Conversion, line should follow suit and go low as a result of the compass' internal functions. The EOC will remain unchanged until new data is available, at which point EOC reverts to high. Once this occurs, $\overline{SS}$ can be brought low and data can be transferred by clocking the C-CLK line low and high repeatedly 16 times. With each cycle, data is sent from the compass to the controller on C-SDO with the first 7 values being non-significant. The following 9 bits represent the compass heading from most to least significant. After all 16 its have been received, $\overline{SS}$ can be reasserted and the compass sampling cycle can be restarted.

5.6.1b Sonar Altimeter

The sonar altimeter is a device that transmits ultrasonic pulses and senses their echoes to determine height. This sensor's usage is somewhat similar to the compass in that its operation cycle calls for the repeated issuance of commands on a specific time schedule in order to attain the desired information as Figure 5.6.1b-1 shows.

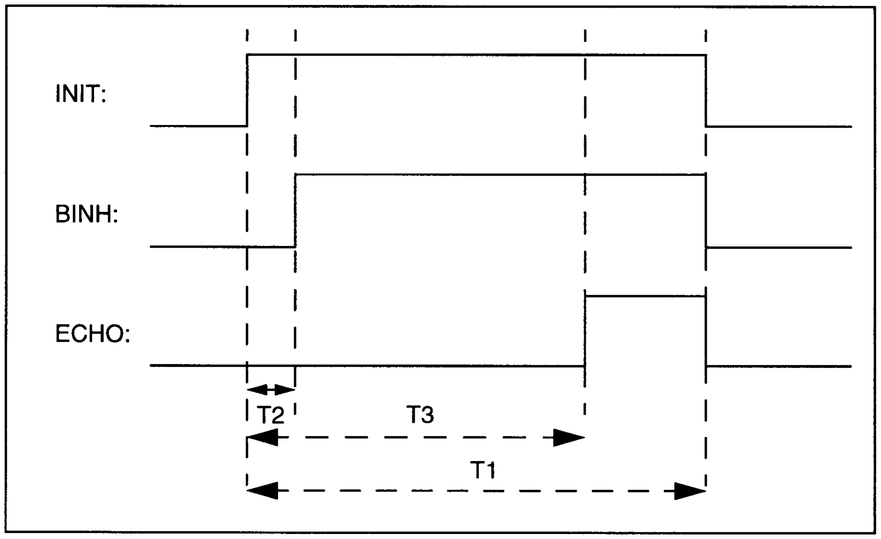


Figure 5.6.1b-1: Sonar Signal Timing Schedule [7]

Like the compass, the sonar altimeter is sampled repeatedly, where T1 is the period length. Each period of sonar operation begins with an initiation of an ultrasonic pulse, or ping, by asserting the INIT line. Once this pulse is transmitted, the sonar awaits the arrival of an echo which is indicated by a low-to-high transition on the ECHO signal line. But, in the time span immediately following the INIT initiated ping, transient signals on the ultrasonic transducer can cause false ECHO triggers. Consequently, all possible echo signals are suppressed for an initial period of length T2 by holding the blanking inhibit, or BINH, line low. Once BINH goes high, echoes can be received. The time between starting the ping and receiving an echo, T3, determines the altitude of the aircraft.

5.6.1c Inertial Measuring Unit

Compared to the other sensors, sampling the IMU is rather trivial. Since the IMU requires no command signals to initiate data transfer, the SPU must simply poll the device at a rate which would allow the complete data frequency spectrum to be obtained i.e., twice the rate of the maximum IMU frequency content. Since anti-aliasing filters remove data above 50 hertz, the IMU must be polled at a minimum of 100 hertz. But, due to the imperfect nature of filters, some degree of high frequency aliasing errors result albeit small. Consequently, a higher IMU sampling rate behooves the inertial reading accuracy.

5.6.1d Optimal Sensor Sampling Rates

All navigational and internal status information must be acquired by sampling the various SPU components for data. But, this polling cannot simply happen as a matter of fact. Instead, optimal rates for sampling must be found in order to effectively gather sensor data. For example, essential navigational sensors use an optimal sampling rate that removes aliasing errors i.e., twice the maximum sensor signal frequency content. The IMU channels are prefiltered to attenuate the frequency band above 50 hertz. As a result, a minimum 100 hertz sampling rate would be necessary to prevent sampling errors due to aliasing. Nevertheless, since anti-aliasing filters on the IMU lines cannot completely remove spectral content above 50 hertz, there are some residuals of the higher frequencies that pass. Consequently, since increased sample rates reduce aliasing effects, the optimal sampling frequency of the IMU channels is the highest achievable above the minimum 100 hertz without overburdening the microcontroller. Unfortunately, it is difficult to determine what this ideal IMU sample rate is without a test bed to verify, but a good first guess is 100 hertz for both linear and angular acceleration channels. At this rate, the Nyquist criterion is satisfied thus allowing the frequency bands of the IMU channels below 50 hertz to be captured without aliasing. But, this rate is not so high as to be excessively taxing to the microcontroller and in fact minimizes the amount of time the PIC17C756 spends sampling IMU channels, freeing it to manage its various other tasks.

As for the compass and sonar altimeter, the anti-aliasing consideration does not apply at all. Unlike the IMU channels that continuously provide data, the other two sensors provide data only on command. Consequently, their optimal data rate is the maximum frequency at which the sensors can be polled. For the compass, this rate is 4 hertz, and for the sonar-altimeter, it is approximately 16.7 hertz. Finally, as for the internal SPU status readings, the battery voltage level and IMU temperature have long enough time constants such that their sampling can be relatively slow, up to a limit of approximately one update per second. All of the optimal sensor and status sampling rates for the 1998 SPU are summarized in Table 5.6.1d-1.

1998 SPU Optimal Sensor and Status Sampling Rates	
IMU Acceleration Channels:	100 hertz
Electronic Compass:	4 hertz
Sonar Altimeter:	16.7 hertz
Battery Voltage Level:	<= 1 hertz
IMU Temperature:	<= 1 hertz

Table 5.6.1d-1: 1998 SPU Optimal Sensor and Status Sampling Rates

5.6.2 Software Event Categorizations

In examining the software architecture for the 1998 SPU, a process-oriented, event-driven approach has been established to be an ideal methodology for laying out code structure. With this in mind, the actual SPU program implementation can begin by identifying SPU-associated events and proposing a process handler for those occurrences.

5.6.2a Period-Based Events

A period-based event is associated with the moment of a periodic phenomenon's occurrence. For the most part, such events are predictable to the extent that they happen on a regular, consistent basis, but are nevertheless unpredictable in terms of their circumstantial causes or conditions. In the case of the SPU, the sensors, temperature, and battery sampling fall into the category of period-based events since each of the three sensors and two system states are polled at their most optimal sampling rates as previously determined.

With the polling frequencies of all three sensors and two SPU states in mind, a basis exists for developing software handlers that respond to the elapse of the various sampling periods. In the case of the 1998 SPU, the software sensor handlers could be called in response to a periodically invoked timer interrupt that detects the arrival of a given sensor sampling period. Therefore, this timer interrupt would need to keep track of sensor or status period stopwatches. At each invocation, the timer interrupt with an assumed period of T would increment all stopwatches by T and check to see if any sensor or status has reached the end of its polling cycle. If so, an appropriate handler would be called to begin the next sample for that device and its associated stopwatch would be reset. This approach works well for reading the IMU acceleration channels as well as the battery voltage and IMU temperature since they must simply be read at each sample period. But, the compass and sonar altimeter cannot be sampled for data as easily. In order to receive information, they must adhere to the specific command signal protocols previously described.

In the case of the compass, all of its control protocol is bound to a strict timing schedule. Consequently, each phase of the compass' command sequence can be invoked periodically by the timer interrupt as already explained. Of course, every control signal occurs cyclically with the same period as the overall compass cycle, but each command is issued with a unique phase shift relative to the beginning of the compass period in accordance with the timing schedule portrayed in Figure 5.6.1a-1.

The sonar altimeter also has a command sequence which it must follow in order to function properly. Since the sonar control signals follow a tight timing schedule just as those of the compass, the same handling methodology can be applied to both devices. But, while the sonar's control signals are executed periodically, its data signal, or echo, has an arrival time that is a function of the sensor's distance from the ground and is therefore inherently unpredictable. This requires another form of event handling involving occurrence interrupts.

5.6.2b Occurrence-Based Events

Occurrence-based events are associated with the happening or arrival of prespecified conditions, information, or time. Unlike period-based events, the moment at which these incidents appear cannot be predetermined, but their arrival is always associated with a specific occurrence. For example, the SPU's sonar altimeter echo response is an occurrence-based event that is predicated by the detection of a bounced ultrasonic ping on the sonar transducer. The SPU can be equipped to handle such a happening by defining an interrupt that is activated by a low-to-high transition on the sonar altimeter's ECHO line, indicating the presence of a bounced ping. This echo interrupt can then proceed to administer the event by calculating the time between the ping transmission and its bounced return. Using this elapsed time, the distance between the sonar altimeter and the bouncing object (i.e., ground) can be calculated.

5.6.3 Software Implementation Concerns

5.6.3a Communication with the Navigational Computer

The SPU is designed to package and transmit sensor data to the navigational computer via serial lines. In order to form a data package, the SPU must take the most recently acquired sensory information and copy it into a transmission buffer for sending to the on-board computer. These packets should be sent at a rate which allows all data from the fastest sampled sensor (i.e., 100 hertz IMU data) to be transmitted. Of course, the Nyquist criterion can be invoked to set the minimum transmission rate at twice that of the maximum data sampling frequency. But, if the transmission frequency is set to be the same rate as the fastest sensor sample rate and the two are in phase (i.e., data is gathered and sent immediately), then the SPU data packet transmissions can occur at the fastest sampled sensor rate as opposed to twice that value, without losing information. But, these rates must be identical and in phase to achieve the communications overhead savings.

But, while the Nyquist rate does not apply to the fastest sampled sensor as long as its sample frequency is equal and in phase with the transmission rate, it does apply to the sensors polled more slowly. If the fastest sampled sensor is used as the transmission rate, this rate must be at least twice the frequency of the sensor with the second fastest sample rate in order to ensure that all of the slower sensors' data is captured. If the sensor with the second fastest sampling rate is being polled at its Nyquist rate, then all sensors sampled more slowly are implicitly being prompted for data at better than their Nyquist frequencies. Consequently, if the fastest sensor sampling rate is at least twice that of any other sensor, then the transmission rate is equivalent. Otherwise, the transmission rate must be twice that of the fastest sampled sensor to ensure that all of the slower sampled sensors' have their Nyquist rates satisfied. In the case of the 1998 SPU, since the IMU channels are the fastest sampled at 100 hertz and all other sensors are sampled at well below half that rate (i.e., compass at 4 hertz, sonar altimeter at 16.7 hertz), packet transmission can also occur at 100 hertz thus saving the transmission overhead from a 200 hertz Nyquist data rate.

As an aside, since the packet data is sent at regular intervals, transmissions can be seen as a periodic events as described in the previous passages. Consequently, the sending of packets can also be seen as an process that can be executed via a period-based interrupt handler.

5.6.3b Interrupt Handlers versus Linear Code

As discussed, interrupts are well suited to the handling of period- and occurrence-based events. Unlike linear code, interrupts are called and executed when prespecified conditions are met. Once the requisite circumstances occur, the appropriate interrupt is immediately invoked regardless of the current location of the program during run-time. Once its duties are complete, the interrupt proceeds to return control to the part of the program into which it intervened and the software continues as normal. But, now that the 1998 SPU periodic and occurrence events have been identified and associated with interrupt handlers, the linear code content must be determined. But, in the case of the 1998 SPU, all functions except microprocessor initializations can be described as events with associated interrupts. So, beyond the necessary preparatory routines for the PIC17C756, no linear code should be necessary!

Without linear code, the 1998 SPU code can be entirely described as various interrupt handlers that manage independent event-invoked processes. This is completely in line with distributed processing methodology which indicates that independent systems should be modular, autonomous, and cooperative. Clearly, since the individual processes can be exported to other applications simply by transplanting the interrupt handler as a complete package, they demonstrate modularity. Further, by being implemented as individual processes, each sensor or status event handler can be generalized to an autonomous system. And finally, by virtue of working together to generate SPU packet data, the interrupts are cooperative. Consequently, the almost complete usage of interrupt handlers banishes the need to understand code flow as in linear programs and only requires the programmer to know which handler is associated with what event. This ultimately generates a simpler, more efficient, and robust software architecture whose control scheme can be easily depicted.

5.6.4 Software Control Schemes and Schematics

5.6.4a Overall Implementation

Having identified all the possible processes that can be handled via interrupts, namely the sampling of the IMU, compass, sonar altimeter, battery and IMU temperature and the transmission of data packets to the navigational computer, the 1998 SPU software can be realized. Since all these processes except the altimeter sampling are completely period-based events, they can be assigned to one of the four separate timers and associated interrupts.

Since IMU, battery, and temperature sampling all involve A/D conversion, their handling can be combined under a single conversion-time interrupt. Of course, the highest sampling rate of 100 hertz for the IMU will be the frequency at which this conversion-time interrupt is invoked and should not be a hindrance for the battery and IMU temperature since they will only be harmlessly oversampled. Once started, the interrupt handler begins the first A/D conversion and exits. After this first conversion is completed, a "conversion done" interrupt occurs (an inherent PIC17C756 hardware feature) and can be used to trigger the next A/D conversion. This conversion cycle repeats itself until all conversions have been implemented and does not begin anew until the next conversion-time interrupt is conducted.

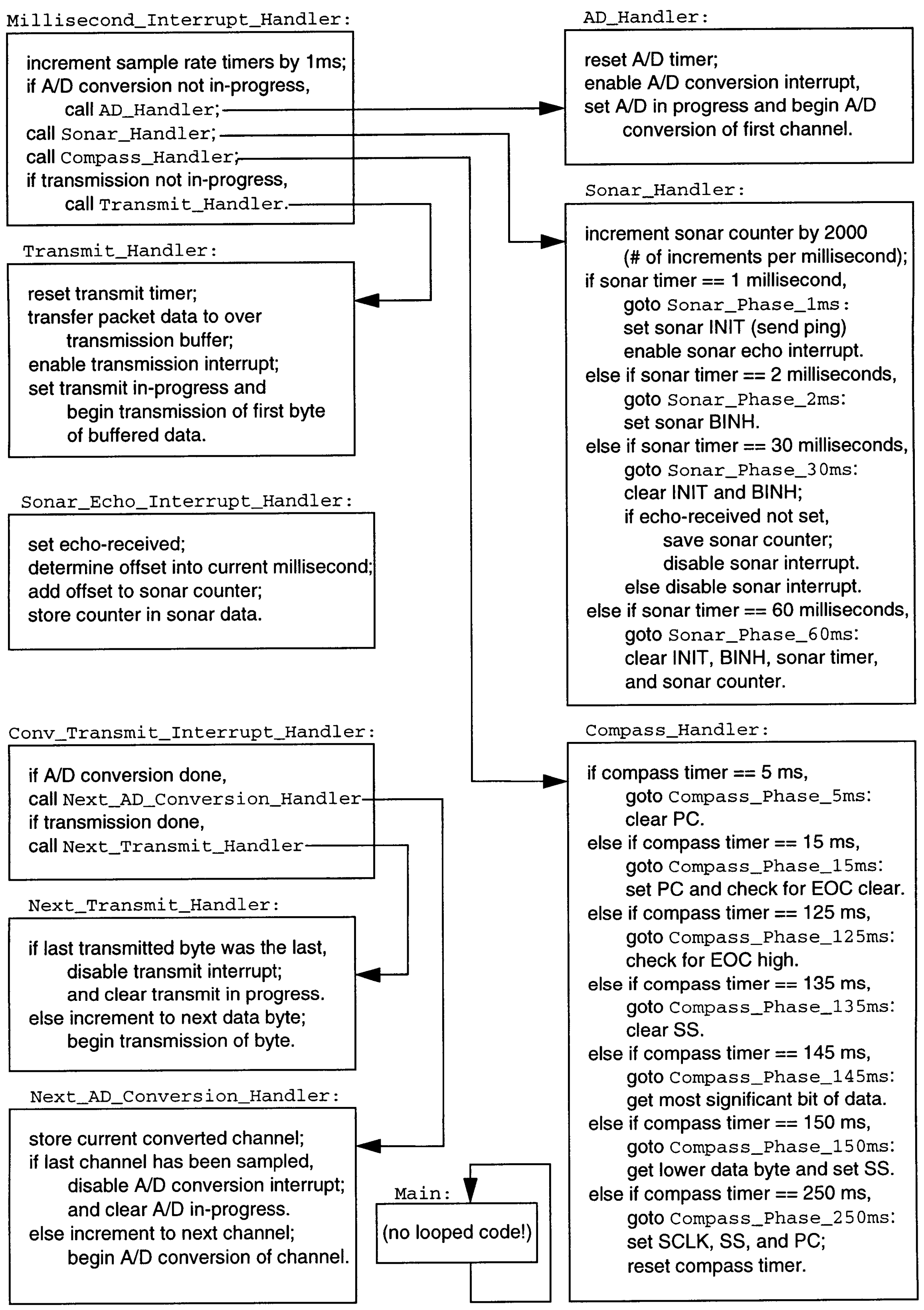
The compass control sequence is based entirely on a time schedule and can therefore be handled totally by a timer interrupt. Since the optimal sampling rate for the compass is 4 hertz, the compass timer interrupt begins a fresh sampling every 250 milliseconds. But, note that the compass timer interrupt is always being called at a rate greater than 4 hertz. This allows the interrupt handler to check for and execute intermediary command phases if one is pending. If no control job is waiting, the interrupt exits without effect. In any case, once a compass sampling cycle is begun, the compass interrupt executes its first command phase to initiate a reading and exits. Subsequent compass timer interrupts determine if the next command phase time has been reached. If so, the appropriate command signal is applied or data is captured by the handler and again the interrupt exits. Each command occurs according to the control schedule described in Section 5.6.1a until the last data bit has been recovered. At that point, the compass timer interrupt has executed all its commands and does not restart until another compass period is reached.

As discussed before, the sonar works in a manner very similar to the compass except that it also must incorporate a second type of interrupt - an occurrence-based interrupt. The sonar uses a periodic timer interrupt in the same way as the compass to implement its command sequence according to the timing schedule specified in Section 5.6.1.b. But, the sonar also utilizes the occurrence interrupt. This interrupt is generated by an echo signal sent by the sonar altimeter and received by the PIC17C756 on its external interrupt pin (RA0/INT). Once started, the echo interrupt handler determines the exact arrival time with respect to the ping transmission time in order to calculate altitude. Since the sonar sampling rate is 16.7 hertz, each sonar cycle of 60 milliseconds must pass before another period can begin.

Figure 5.6.4-1 depicts the interrupts and their handlers that form the software control schematic for the 1998 SPU and Appendices A and B contain the actual code.

5.6.4b Technical Details

One aspect of the 1998 SPU software implementation that needs further explanation is the choice to establish a base sensor interrupt timer resolution of 1 millisecond, similar to the resolution of the 1997 SPU peripheral subsystem. The 1997 SPU uses a 1 millisecond interrupt base to keep track of elapsed time. This was a somewhat arbitrary decision but provided a high enough resolution to satisfy the timing needs of sensor sampling (which occurs on a tens of milliseconds rate) but low enough to prevent excessive timer interrupt overhead. To a certain extent, the decision to use a 1 millisecond resolution on the 1998 SPU comes out of a desire to satisfy those same needs. But, the resolution of the sensor interrupt timers for the 1998 SPU has definitive affects on sensor accuracy that should be quantified.



```
Millisecond_Interrupt_Handler:
    increment sample rate timers by 1ms;
    if A/D conversion not in-progress,
        call AD_Handler;
    call Sonar_Handler;
    call Compass_Handler;
    if transmission not in-progress,
        call Transmit_Handler.
```

```
Transmit_Handler:
    reset transmit timer;
    transfer packet data to over
        transmission buffer;
    enable transmission interrupt;
    set transmit in-progress and
        begin transmission of first byte
        of buffered data.
```

```
Sonar_Echo_Interrupt_Handler:
    set echo-received;
    determine offset into current millisecond;
    add offset to sonar counter;
    store counter in sonar data.
```

```
Conv_Transmit_Interrupt_Handler:
    if A/D conversion done,
    call Next_AD_Conversion_Handler
    if transmission done,
    call Next_Transmit_Handler
```

```
Next_Transmit_Handler:
    if last transmitted byte was the last,
        disable transmit interrupt;
        and clear transmit in progress.
    else increment to next data byte;
        begin transmission of byte.
```

```
Next_AD_Conversion_Handler:
    store current converted channel;
    if last channel has been sampled,
        disable A/D conversion interrupt;
        and clear A/D in-progress.
    else increment to next channel;
        begin A/D conversion of channel.
```

```
Main:
    (no looped code!)
```

```
AD_Handler:
    reset A/D timer;
    enable A/D conversion interrupt,
    set A/D in progress and begin A/D
        conversion of first channel.
```

```
Sonar_Handler:
    increment sonar counter by 2000
        (# of increments per millisecond);
    if sonar timer == 1 millisecond,
        goto Sonar_Phase_1ms:
        set sonar INIT (send ping)
        enable sonar echo interrupt.
    else if sonar timer == 2 milliseconds,
        goto Sonar_Phase_2ms:
        set sonar BINH.
    else if sonar timer == 30 milliseconds,
        goto Sonar_Phase_30ms:
        clear INIT and BINH;
        if echo-received not set,
            save sonar counter;
            disable sonar interrupt.
        else disable sonar interrupt.
    else if sonar timer == 60 milliseconds,
        goto Sonar_Phase_60ms:
        clear INIT, BINH, sonar timer,
        and sonar counter.
```

```
Compass_Handler:
    if compass timer == 5 ms,
        goto Compass_Phase_5ms:
        clear PC.
    else if compass timer == 15 ms,
        goto Compass_Phase_15ms:
        set PC and check for EOC clear.
    else if compass timer == 125 ms,
        goto Compass_Phase_125ms:
        check for EOC high.
    else if compass timer == 135 ms,
        goto Compass_Phase_135ms:
        clear SS.
    else if compass timer == 145 ms,
        goto Compass_Phase_145ms:
        get most significant bit of data.
    else if compass timer == 150 ms,
        goto Compass_Phase_150ms:
        get lower data byte and set SS.
    else if compass timer == 250 ms,
        goto Compass_Phase_250ms:
        set SCLK, SS, and PC;
        reset compass timer.
```

Figure 5.6.4-1: 1998 SPU Software Control Diagram

The oscillator rate that drives the PIC17C756 microcontroller is 32 megahertz. The microcontroller scales this frequency down by a factor of four and again by a prescale factor of 4 to produce an instruction rate of 2 megahertz. For both the PIC16C73 and PIC17C756, each instruction cycle increments the internal timer registers thus producing an update rate of 2 megahertz. These timer registers differ from the sensor interrupt timers. The timer registers increment until their values match a comparison amount established in code. Once this occurs, the timer registers generate an interrupt and roll over to start accumulating instruction cycles all over again. The interrupt rate caused by the overflow of the timer registers is used as the basis for incrementing the sensor interrupt timers of the SPU.

By choosing a 1 millisecond sensor timer interrupt resolution, their corresponding timer registers must have a comparison value of

$$2,000,000\left(\frac{instructions}{\mathrm{sec}ond}\right)\times 1(milli\mathrm{sec}ond) = 2,000(instructions)$$

Consequently, since the timer registers generate an interrupt every 2000 instruction cycles, the sensor timer interrupts occur every 1 milliseconds. This has significance for the sonar because its echo arrival time must be determined as accurately as possible in order to produce good altitude predictions. Since the speed of sound is 330 meters/second, every millisecond corresponds to 0.33 meters, or 33 centimeters, of sound travel. Consequently, if the sonar arrival time is determined with a finite resolution of 1 millisecond as given by the sensor timer interrupt rate, this arrival time could contain a time error of +/- 1 millisecond corresponding to +/- 33 centimeters of altitude error which is unacceptable. To avoid this problem, the actual offset into the current millisecond can be determined by looking at the number of accumulated instructions in the timer register. Since the register increments once every instruction and therefore at a rate of 8 megahertz, the register carries a finer time resolution of 1 / (2,000,000 instructions/second) = 0.5 microseconds. This corresponds to 165 micrometers of sound travel and should be more than adequate resolution to provide reasonable sonar altitude prediction.

5.7 Reviewing the 1998 SPU

5.7.1 Comparing the 1998 SPU to 1997 SPU

5.7.1a Hardware Comparison

For the most part, the hardware variations in the 1998 SPU from the 1997 SPU ultimately reduce the hardware overhead necessary. Contrary to the 1997 implementation, only one microprocessor is used and the need for an separate A/D converter has been eliminated. As a natural consequence, this also lowers the number of interconnections required. In short, the slimmer 1998 SPU hardware implementation allows for a simpler layout and design and increase robustness by removing excess parts.

5.7.1b Software Comparison

Immediately, the 1998 SPU software presents several differences when compared to that of the 1997 SPU. First and foremost, the code architecture is completely reworked to move all major functionality to the interrupt handlers, thus dismantling the combined interrupt and linear code layout of its predecessor. This comes from a distributed processing design methodology that identifies sensor processing on the SPU as event-driven as opposed to a simple progression of commands as in linear code. Furthermore, the 1998 software lacks the communication overhead that the 1997 dual-microcontroller layout incurred from transferring data from one PIC 16C73 to the other. Clearly, this not only reduces the overall code size and execution rate but increases the simplicity of design. In addition, since the 1998 SPU uses the PIC17C756 to carry out its own A/D conversion, this adds a computational burden to the microcontroller that the 1997 SPU circumvents by using a separate converter chip. But this added liability takes the place of the 1997 SPU communication overhead between the PIC16C73 and the off-board converter making it a one-to-one trade. Finally, both the data and transmission frequencies from the SPU have been increased.

5.7.2 1998 SPU Advantages Over 1997 SPU

The 1998 Sensor Processing Unit poses several advantages to the 1997 implementation. Most obviously, the hardware and software errors of its predecessors should no longer exist. This includes the correction of the anti-aliasing sampling rates, increased IMU and SPU packet transmission rates, as well as possibly removing spurious data packets by eliminating multiprocessor communications (see Section 4.2.3b). But more importantly, all of the original and revision objectives for the SPU have been fulfilled: Sensors are integrated into a simpler yet effective and robust design, computational load has been effectively distributed to a single microcontroller with an integrated A/D converter, and system accuracy and bandwidth have been increased via higher sampling rates and corrected 1997 implementation errors. Furthermore, the 1998 software package has been reengineered to follow a distributed processing approach that produces a process-oriented, event-driven architecture that is efficient, modular, reusable, and robust.

5.7.3 1998 SPU Disadvantages Over 1997 SPU

While the proposed 1998 Sensor Processing Unit purports to have distinct advantages to the 1997 design, the 1998 SPU has a singular disadvantage: it is not built. While the next-generation SPU has undergone and extensive development process, no prototype exists as a proof-of-concept. Without this model, experimental testing and evaluation cannot be carried out.

5.8 Summary

The 1998 Sensor Processing Units builds blueprints for the next generation in SPU design. In devising its architecture, the 1998 SPU addresses not only the original SPU objectives but also the revision concerns raised by the evaluation of the 1997 SPU. By following a holistic approach to design that incorporates distributed processing techniques, a generalized model for SPU structure has been developed from which the 1998 SPU bases its implementation. The final result is a product intended to effectively consolidates and administers the DSAAV navigational sensors, thus removing computational load away from the on-board computer, and uses less hardware and software to afford more accuracy and bandwidth than the 1997 SPU.

Chapter 6

1998 SPU Design Realization: Pipe Dreams

6.1 Introduction

While the 1998 Sensor Processing Unit looks promising on paper, it lacks a physical proof-of-concept and proper experimental analysis both in hardware and software. Nevertheless, speculations can be made for the most appropriate courses of action in realizing and investigating the actual implementation of the 1998 SPU.

6.2 SPU Hardware Prototyping Considerations

In developing any hardware device, several consideration exist which can guide the final implementation process. On a aircraft as small as the DSAAV, weight is a fundamental concern since this directly affects the load on the vehicle and, as a result, the aircraft's maneuverability. A lighter helicopter is much more agile than a similar but heavier aircraft. Power consumption is a large concern especially for autonomous systems since there is a limited power supply. Determining the power needs to adequately sustain the hardware directly affects the methods for delivering that energy. In addition, when dealing with a small autonomous vehicle, space is truly a commodity. As a result, the packaging technique becomes paramount in making efficient use of available real estate. Further, the packaging material itself adds to the weight of the aircraft thus reducing its nimbleness. Finally, due to the highly volatile environment caused by mechanical vibration, electronic interconnects must be extremely reliable and resistant to fatigue. This cannot be stressed enough since a simple connector failure can render the entire sensor processing system dysfunctional.

Each of these aspects of SPU prototyping should be considered when final implementation proceeds. In doing so, the resultant product will benefit in physical layout, reliability, as well as efficiency in power.

6.3 Software Simulation and Analysis

While the hardware for the SPU requires a physical prototype for experimental analysis, the 1998 software can be tested using simulation techniques. Applications for limited evaluation of PIC17C756 software is available from Microchip Technology, Inc. With these in hand, interrupt timing and responses as well as event handling can be examined for bugs. Furthermore, the software can be more quantitatively analyzed to determine whether or not the actual sensor interactions and timing rates are realizable. If so, the simulation programs could also be used as a tool to further refine and optimize these interactions in both speed and algorithmic efficiency thus producing more effective and powerful code.

6.4 Summary

While the 1998 SPU is in some ways a pipe dream, it is one which is founded in solid design methodology. Nevertheless, research and theories must ultimately be put to fire not with the intent to destroy, but rather refine and temper. Though well investigated and evaluated with respect to past designs, the 1998 SPU should be prototyped both in hardware and analyzed in software to further refine its implementation details. Nevertheless, the architectural foundation established for the 1998 SPU should prove to be a good basis for confirming, testing, and inspecting speculated behavior patterns as well as understanding any anomalies that might present themselves.

Chapter 7
Conclusion: Matters to Mind

By exploring the derivations of autonomous vehicles, the fundamental ideas and concepts that form the basis of these agents appear and clearly indicate the complexity of such machines. Able to make intelligent decisions as directed by their mission and scope, autonomous vehicles interact with their surroundings independent of human control. In order to do so, these devices clearly require a mode or modes of sensory perception that allow them to apprehend and interpret their environment, whether internal or external. But, achieving autonomous intelligent sensing is no easy task.

In revising the electronic sensor integration for the Draper Small Autonomous Vehicle, the 1998 Sensor Processing Unit came to fruition. By taking a holistic and systematic view of design, the 1998 SPU development process established a clear plan of attack for both SPU hardware and software implementation that circumvented the pitfalls inherent with the 1997 SPU's linear design approach in which one solution led to another, but not necessarily a best end result. Distributed processing techniques were applied to identify, formalize, and implement subsystems within the DSAAV to consolidate functions and functionality. This clarified the role of the SPU in the context of the overall application and formed the basis for evaluating the 1997 SPU. After revisiting the 1997 Sensor Processing Unit, a more effective architecture for a new real-time embedded sensor processing unit evolved which fulfilled and exceeded the original goals for the device. Designed to consolidate sensors into a single unit that removes the burden of sensor administration away from the navigational computer, the 1998 SPU works in tandem with the on-board CPU to provide the DSAAV with sensory perception that surpasses the system accuracy and bandwidth of its progenitor.

Interestingly, the framework for the 1998 SPU came not simply out of a reengineering process, but rather a fundamental discussion of autonomy, autonomous agents, and what they entail. As mentioned in Chapter 1, autonomous vehicles are complex devices that in some ways attempt to mimic both human mind and sight. Almost by definition, these machines implicitly carry some intelligence and an ability to interact with its environment. Both are required for handling even the most mundane of situations, let alone those that present hazards or evidence hostility. And while for the purposes of design the levels of intelligence and environmental awareness are limited to the scope of the project at hand, an underlying question remains: Can a machine operate and interact with human intelligence and purpose? While the answer may be unclear, the obvious point remains: technology has its limits, and these limits are ultimately human in nature.

In striving to reproduce in ways beyond the biological, mankind cannot avoid the bounds of its own being, including misconceptions and misunderstandings of self. In reaching for true, man-made autonomy, humans press on these confines whether technological or philosophical in nature. Some claim these restrictions are only temporary, capable of being broken with human advancement, and others accept different views. But even as people push on this fence, trying to see what lies behind, more questions appear, one after another. "Who created these limits?" "What and who is out there?" "Can man ever go beyond the heavens, or even ourselves?" "How?" "Why ask?" These matters to mind have enormous implication in many realms, metaphysical or even spiritual. But ironically, though hidden behind a veil of numbers and equations, the engineering world also faces them in many of its "real world" problems. As in the development of autonomous agents - machines that mirror men, the questions lie in wait for a very real answer.

Appendix A

1998 SPU PIC17C756 Software: 17c756.h

```
;***********************************************************************
;
;    17C756.H:  Ver. 1.00
;
;        Header File for the PIC17C756 microprocessor.
;
;        Programmer:              Chinsan Han
;        Original Release Date:   May 26, 1998
;
;
;        Notes:
;
;          - Future modified versions of this file should be
;            identified with a new version number.  Also, a
;            modification date should be added along with the
;            name of the programmer.
;
;***********************************************************************


#ifndef _17C756_H
#define _17C756_H

     processor 17c756


;=======================================================================
;    Interrupt Vector Assignments
;=======================================================================

#define   RESET_VECTOR                 0x0000
#define   RA0_INT_INTERRUPT_VECTOR     0x0008
#define   TMR0_INTERRUPT_VECTOR        0x0010
#define   T0CK1_INTERRUPT_VECTOR       0x0018
```

```
#define    PERIPHERAL_INTERRUPT_VECTOR    0x0020

;==========================================================================
;    Register File Map
;==========================================================================

;---------------------------------
;  Unbanked:  0x00 through 0x0F
;---------------------------------

     CBLOCK 0x00
          indf0, fsr0, pcl, pclath, alusta, t0sta, cpusta, intsta
          indf1, fsr1, wreg, tmr0l, tmr0h, tblptrl, tblptrh, bsr
     ENDC

;---------------------------------
;  Bank 0:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          porta, ddrb, portb, rcsta1, rcreg1, txsta1, txreg1, spbrg1
     ENDC

;---------------------------------
;  Bank 1:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          ddrc, portc, ddrd, portd, ddre, porte, pir1, pie1
     ENDC

;---------------------------------
;  Bank 2:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          tmr1, tmr2, tmr3l, tmr3h, pr1, pr2, pr3l, pr3h
     ENDC

     CBLOCK 0x10
          tmr1, tmr2, tmr3l, tmr3h, pr1, pr2, ca1l, ca1h
     ENDC

;---------------------------------
;  Bank 3:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          pw1dcl, pw2dcl, pw1dch, pw2dch, ca2l, ca2h, tcon1, tcon2
     ENDC

;---------------------------------
;  Bank 4:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          pir2, pie2
     ENDC
     CBLOCK 0x13
          rcsta2, rcreg2, txsta2, txreg2, spbrg2
     ENDC

;---------------------------------
;  Bank 5:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          ddrf, portf, ddrg, portg, adcon0, adcon1, adresl, adresh
     ENDC

;---------------------------------
;  Bank 6:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          sspadd, sspcon1, sspcon2, sspstat, sspbuf
     ENDC

;---------------------------------
;  Bank 7:  0x10 through 0x17
;---------------------------------

     CBLOCK 0x10
          pw3dcl, pw3dch, ca3l, ca3h, ca4l, ca4h, tcon3
     ENDC

;---------------------------------
;  Unbanked:  0x18 through 0x19
;---------------------------------

     CBLOCK 0x18
          prodl, prodh
     ENDC


;=======================================================================
;    Status and Control Register Bit Assignments
;=======================================================================

;------------------------------------------------
;  ALUSTA:  ALU Status Register
;------------------------------------------------

#define   ALUSTA_C            alusta,0
#define   ALUSTA_DC           alusta,1
#define   ALUSTA_Z            alusta,2
#define   ALUSTA_OV           alusta,3
#define   ALUSTA_FS0          alusta,4
#define   ALUSTA_FS1          alusta,5
#define   ALUSTA_FS2          alusta,6
#define   ALUSTA_FS3          alusta,7

;------------------------------------------------
;  T0STA:  Timer0 Status Register
;------------------------------------------------
```

```
#define   T0STA_T0PS0         t0sta,1
#define   T0STA_T0PS1         t0sta,2
#define   T0STA_T0PS2         t0sta,3
#define   T0STA_T0PS3         t0sta,4
#define   T0STA_T0CS          t0sta,5
#define   T0STA_T0SE          t0sta,6
#define   T0STA_INTEDG        t0sta,7

;-------------------------------------------------
;  CPUSTA:  CPU Status Register
;-------------------------------------------------

#define   CPUSTA__BOR         cpusta,0
#define   CPUSTA__POR         cpusta,1
#define   CPUSPA__PD          cpusta,2
#define   CPUSTA__TO          cpusta,3
#define   CPUSTA_GLINTD       cpusta,4
#define   CPUSTA_STKAV        cpusta,5

;-------------------------------------------------
;  INTSTA:  Interrupt Status Register
;-------------------------------------------------

#define   INTSTA_INTE         intsta,0
#define   INTSTA_T0IE         intsta,1
#define   INTSTA_T0CKIE       intsta,2
#define   INTSTA_PEIE         intsta,3
#define   INTSTA_INTF         intsta,4
#define   INTSTA_T0IF         intsta,5
#define   INTSTA_T0CKIF       intsta,6
#define   INTSTA_PEIF         intsta,7

;-------------------------------------------------
;  PIE1:  Peripheral Interrupt Enable Registers
;-------------------------------------------------

#define   PIE1_RC1IE          pie1,0
#define   PIE1_TX1IE          pie1,1
#define   PIE1_CA1IE          pie1,2
#define   PIE1_CA2IE          pie1,3
#define   PIE1_TMR1IE         pie1,4
#define   PIE1_TMR2IE         pie1,5
#define   PIE1_TMR3IE         pie1,6
#define   PIE1_RBIE           pie1,7

;-------------------------------------------------
;  PIE2:  Peripheral Interrupt Enable Registers
;-------------------------------------------------

#define   PIE2_RC2IE          pie2,0
#define   PIE2_TX2IE          pie2,1
#define   PIE2_CA3IE          pie2,2
#define   PIE2_CA4IE          pie2,3
#define   PIE2_ADIE           pie2,5
#define   PIE2_BCLIE          pie2,6
#define   PIE2_SSPIE          pie2,7

;-------------------------------------------------
;  PIR1:  Peripheral Interrupt Registers
```

```
;------------------------------------------------

#define   PIR1_RC1IF          pir1,0
#define   PIR1_TX1IF          pir1,1
#define   PIR1_CA1IF          pir1,2
#define   PIR1_CA2IF          pir1,3
#define   PIR1_TMR1IF         pir1,4
#define   PIR1_TMR2IF         pir1,5
#define   PIR1_TMR3IF         pir1,6
#define   PIR1_RBIF           pir1,7

;------------------------------------------------
;  PIR2:  Peripheral Interrupt Registers
;------------------------------------------------

#define   PIR2_RC2IF          pir2,0
#define   PIR2_TX2IF          pir2,1
#define   PIR2_CA3IF          pir2,2
#define   PIR2_CA4IF          pir2,3
#define   PIR2_ADIF           pir2,5
#define   PIR2_BCLIF          pir2,6
#define   PIR2_SSPIF          pir2,7

;------------------------------------------------
;  TXSTA1:  Transmit Status Registers
;------------------------------------------------

#define   TXSTA1_TX9D         txsta1,0
#define   TXSTA1_TRMT         txsta1,1
#define   TXSTA1_SYNC         txsta1,4
#define   TXSTA1_TXEN         txsta1,5
#define   TXSTA1_TX9          txsta1,6
#define   TXSTA1_CSRC         txsta1,7

;------------------------------------------------
;  TXSTA2:  Transmit Status Registers
;------------------------------------------------

#define   TXSTA2_TX9D         txsta2,0
#define   TXSTA2_TRMT         txsta2,1
#define   TXSTA2_SYNC         txsta2,4
#define   TXSTA2_TXEN         txsta2,5
#define   TXSTA2_TX9          txsta2,6
#define   TXSTA2_CSRC         txsta2,7

;------------------------------------------------
;  RCSTA1:  Receive Status Registers
;------------------------------------------------

#define   RCSTA1_RX9D         rcsta1,0
#define   RCSTA1_OERR         rcsta1,1
#define   RCSTA1_FERR         rcsta1,2
#define   RCSTA1_CREN         rcsta1,4
#define   RCSTA1_SREN         rcsta1,5
#define   RCSTA1_RX9          rcsta1,6
#define   RCSTA1_SPEN         rcsta1,7

;------------------------------------------------
;  RCSTA2:  Receive Status Registers
```

```
;-----------------------------------------------

#define   RCSTA2_RX9D        rcsta2,0
#define   RCSTA2_OERR        rcsta2,1
#define   RCSTA2_FERR        rcsta2,2
#define   RCSTA2_CREN        rcsta2,4
#define   RCSTA2_SREN        rcsta2,5
#define   RCSTA2_RX9         rcsta2,6
#define   RCSTA2_SPEN        rcsta2,7

;-----------------------------------------------
;  TCON1:  Timer Control Registers
;-----------------------------------------------

#define   TCON1_TMR1CS       tcon1,0
#define   TCON1_TMR2CS       tcon1,1
#define   TCON1_TMR3CS       tcon1,2
#define   TCON1_T16          tcon1,3
#define   TCON1_CA1ED0       tcon1,4
#define   TCON1_CA1ED1       tcon1,5
#define   TCON1_CA2ED0       tcon1,6
#define   TCON1_CA2ED1       tcon1,7

;-----------------------------------------------
;  TCON2:  Timer Control Registers
;-----------------------------------------------

#define   TCON2_TMR1ON       tcon2,0
#define   TCON2_TMR2ON       tcon2,1
#define   TCON2_TMR3ON       tcon2,2
#define   TCON2_CA1_PR3      tcon2,3
#define   TCON2_PWM1ON       tcon2,4
#define   TCON2_PWM2ON       tcon2,5
#define   TCON2_CA1OVF       tcon2,6
#define   TCON2_CA2OVF       tcon2,7

;-----------------------------------------------
;  TCON3:  Timer Control Registers
;-----------------------------------------------

#define   TCON3_PWM3ON       tcon3,0
#define   TCON3_CA3ED0       tcon3,1
#define   TCON3_CA3ED1       tcon3,2
#define   TCON3_CA4ED0       tcon3,3
#define   TCON3_CA4ED1       tcon3,4
#define   TCON3_CA3OVF       tcon3,5
#define   TCON3_CA4OVF       tcon3,6

;-----------------------------------------------
;  ADCON0:  Analog-to-Digital Control Registers
;-----------------------------------------------

#define   ADCON0_ADON        adcon0,0
#define   ADCON0_GO_DONE     adcon0,2
#define   ADCON0_CHS0        adcon0,4
#define   ADCON0_CHS1        adcon0,5
#define   ADCON0_CHS2        adcon0,6
#define   ADCON0_CHS3        adcon0,7
```

```
;------------------------------------------------
;  ADCON1:  Analog-to-Digital Control Registers
;------------------------------------------------

#define   ADCON1_PCFG0        adcon1,0
#define   ADCON1_PCFG1        adcon1,1
#define   ADCON1_PCFG2        adcon1,2
#define   ADCON1_PCFG3        adcon1,3
#define   ADCON1_ADFM         adcon1,5
#define   ADCON1_ADCS0        adcon1,6
#define   ADCON1_ADCS1        adcon1,7

;------------------------------------------------
;  SSPSTAT:  SSP Status Register
;------------------------------------------------

#define   SSPSTAT_BF          sspstat,0
#define   SSPSTAT_UA          sspstat,1
#define   SSPSTAT_R_W         sspstat,2
#define   SSPSTAT_S           sspstat,3
#define   SSPSTAT_P           sspstat,4
#define   SSPSTAT_D_A         sspstat,5
#define   SSPSTAT_CKE         sspstat,6
#define   SSPSTAT_SMP         sspstat,7

;------------------------------------------------
;  SSPCON1:  SSP Control Registers
;------------------------------------------------

#define   SSPCON1_SSPM0       sspcon1,0
#define   SSPCON1_SSPM1       sspcon1,1
#define   SSPCON1_SSPM2       sspcon1,2
#define   SSPCON1_SSPM3       sspcon1,3
#define   SSPCON1_CKP         sspcon1,4
#define   SSPCON1_SSPEN       sspcon1,5
#define   SSPCON1_SSPOV       sspcon1,6
#define   SSPCON1_WCOL        sspcon1,7

;------------------------------------------------
;  SSPCON2:  SSP Control Registers
;------------------------------------------------

#define   SSPCON2_SEN         sspcon2,0
#define   SSPCON2_RSEN        sspcon2,1
#define   SSPCON2_PEN         sspcon2,2
#define   SSPCON2_RCEN        sspcon2,3
#define   SSPCON2_ACKEN       sspcon2,4
#define   SSPCON2_ACKDT       sspcon2,5
#define   SSPCON2_ACKSTAT     sspcon2,6
#define   SSPCON2_GCEN        sspcon2,7


;==================================================================
;    Port Bit Assignments
;==================================================================

;-----------
;  Port A
;-----------
```

```
#define   RA0       porta,0
#define   RA1       porta,1
#define   RA2       porta,2
#define   RA3       porta,3
#define   RA4       porta,4
#define   RA5       porta,5

;-----------
;  Port B
;-----------

#define   RB0       portb,0
#define   RB1       portb,1
#define   RB2       portb,2
#define   RB3       portb,3
#define   RB4       portb,4
#define   RB5       portb,5
#define   RB6       portb,6
#define   RB7       portb,7

;-----------
;  Port C
;-----------

#define   RC0       portc,0
#define   RC1       portc,1
#define   RC2       portc,2
#define   RC3       portc,3
#define   RC4       portc,4
#define   RC5       portc,5
#define   RC6       portc,6
#define   RC7       portc,7

;-----------
;  Port D
;-----------

#define   RD0       portd,0
#define   RD1       portd,1
#define   RD2       portd,2
#define   RD3       portd,3
#define   RD4       portd,4
#define   RD5       portd,5
#define   RD6       portd,6
#define   RD7       portd,7

;-----------
;  Port E
;-----------

#define   RE0       porte,0
#define   RE1       porte,1
#define   RE2       porte,2
#define   RE3       porte,3

;-----------
;  Port F
;-----------
```

```
#define   RF0       portf,0
#define   RF1       portf,1
#define   RF2       portf,2
#define   RF3       portf,3
#define   RF4       portf,4
#define   RF5       portf,5
#define   RF6       portf,6
#define   RF7       portf,7

;-----------
;  Port G
;-----------

#define   RG0       portg,0
#define   RG1       portg,1
#define   RG2       portg,2
#define   RG3       portg,3
#define   RG4       portg,4
#define   RG5       portg,5
#define   RG6       portg,6
#define   RG7       portg,7


;==========================================================================
;    Analog Input Assignments
;==========================================================================

#define   AN0       portg,3
#define   AN1       portg,2
#define   AN2       portg,1
#define   AN3       portg,0
#define   AN4       portf,0
#define   AN5       portf,1
#define   AN6       portf,2
#define   AN7       portf,3
#define   AN8       portf,4
#define   AN9       portf,5
#define   AN10      portf,6
#define   AN11      portf,7

#define   AN_VREF_NEG         portg,2
#define   AN_VREF_POS         portg,3


;==========================================================================
;    Equivalence Assignments
;==========================================================================

W         equ       0
F         equ       1


#endif
```

Appendix B

1998 SPU PIC17C756 Software: spu-p17.asm

```
;***********************************************************************
;
;  SPU-P17.ASM:  Ver. 1.00
;
;    PIC17C756 microprocessor controller code for the
;    Sensor Processing Unit.
;
;    Original Programmer:               Chinsan Han
;    Original Release Date:             May 26, 1998
;
;    Notes:
;
;         Future modified versions of this file should be
;         identified with a new version number.  Also, a
;         modification date should be added along with the
;         name of the programmer.
;
;***********************************************************************


#include  "17c756.h"


;=======================================================================
;  Aliases
;=======================================================================

;-----------------------------------------------------------------------
;  Sensor Enable Aliases
;
;    Each sensor may enabled or disabled by assigning a value of
;    one or zero respectively to their corresponding enable alias.
;-----------------------------------------------------------------------
```

```
#define   IMU_ANGULAR_ENABLE       0x01
#define   IMU_LINEAR_ENABLE        0x01
#define   SONAR_ENABLE             0x01
#define   COMPASS_ENABLE           0x01
#define   BATTERY_ENABLE           0x01
#define   TEMPERATURE_ENABLE       0x01

;--------------------------------------------------------------------------
;  Sample Period Aliases
;
;    All sample periods must be given in integer global timer
;    periods per sample and should be stated in hexidecimal.  The
;    minimum number of periods per sample is one.  As a result,
;    the minimum sample period is equal to the global timer period.
;
;    Currently, the global timer period is one millisecond.
;    See "Calibrate_Global_Timer" subroutine for details.
;
;    Example:  Global Timer Period == 1 millisecond
;         Desired Sample Rate == 25 Hertz
;         Sample Period = 1 / (25 Hertz) == 40 milliseconds
;
;         # Sensor Timer Increments Per Sample Period
;              == Sample Period / Global Timer Period
;              == 40 milliseconds / 1 millisecond
;              == 40 (0x28)
;--------------------------------------------------------------------------

#define   AD_PERIOD           0x0A ;  10 milliseconds (100 hertz)
#define   SONAR_PERIOD        0x32 ;  50 milliseconds (20 hertz)
#define   COMPASS_PERIOD      0xC8 ;  200 milliseconds (5 hertz)

#define   TRANSMIT_PERIOD     0x0A ;  10 milliseconds (100 hertz)

;--------------------------------------------------------------------------
;  Timer0 Offset Aliases
;--------------------------------------------------------------------------

#define   TMR0_OFFSET_HIGH    0xF8
#define   TMR0_OFFSET_LOW     0x2F ; 0xF82F == 2000

;--------------------------------------------------------------------------
;  Pin Aliases
;--------------------------------------------------------------------------

#define   SERIAL_PORT         RB5

#define   SONAR_ECHO          RA0
#define   SONAR_BINH          RC0
#define   SONAR_INIT          RC1

#define   COMPASS_SDO         RC3
#define   COMPASS_SCLK        RC4
#define   COMPASS_SS          RC5
#define   COMPASS_EOC         RC6
#define   COMPASS_PC          RC7

#define   IMU_GYRO_X          AN4
#define   IMU_GYRO_Y          AN5
```

```
#define   IMU_GYRO_Z          AN6
#define   IMU_LINEAR_X        AN7
#define   IMU_LINEAR_Y        AN8
#define   IMU_LINEAR_Z        AN9
#define   IMU_TEMPERATURE     AN10
#define   BATTERY             AN11

;----------------------------------------------------------------------
;  First Analog Channel Alias
;
;    Currently, analog channel 4 is the first.
;----------------------------------------------------------------------

#define   ANALOG_CHANNEL_START          4

;----------------------------------------------------------------------
;  "sensor_status" Bit Aliases
;
;    See "Sensor Status Variable Address Assignments" for
;    variable declaration.
;
;    Note:  "sensor_status" is unbanked.
;----------------------------------------------------------------------

#define   IMU_ALIVE           sensor_status, 0
#define   COMPASS_ALIVE       sensor_status, 1
#define   SONAR_FRESH         sensor_status, 2
#define   IMU_FRESH           sensor_status, 3
#define   COMPASS_FRESH       sensor_status, 4

;----------------------------------------------------------------------
;  "in_progress" Bit Aliases
;
;    See "Sensor Status Variable Address Assignments" for
;    variable declaration.
;
;    Note:  "in_progress" is unbanked.
;----------------------------------------------------------------------

#define   AD_IN_PROGRESS            in_progress, 0
#define   TRANSMIT_IN_PROGRESS      in_progress, 1

;----------------------------------------------------------------------
;  Sensor Information Bit Aliases
;
;    See "Sensor Status Variable Address Assignments" for
;    variable declaration.
;
;    Note:  "sensor_info" is unbanked.
;----------------------------------------------------------------------

#define   SONAR_ECHO_RECEIVED       sensor_info, 0
#define   COMPASS_ERROR             sensor_info, 1

;----------------------------------------------------------------------
;  Data Address Aliases
;----------------------------------------------------------------------

#define   AD_DATA_START_ADDRESS             0x49
```

```
#define   NON_IMU_DATA_START_ADDRESS        0x55 ;  Temperature
#define   AD_DATA_END_ADDRESS               0x56

#define   CHECKSUM_START_ADDRESS            0x43 ;  Starts w/ size
#define   CHECKSUM_END_ADDRESS              0x57

#define   PACKET_DATA_START_ADDRESS         0x40
#define   PACKET_DATA_END_ADDRESS           0x57

#define   TRANSMIT_DATA_START_ADDRESS       0x60
#define   TRANSMIT_DATA_END_ADDRESS         0x77


;========================================================================
;  Data Variable Address Assignments
;========================================================================

;------------------------------------------------------------------------
;  Sensor Status Variable Address Assignments:  Unbanked
;
;    Sensor Status Variable Information:
;         sensor_status: Sensor status information
;                             Bit 0:  IMU alive
;                             Bit 1:  Compass alive
;                             Bit 2:  Sonar data fresh
;                             Bit 3:  IMU data fresh
;                             Bit 4:  Compass data fresh
;         in_progress:   Sensor sampling state information
;                             Bit 0:  AD in progress
;                             Bit 1:  Transmission in progress
;         sensor_info:   Sensor Information
;                             Bit 0:  Sonar Ping Received
;                             Bit 1:  Compass error
;------------------------------------------------------------------------

     CBLOCK 0x1A
          sensor_status  ;  @ 0x1A, unbanked
          in_progress
          sensor_info    ;  @ 0x1C, unbanked
     ENDC

;------------------------------------------------------------------------
;  Temporary Variable Address Assignments:  Unbanked
;------------------------------------------------------------------------

     CBLOCK 0x1D
          temp0          ;  @ 0x1D, unbanked
          temp1
          temp2          ;  @ 0x1F, unbanked
     ENDC

;------------------------------------------------------------------------
;  Interrupt Register Save Variable Address Assignments:  GPR Bank 0
;------------------------------------------------------------------------

     CBLOCK 0x20
          alusta_save    ;  @ 0x20, GPR bank 0
          wreg_save
          fsr0_save
          fsr1_save
          indf0_save
          indf1_save
          bsr_save       ;  @ 0x26, GPR bank 0
     ENDC

;-----------------------------------------------------------------------
;  Timer Variable Address Assignments:  GPR Bank 0
;-----------------------------------------------------------------------

     CBLOCK 0x30
          ad_timer       ;  @ 0x30, GPR bank 0
          sonar_timer
          compass_timer
          transmit_timer ;  @ 0x33, GPR bank 0
     ENDC

;-----------------------------------------------------------------------
;  Packet Data Variable Address Assignments:  GPR Bank 0
;
;    All sensor and packet data is stored in this bank of variables.
;
;    Packet Data Variable Information:
;         start1:             First packet header byte (0xAB)
;         start2:             Second packet header byte     (0xCD)
;         start3:             Third packet header byte (0xCD)
;         packet_size:        # of packet bytes excluding
;                             start bytes
;         status:             Sensor status information
;                                  Bit 0:  IMU alive
;                                  Bit 1:  Compass alive
;                                  Bit 2:  Sonar data fresh
;                                  Bit 3:  IMU data fresh
;                                  Bit 4:  Compass data fresh
;         sonar_high/low:     High/low byte of sonar data
;         compass_high/low:   High/low byte of compass data
;         gyro_x_high/low:    High/low byte of IMU x-gyro data
;         gyro_y_high/low:    High/low byte of IMU y-gyro data
;         gyro_z_high/low:    High/low byte of IMU z-gyro data
;         linear_x_high/low:  High/low byte of IMU x-linear data
;         linear_y_high/low:  High/low byte of IMU y-linear data
;         linear_z_high/low:  High/low byte of IMU z-linear data
;         temperature:        Vehicle temperature data
;         battery:            Battery voltage data
;         checksum:           Packet data checksum
;-----------------------------------------------------------------------

     CBLOCK 0x40
          start1         ;  @ 0x40, GPR bank 0
          start2
          start3
          packet_size
          status
          sonar_high
          sonar_low
          compass_high
          compass_low
          gyro_x_high    ;  @ 0x49, GPR bank 0
          gyro_x_low
          gyro_y_high
          gyro_y_low
          gyro_z_high
          gyro_z_low     ;  @ 0x4E, GPR bank 0
          linear_x_high  ;  @ 0x4F, GPR bank 0
          linear_x_low
          linear_y_high
          linear_y_low
          linear_z_high
          linear_z_low   ;  @ 0x54, GPR bank 0
          temperature    ;  @ 0x55, GPR bank 0
          battery        ;  @ 0x56, GPR bank 0
          checksum       ;  @ 0x57, GPR bank 0
     ENDC

;-----------------------------------------------------------------------
;  Transmission Variable Address Assignments:  GPR Bank 0
;
;    When a packet is ready for transmission, the individual data
;    values are copied to a transmission bank which is protected
;    from updates during the transmission.
;
;    All transmission bank variables add the prefix "tx_hold" to
;    their corresponding packet data identifiers.
;-----------------------------------------------------------------------

     CBLOCK 0x60
          tx_hold_start1           ;  @ 0x60, GPR bank 0
          tx_hold_start2
          tx_hold_start3
          tx_hold_packet_size
          tx_hold_status
          tx_hold_sonar_high
          tx_hold_sonar_low
          tx_hold_compass_high
          tx_hold_compass_low
          tx_hold_gyro_x_high
          tx_hold_gyro_x_low
          tx_hold_gyro_y_high
          tx_hold_gyro_y_low
          tx_hold_gyro_z_high
          tx_hold_gyro_z_low
          tx_hold_linear_x_high
          tx_hold_linear_x_low
          tx_hold_linear_y_high
          tx_hold_linear_y_low
          tx_hold_linear_z_high
          tx_hold_linear_z_low
          tx_hold_temperature
          tx_hold_battery
          tx_hold_checksum         ;  @ 0x77, GPR bank 0
     ENDC

;-----------------------------------------------------------------------
;  Temporary Sensor Data Address Assignments:  GPR Bank 0
;-----------------------------------------------------------------------

     CBLOCK 0x80
          sonar_counter_high       ;  @ 0x80, GPR bank 0
          sonar_counter_low
          sonar_offset_high
          sonar_offset_low
          compass_temp_high
          compass_temp_low
          AD_data_temp            ;  @ 0x84, GPR bank 0
     ENDC

;-----------------------------------------------------------------------
;  Miscellaneous Variable Address Assignments:  GPR Bank 0
;-----------------------------------------------------------------------

     CBLOCK 0x90
          current_channel              ;  @ 0x90, GPR bank 0
          current_channel_data_address
          current_packet_byte_address  ;  @ 0x92, GPR bank 0
     ENDC


;=======================================================================
;  Vector Directives
;
;    This is the code starting point after any reset of the
;    PIC17C756 microprocessor.
;
;=======================================================================

     ORG       RESET_VECTOR
     GOTO      Main

     ORG       RA0_INT_INTERRUPT_VECTOR
     GOTO      Sonar_Echo_Interrupt_Handler

     ORG       TMR0_INTERRUPT_VECTOR
     GOTO      Millisecond_Interrupt_Handler

     ORG       PERIPHERAL_INTERRUPT_VECTOR
     GOTO      Conversion_Transmission_Interrupt_Handler


;=======================================================================
;  Main
;=======================================================================

Main:

;  BEGIN Main
     CALL      Initialize_Microprocessor     ;  init. microprocessor
     CALL      Initialize_Variables          ;  init. variables
     CALL      Calibrate_Global_Timer        ;  prepare tmr0 for first
                                             ;  millisecond interrupt

     CALL      Initialize_Sonar
     CALL      Initialize_Compass

     BCF       CPUSTA_GLINTD                 ;  enable all interrupts

Main_Loop:
     GOTO      Main_Loop

;  END Main


;=========================================================================
;  Initialize_Microprocessor
;
;    Carry out necessary microprocessor initializations for the
;    control registers, ports, etc. upon microprocessor reset.
;=========================================================================

Initialize_Microprocessor:

;  BEGIN Initialize_Microprocessor
;-------------------------------------------------------------------------
;    Control Register Initializations
;
;    Erring on the side of caution, ALL control register values are
;    initialized regardless of whether or not they are utilized.
;    Consequently, any future modification which seeks to take
;    advantage of currently unimplemented registers will need to
;    make initialization code corrections as necessary.
;-------------------------------------------------------------------------

;  ALUSTA Register:  Unbanked
     BSF  ALUSTA_FS0          ;**FSR0, AUTO POST-INCREMENT
     BCF  ALUSTA_FS1
     BSF  ALUSTA_FS2          ;**FSR1, AUTO POST-INCREMENT
     BCF  ALUSTA_FS3

;  T0STA Register:  Unbanked
     BCF  T0STA_T0PS0         ;**1:4 PRESCALE ON TMR0
     BSF  T0STA_T0PS1
     BCF  T0STA_T0PS2
     BCF  T0STA_T0PS3
     BSF  T0STA_T0CS          ;**TMR0 SOURCE IS INTERNAL (Fosc/4)
     BCF  T0STA_T0SE          ;  not used (don't care)
     BSF  T0STA_INTEDG        ;**RA0/INT INTERRUPT ON RISING EDGE

;  CPUSTA Register:  Unbanked
     BSF  CPUSTA_GLINTD       ;**ALL INTERRUPTS DISABLED

;  INTSTA Register:  Unbanked
     BCF  INTSTA_INTE         ;**RA0/INT INTERRUPT DISABLED
     BSF  INTSTA_T0IE         ;**TMR0 OVERFLOW INTERRUPT ENABLED
     BCF  INTSTA_T0CKIE       ;  not used (no ra1/t0ck1 int)
     BSF  INTSTA_PEIE         ;**PIE INTERRUPTS ENABLED
     BCF  INTSTA_INTF         ;  not used (no edge on ra0/int)
     BCF  INTSTA_T0IF         ;**TMR0 NOT OVERFLOWED
     BCF  INTSTA_T0CKIF       ;  not used (no edge on ra1/tock1)

;  PIE1 Register:  SFR Bank 1
     MOVLB     1              ;  switch to SFR bank 1
     BCF  PIE1_RC1IE          ;  not used (no usart1 receive int)
     BCF  PIE1_TX1IE          ;**USART1 TRANSMIT INT. DISABLED
     BCF  PIE1_CA1IE          ;  not used (no capture1 int)
     BCF  PIE1_CA2IE          ;  not used (no capture2 int)
     BCF  PIE1_TMR1IE         ;  not used (no tmr1 int)
```

```
    BCF   PIE1_TMR2IE          ;  not used (no tmr2 int)
    BCF   PIE1_TMR3IE          ;  not used (no tmr3 int)
    BCF   PIE1_RBIE            ;  not used (no int on PORTB change)

;  PIE2 Register:  SFR Bank 4
    MOVLB     4                ;  switch to SFR bank 4
    BCF   PIE2_RC2IE           ;  not used (no usart2 receive int)
    BCF   PIE2_TX2IE           ;  not used (no usart2 transmit int)
    BCF   PIE2_CA3IE           ;  not used (no capture3 interrupt)
    BCF   PIE2_CA4IE           ;  not used (no capture4 interrupt)
    BCF   PIE2_ADIE            ;**A/D CONVERSION INTERRUPT DISABLED
    BCF   PIE2_BCLIE           ;  not used (no bus collision int)
    BCF   PIE2_SSPIE           ;  not used (no ssp int)

;  PIR1 Register:  SFR Bank 1
    MOVLB     1
    BCF   PIR1_CA1IF           ;  not used (capture1 int flag)
    BCF   PIR1_CA2IF           ;  not used (capture2 int flag)
    BCF   PIR1_TMR1IF          ;  not used (tmr1 interrupt flag)
    BCF   PIR1_TMR2IF          ;  not used (tmr2 interrupt flag)
    BCF   PIR1_TMR3IF          ;  not used (tmr3 interrupt flag)
    BCF   PIR1_RBIF            ;  not used (portb change int flag)

;  PIR2 Register:  SFR Bank 4
    MOVLB     4
    BCF   PIR2_CA3IF           ;  not used (capture3 int flag)
    BCF   PIR2_CA4IF           ;  not used (capture4 int flag)
    BCF   PIR2_ADIF            ;**A/D MODULE INTERRUPT FLAG
    BCF   PIR2_BCLIF           ;  not used (bus collision int flag)
    BCF   PIR2_SSPIF           ;  not used (ssp interrupt flag)

;  TXSTA1 Register:  SFR Bank 0
    MOVLB     0                ;  switch to SFR bank 0
    BCF   TXSTA1_TX9D          ;  not used (usart1 9th bit is 0)
    BCF   TXSTA1_SYNC          ;**USART1 IN ASYCHRONOUS MODE
    BCF   TXSTA1_TXEN          ;**USART1 TRANSMIT DISABLED
    BCF   TXSTA1_TX9           ;**USART1 8-BIT TRANSMIT
    BCF   TXSTA1_CSRC          ;  not used (don't care in asynch)

;  TXSTA2 Register:  SFR Bank 4
    MOVLB     4                ;  switch to SFR bank 4
    BCF   TXSTA2_TX9D          ;  not used (usart2 9th bit is 0)
    BCF   TXSTA2_SYNC          ;  not used (usart2 in asynch mode)
    BCF   TXSTA2_TXEN          ;  not used (usart2 tx disabled)
    BCF   TXSTA2_TX9           ;  not used (usart2 8-bit transmit)
    BCF   TXSTA2_CSRC          ;  not used (don't care in asynch)

;  RCSTA1 Register:  SFR Bank 0
    MOVLB     0                ;  switch to SFR bank 0
    BCF   RCSTA1_CREN          ;  not used (no continuous receive)
    BCF   RCSTA1_SREN          ;  not used (don't care in asynch)
    BCF   RCSTA1_RX9           ;  not used (usart1 8-bit receive)
    BCF   RCSTA1_SPEN          ;**USART1 DISABLED

;  RCSTA2 Register:  SFR Bank 4
    MOVLB     4                ;  switch to SFR bank 4
    BCF   RCSTA2_CREN          ;  not used (no continuous receive)
    BCF   RCSTA2_SREN          ;  not used (don't care in asynch)
    BCF   RCSTA2_RX9           ;  not used (usart2 8-bit receieve)
```

```
      BCF  RCSTA2_SPEN          ;  not used (usart2 disabled)

;  TCON1 Register:  SFR Bank 3
      MOVLB     3               ;  switch to SFR bank 3
      BCF  TCON1_TMR1CS         ;  not used (tmr1 at Fosc/4)
      BCF  TCON1_TMR2CS         ;  not used (tmr2 at Fosc/4)
      BCF  TCON1_TMR3CS         ;  not used (tmr3 at Fosc/4)
      BCF  TCON1_T16            ;  not used (tmr2/tmr1 8-bit timers)
      BCF  TCON1_CA1ED0         ;  not used (falling edge capture)
      BCF  TCON1_CA1ED1
      BCF  TCON1_CA2ED0         ;  not used (falling edge capture)
      BCF  TCON1_CA2ED1

;  TCON2 Register:  SFR Bank 3
      MOVLB     3               ;  switch to SFR bank 3
      BCF  TCON2_TMR1ON         ;  not used (tmr1 disabled)
      BCF  TCON2_TMR2ON         ;  not used (tmr2 disabled)
      BCF  TCON2_TMR3ON         ;  not used (tmr3 disabled)
      BCF  TCON2_CA1_PR3        ;  not used (tmr3 period register)
      BCF  TCON2_PWM1ON         ;  not used (pwm1 disabled)
      BCF  TCON2_PWM2ON         ;  not used (pwm2 disabled)

;  TCON3 Register:  SFR Bank 7
      MOVLB 7                   ;  switch to SFR bank 7
      BCF  TCON3_PWM3ON         ;  not used (pwm 3 disabled)
      BCF  TCON3_CA3ED0         ;  not used (falling edge capture)
      BCF  TCON3_CA3ED1
      BCF  TCON3_CA4ED0         ;  not used (falling edge capture)
      BCF  TCON3_CA4ED1

;  ADCON0 Register:  SFR Bank 5
      MOVLB     5               ;  switch to SFR bank 5
      BCF  ADCON0_ADON          ;**A/D CONVERTER OFF
      BCF  ADCON0_GO_DONE       ;**NO A/D CONVERSION IN PROGRESS
      BCF  ADCON0_CHS0          ;**ANALOG CHANNEL 0 SELECTED
      BCF  ADCON0_CHS1
      BCF  ADCON0_CHS2
      BCF  ADCON0_CHS3

;  ADCON1 Register:  SFR Bank 5
      MOVLB     5               ;  switch to SFR bank 5
      BSF  ADCON1_PCFG0         ;**A/D REFERENCE IS Vref+ and Vref-
      BCF  ADCON1_PCFG1         ;**AN<0:11> ARE ANALOG INPUTS
      BCF  ADCON1_PCFG2
      BCF  ADCON1_PCFG3
      BSF  ADCON1_ADFM          ;**A/D RESULT RIGHT JUSTIFIED
      BCF  ADCON1_ADCS0         ;**A/D CONVERSION CLOCK AT Fosc/64
      BSF  ADCON1_ADCS1

;  SSPSTAT Register:  SFR Bank 6
      MOVLB     6               ;  switch to SFR bank 6
      BCF  SSPSTAT_CKE          ;  not used (SCK falling edge tx)
      BCF  SSPSTAT_SMP          ;  not used (middle of time sample)

;  SSPCON1 Register:  SFR Bank 6
      MOVLB     6               ;  switch to SFR bank 6
      BCF  SSPCON1_SSPM0        ;  not used (SPI master mode,Fosc/4)
      BCF  SSPCON1_SSPM1
      BCF  SSPCON1_SSPM2
```

```
    BCF  SSPCON1_SSPM3
    BCF  SSPCON1_CKP           ;  not used (clock idle state low)
    BCF  SSPCON1_SSPEN         ;  not used (ssp disabled)

;  SSPCON2 Register:  SFR Bank 6
    MOVLB     6                ;  switch to SFR bank 6
    BCF  SSPCON2_SEN           ;  not used (start condition idle)
    BCF  SSPCON2_RSEN          ;  not used (restart condition idle)
    BCF  SSPCON2_PEN           ;  not used (stop condition idle)
    BCF  SSPCON2_RCEN          ;  not used (receive idle)
    BCF  SSPCON2_ACKEN         ;  not used (acknowledge seq idle)
    BCF  SSPCON2_ACKDT         ;  not used (acknowledge)
    BCF  SSPCON2_GCEN          ;  not used (no gen call address)

;----------------------------------------------------------------------
;    Port Register Initializations
;----------------------------------------------------------------------

;  Port A
    MOVLB     0          ;  switch to SFR bank 0
    MOVLW     0xFF
    MOVWF     porta      ;  inputs/floating outputs: RA<0:5>
                         ;    RA<0> == SONAR_ECHO (input)
                         ;    RA<5> == TX1 (output)

;  Port B
    MOVLB     0          ;  switch to SFR bank 0
    CLRF      portb      ;  init. port B data latches
                         ;  before setting DDRB
    MOVLW     0xFF
    MOVWF     ddrb       ;  inputs:      RB<0:7>
                         ;  outputs:  none

;  Port C
    MOVLB     1          ;  switch to SFR bank 1
    CLRF      portc      ;  init. port C data latches
                         ;  before setting DDRC
    MOVLW     0x4C
    MOVWF     ddrc       ;  inputs:   RC<2:3>, RC<6>
                         ;  outputs:  RC<0:1>, RC<4:5>, RC<7>
                         ;    RC<0> == SONAR_BINH  (output)
                         ;    RC<1> == SONAR_INIT  (output)
                         ;    RC<3> == COMPASS_SDO      (input)
                         ;    RC<4> == COMPASS_SCLK    (output)
                         ;    RC<5> == COMPASS_SS  (output)
                         ;    RC<6> == COMPASS_EOC     (input)
                         ;    RC<7> == COMPASS_PC  (output)

;  Port D
    MOVLB     1          ;  switch to SFR bank 1
    CLRF      portd      ;  init. port d data latches
                         ;  before setting DDRD
    MOVLW     0xFF
    MOVWF     ddrd       ;  inputs:      RD<0:7>
                         ;  outputs:  none

;  Port E
    MOVLB     1          ;  switch to SFR bank 1
    CLRF      porte      ;  init. port E data latches
```

```
                          ;  before setting DDRE
     MOVLW     0x0F
     MOVWF     ddre       ;  inputs:      RE<0:3>
                          ;  outputs:  none
                          ;    RE<4:7> always read as '0'

;  Port F
     MOVLB     5          ;  switch to SFR bank 5
     CLRF      portf      ;  init. port F data latches
                          ;  before setting DDRF
     MOVLW     0xFF
     MOVWF     ddrf       ;  inputs:   RF<0:7>
                          ;  outputs:  none

;  Port G
     MOVLB     5          ;  switch to SFR bank 5
     CLRF      portg      ;  init. port G data latches
                          ;  before setting DDRG
     MOVLW     0xFF
     MOVWF     ddrg       ;  inputs:   RG<0:7>
                          ;  outputs:  none

;  END Initialize_Microprocessor, return

     RETURN


;========================================================================
;    Initialize_Variables
;========================================================================

Initialize_Variables:

;  BEGIN Initialize_Variables
;  Unbanked Variables
     CLRF      sensor_status        ;  clear sensor status
                                    ; assumes IMU and compass are dead
                                    ;  assumes no fresh data available

     CLRF      in_progress          ;  assumes no sample or transmission
                                    ;  in progress

     CLRF      sensor_info          ;  assumes no sensor info

     CLRF      current_channel
     CLRF      current_channel_data_address
     CLRF      current_packet_byte_address

;  GPR Bank 0 Variables
     MOVLR     0                    ;  switch to GPR bank 0

     CLRF      ad_timer             ;  clear all timer values
     CLRF      sonar_timer
     CLRF      compass_timer
     CLRF      transmit_timer

     MOVLW     0xAB                 ;  data packet header bytes
     MOVWF     start1
     MOVLW     0xCD
```

```
        MOVWF     start2
        MOVLW     0xEF
        MOVWF     start3

        MOVLW     0x15                ;  packet size == 21 (0x15) bytes
        MOVWF     packet_size

        CLRF      status              ;  clear sensor status
                                      ;  assumes IMU and compass are dead
                                      ;  assumes no fresh data available

        CLRF      sonar_high          ;  sonar data initialized zero
        CLRF      sonar_low

        CLRF      compass_high        ;  compass data initialized zero
        CLRF      compass_low

        CLRF      gyro_x_high         ;  IMU gyro data initialized zero
        CLRF      gyro_x_low
        CLRF      gyro_y_high
        CLRF      gyro_y_low
        CLRF      gyro_z_high
        CLRF      gyro_z_low

        CLRF      linear_x_high       ;  IMU linear data initialized zero
        CLRF      linear_x_low
        CLRF      linear_y_high
        CLRF      linear_y_low
        CLRF      linear_z_high
        CLRF      linear_z_low

        CLRF      temperature         ;  temperature initalized zero

        CLRF      battery             ;  battery initialized zero

        CLRF      checksum            ;  checksum initialized zero

        CLRF      current_channel     ;  current_channel initialized zero

;  END Initialize_Variables, return
        RETURN


;=======================================================================
;    Calibrate_Global_Timer
;
;    The "Global Timer" is a millisecond interrupt generated by an
;    overflow (roll over from 0xFFFF to 0x0000) in the 16-bit timer0
;    register, TMR0H:TMR0L.
;
;    The 16-bit timer0 register increments at a rate of Fosc/4.
;    This register needs to be initialized with an offset value
;    such that number of increments between the offset and the
;    overflow value 0xFFFF is equal to the number of timer
;    increments per millisecond.
;
;    Example:  Ftmr0 == (Fosc / 4) * (1/4) [prescale]
;                     == 32 megahertz / 16
;                     == 2 megahertz
```

```
;
;               Ttmr0 == 1 / Ftmr0
;                    == 1 / 2 megahertz
;                    == 0.5 microsecond
;
;               # of Timer Increments Per Millisecond
;                    == 1 millisecond / Ttmr0
;                    == 1 millisecond / 0.5 microsecond
;                    == 2000 (0x7D0)
;
;               Timer0 Offset Value
;                    == Maximum Timer0 Value -
;                       # of Timer Increments/Millisecond
;                    == 0xFFFF - 0x7D0
;                    == 0xF82F
;
;               As seen in this example, each timer increment
;               has a period of Ttmr0 == 0.5 microseconds.
;               By initalizing the timer to 0xF82F, the total
;               number of timer increments between this
;               inital value and the rollover at 0xFFFF is
;               0x7D0, or 2000 in decimal. This difference
;               represents a total time of:
;
;               2000 * 0.5 microseconds == 1 millisecond.
;
;               After each rollover, an interrupt is
;               generated and timer0 is at 0x0000.
;               Consequently, timer0 must be reinitialized to
;               0xF82F after each rollover to maintain the
;               one millisecond interrupt period.
;========================================================================

Calibrate_Global_Timer:

;  BEGIN Calibrate_Global_Timer
     MOVLW      TMR0_OFFSET_LOW      ;  to generate 1 millisecond int,
     MOVWF      tmr0l                ;  move literal 0xF82F into
                                     ;  tmr0h:tmr0l (see comments above)
     MOVLW      TMR0_OFFSET_HIGH
     MOVWF      tmr0h

;  END Calibrate_Global_Timer, return
     RETURN


;========================================================================
;    Initialize_Sonar
;========================================================================

Initialize_Sonar:

;  BEGIN Initialize_Sonar
     MOVLB      1               ;  switch to SFR bank 1

     BCF        SONAR_INIT
     BCF        SONAR_BINH
     BCF        SONAR_ECHO_RECEIVED
     BCF        SONAR_FRESH
     MOVLR      0               ;  switch to GPR bank 0

     CLRF       sonar_timer
     CLRF       sonar_counter_high
     CLRF       sonar_counter_low
     CLRF       sonar_offset_high
     CLRF       sonar_offset_low

;  END Initialize_SONAR, return
     RETURN


;=======================================================================
;    Initialize_Compass
;=======================================================================

Initialize_Compass:

;  BEGIN Initialize_Compass
     MOVLB      1               ;  switch to SFR bank 1

     BSF        COMPASS_SCLK
     BSF        COMPASS_SS
     BSF        COMPASS_PC
     BCF        COMPASS_FRESH
     BCF        COMPASS_ALIVE
     BCF        COMPASS_ERROR

     MOVLR      0               ;  switch to GPR bank 0

     CLRF       compass_timer
     CLRF       compass_temp_high
     CLRF       compass_temp_low

;  END Initialize_Compass, return
     RETURN


;=======================================================================
;    Millisecond_Interrupt_Handler
;=======================================================================

Millisecond_Interrupt_Handler:

;  BEGIN Millisecond_Interrupt_Handler
;  Save Key Registers
     MOVFP      alusta, temp0             ;  save ALUSTA register;
                                          ;  since MOVPF affects
                                          ;  Z-bit (MOVFP does not
                                          ;  affect Z-bit)

     MOVFP      bsr, temp1                ;  save BSR register since
                                          ;  need to switch to GPR
                                          ;  bank 0 in order to
                                          ;  access

     MOVLR      0                         ;  switch to GPR bank 0
```

```
        MOVPF     temp0, alusta_save
        MOVPF     wreg, wreg_save
        MOVPF     fsr0, fsr0_save
        MOVPF     fsr1, fsr1_save
        MOVPF     indf0, indf0_save
        MOVPF     indf1, indf1_save
        MOVPF     temp1, bsr_save

;  Begin Interrupt Handling
        CALL      Calibrate_Global_Timer    ;  prepare tmr0 for next
                                            ;  millisecond

        MOVLR     0                         ;  switch to GPR bank 0

        MOVLW     AD_PERIOD                 ;  increment A/D timer up
        CPFSEQ    ad_timer                  ;  to max value of
        INCF      ad_timer                  ;  AD_PERIOD

        MOVLW     SONAR_PERIOD              ;  increment sonar timer up
        CPFSEQ    sonar_timer               ;  to max value of
        INCF      sonar_timer               ;  SONAR_PERIOD

        MOVLW     COMPASS_PERIOD            ;  increment compass timer
        CPFSEQ    compass_timer             ;  up to max value of
        INCF      compass_timer             ;  COMPASS_PERIOD

        MOVLW     TRANSMIT_PERIOD           ;  increment transmit timer
        CPFSEQ    transmit_timer            ;  up to max value of
        INCF      transmit_timer            ;  TRANSMIT PERIOD

;  Handle A/D Conversion
        BTFSS     AD_IN_PROGRESS            ;  if A/D conversion not in
        CALL      AD_Handler                ;  progress, handle A/D

;  Handle Sonar
        CALL      Sonar_Handler             ;  handle sonar

;  Handle Compass
        CALL      Compass_Handler           ;  handle compass

;  Handle Data Transmission
        BTFSS     TRANSMIT_IN_PROGRESS      ;  if transmission not in
        CALL      Transmit_Handler          ;  progress, handle txmt

;  Restore Key Registers
        MOVLR     0                         ;  switch to GPR bank 0

        MOVFP     alusta_save, alusta
        MOVFP     wreg_save, wreg
        MOVFP     fsr0_save, fsr0
        MOVFP     fsr1_save, fsr1
        MOVFP     indf0_save, indf0
        MOVFP     indf1_save, indf1
        MOVFP     bsr_save, bsr

;  END Millisecond_Interrupt_Handler, return
        RETFIE
```

```
;=======================================================================
;    AD_Handler
;=======================================================================

AD_Handler:

;  BEGIN AD_Handler
     MOVLR      0                       ;  switch to GPR bank 0

     MOVLW     AD_PERIOD                ;  if timer has not reached sample
     CPFSEQ    ad_timer                 ;  time i.e., timer != sample
     RETURN                             ;  period, then return

;  Begin A/D Channel Conversions
     CLRF ad_timer                      ;  reset A/D timer

     MOVLB     4                        ;  switch to SFR bank 4

     BSF       PIE2_ADIE                ;  enable A/D interrupts

     MOVLB     5                        ;  switch to SFR bank 5

     MOVLW     0x0F
     ANDWF     adcon0, F

     MOVLW     ANALOG_CHANNEL_START
     BTFSC     wreg,0                   ;  set first analog channel
     BSF       ADCON0_CHS0
     BTFSC     wreg,1
     BSF       ADCON0_CHS1
     BTFSC     wreg,2
     BSF       ADCON0_CHS2
     BTFSC     wreg,3
     BSF       ADCON0_CHS3

     MOVLW     ANALOG_CHANNEL_START     ;  store starting channel
     MOVWF     current_channel          ;  (channel 4)

     MOVLW     AD_DATA_START_ADDRESS
     MOVWF     current_channel_data_address
                                        ;  store starting channel data
                                        ;  addr. (AD_DATA_START_ADDRESS)

     BSF       AD_IN_PROGRESS           ;  set conv-in-progress flag

     BSF       ADCON0_GO_DONE           ;  start AD conversion

;  END AD_Handler, return
     RETURN


;=======================================================================
;    Sonar_Handler
;=======================================================================

Sonar_Handler:

;  BEGIN Sonar_Handler
     MOVLR     0                        ;  switch to GPR bank 0
```

```
;  Increment Sonar Data by 0x7D0 (== 2000) since there are 2000
;  increments per millisecond

    MOVLW     0xD0                       ;  add lower byte
    ADDWF     sonar_counter_low,F        ;  check for carry
    BTFSC     ALUSTA_C
    INCF      sonar_counter_high,F
    MOVLW     0x07      ;  add high byte
    ADDWF     sonar_counter_high,F

;  Determine Sonar Phase

    MOVLW     0x01
    SUBWF     sonar_timer, W
    BTFSC     ALUSTA_Z
    GOTO      Sonar_Phase_1ms

    MOVLW     0x02
    SUBWF     sonar_timer, W
    BTFSC     ALUSTA_Z
    GOTO      Sonar_Phase_2ms

    MOVLW     0x1E
    SUBWF     sonar_timer, W
    BTFSC     ALUSTA_Z
    GOTO      Sonar_Phase_30ms

    MOVLW     0x3C
    SUBWF     sonar_timer, W
    BTFSC     ALUSTA_Z
    GOTO      Sonar_Phase_60ms

;  END Sonar_Handler, return
    RETURN


;=======================================================================
;    Sonar_Phase_1ms
;=======================================================================

Sonar_Phase_1ms:

;  BEGIN Sonar_Phase_1ms
    MOVLB     1              ;  switch to SFR bank 1

    BSF       SONAR_INIT     ;  start ping

    BSF       INTSTA_INTE    ;  enable external sonar echo
                             ;  interrupt on RA0/INT pin

;  END Sonar_Phase_1ms, return
    RETURN


;=======================================================================
;    Sonar_Phase_2ms
;=======================================================================
Sonar_Phase_2ms:

;  BEGIN Sonar_Phase_2ms
     MOVLB     1               ;  switch to SFR bank 1

     BSF       SONAR_BINH      ;  enable blank inhibit

;  END Sonar_Phase_2ms, return
     RETURN


;=======================================================================
;    Sonar_Phase_30ms
;=======================================================================

Sonar_Phase_30ms:

;  BEGIN Sonar_Phase_30ms
     MOVLB     1                         ;  switch to SFR bank 1

     BCF       SONAR_INIT
     BCF       SONAR_BINH

     BTFSC     SONAR_ECHO_RECEIVED       ;  if sonar echo received,
     GOTO      Sonar_Complete            ;  done.

     MOVLR     0                         ;  switch to GPR bank 0
     MOVFP     sonar_counter_high,W
     MOVWF     sonar_high
     MOVFP     sonar_counter_low,W
     MOVWF     sonar_low

     BCF       INTSTA_INTE               ;  disable external sonar echo
                                         ;  interrupt on RA0/INT pin
     BSF       SONAR_FRESH

;  END Sonar_Phase_30ms, return
     RETURN


;=======================================================================
;    Sonar_Phase_60ms
;=======================================================================

Sonar_Phase_60ms:

;  BEGIN Sonar_Phase_60ms

     CALL      Initialize_Sonar

;  END Sonar_Phase_60ms, return
     RETURN


;=======================================================================
;    Sonar_Complete
;=======================================================================

Sonar_Complete:
```

```
;  BEGIN Sonar_Complete

     BCF       INTSTA_INTE     ;  disable external sonar echo
                               ;  interrupt on RA0/INT pin

;  END Sonar_Complete, return
     RETURN

;========================================================================
;    Compass_Handler
;========================================================================

Compass_Handler:

;  BEGIN Compass_Handler
     MOVLR     0               ;  switch to GPR bank 0

;  Determine Compass Phase

     MOVLW     0x05
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_5ms

     MOVLW     0x0F
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_15ms

     MOVLW     0x7D
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_125ms

     MOVLW     0x87
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_135ms

     MOVLW     0x91
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_145ms

     MOVLW     0x96
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_150ms

     MOVLW     0xFA
     SUBWF     compass_timer, W
     BTFSC     ALUSTA_Z
     GOTO      Compass_Phase_250ms

;  END Compass_Handler, return
     RETURN
```

```
;=======================================================================
;    Compass_Phase_5ms
;=======================================================================

Compass_Phase_5ms:

;  BEGIN Compass_Phase_5ms
     MOVLB      1               ;  switch to SFR bank 1

     BCF        COMPASS_PC      ;  clear PC

;  END Compass_Phase_5ms, return
     RETURN


;=======================================================================
;    Compass_Phase_15ms
;=======================================================================

Compass_Phase_15ms:

;  BEGIN Compass_Phase_15ms
     MOVLB      1               ;  switch to SFR bank 1

     BSF        COMPASS_PC      ;  set PC
     BTFSC      COMPASS_EOC     ;  if EOC not clear, then error
     GOTO       Compass_Error

;  END Compass_Phase_15ms, return
     RETURN


;=======================================================================
;    Compass_Phase_125ms
;=======================================================================

Compass_Phase_125ms:

;  BEGIN Compass_Phase_125ms
     BTFSC      COMPASS_ERROR        ;  return if in error state
     RETURN

     MOVLB      1                    ;  switch to SFR bank 1

     BTFSS      COMPASS_EOC          ;  if EOC not set, then error
     GOTO       Compass_Error

;  END Compass_Phase_125ms, return
     RETURN


;=======================================================================
;    Compass_Phase_135ms
;=======================================================================

Compass_Phase_135ms:

;  BEGIN Compass_Phase_135ms
     BTFSC      COMPASS_ERROR        ;  return if in error state
     RETURN

     MOVLB     1                   ;  switch to SFR bank 1

     BCF       COMPASS_SS          ;  clear SS

;  END Compass_Phase_135ms, return
     RETURN


;=========================================================================
;    Compass_Phase_145ms
;=========================================================================

Compass_Phase_145ms:

;  BEGIN Compass_Phase_5ms
     BTFSC     COMPASS_ERROR       ;  return if in error state
     RETURN

     MOVLB     1                   ;  switch to SFR bank 1

     CALL      Clock_Compass_Data  ;  ignore first seven bits
     CALL      Clock_Compass_Data
     CALL      Clock_Compass_Data
     CALL      Clock_Compass_Data
     CALL      Clock_Compass_Data
     CALL      Clock_Compass_Data
     CALL      Clock_Compass_Data

     MOVLR     0                   ;  switch to GPR bank 0

     CALL      Clock_Compass_Data  ;  Get most Significant Bit
     BTFSC     COMPASS_SDO
     BSF       compass_temp_high,0

;  END Compass_Phase_145ms, return
     RETURN


;=========================================================================
;    Compass_Phase_150ms
;=========================================================================

Compass_Phase_150ms:

;  BEGIN Compass_Phase_150ms
     BTFSC     COMPASS_ERROR       ;  return if in error state
     RETURN

     MOVLB     1                   ;  switch to SFR bank 1
     MOVLR     0                   ;  switch to GPR bank 0

     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,7
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,6
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,5
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,4
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,3
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,2
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,1
     CALL      Clock_Compass_Data
     BTFSC     COMPASS_SDO
     BSF       compass_temp_low,0

     CALL      Clock_Compass_Data

     MOVFP     compass_temp_high, wreg
     MOVWF     compass_high
     MOVFP     compass_temp_low, wreg
     MOVWF     compass_low

     BSF       COMPASS_ALIVE
     BSF       COMPASS_FRESH
     BSF       COMPASS_SS

;  END Compass_Phase_150ms, return
     RETURN


;========================================================================
;    Compass_Phase_250ms
;========================================================================

Compass_Phase_250ms:

;  BEGIN Compass_Phase_250ms

     CALL      Initialize_Compass

;  END Compass_Phase_250ms, return
     RETURN


;========================================================================
;    Compass_Error
;========================================================================

Compass_Error:

;  BEGIN Compass_Error

     BSF       COMPASS_ERROR
     BCF       COMPASS_ALIVE
```

```
;  END Compass_Error, return
     RETURN

;======================================================================
;    Clock_Compass_Data
;
;    Note:     The number of NOP's necessary is highly dependent on
;         clock speed.  10 NOP's were used on the PIC16C73 code,
;         but since the PIC17C756 uses a faster oscillator,
;         15 NOP's are used (arbitrary choice).
;======================================================================

Clock_Compass_Data:

;  BEGIN Clock_Compass_Data
     MOVLB     1              ;  switch to SFR bank 1

     BCF       COMPASS_SCLK
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     BSF       COMPASS_SCLK
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP
     NOP

;  END Clock_compass_Data, return
     RETURN

;======================================================================
;    Transmit_Handler
;======================================================================

Transmit_Handler:
```

```
;  BEGIN Transmit_Handler
     MOVLR      0                    ;  switch to GPR bank 0

     MOVLW      TRANSMIT_PERIOD      ;  if timer has not reached transmit
     CPFSEQ     transmit_timer       ;  time i.e., timer != transmit
     RETURN                          ;  period, then return

     CLRF       transmit_timer       ;  reset transmission timer

;  Compute Checksum
     MOVLW      CHECKSUM_START_ADDRESS
     MOVWF      fsr0
     CLRF       checksum

Checksum_Loop:
     MOVFP      indf0, wreg
     ADDWF      checksum
     MOVFP      CHECKSUM_END_ADDRESS, wreg
                                     ;  copy last data address to wreg

     CPFSGT     fsr0                 ;  if current position in packet
                                     ;  data exceeds address of the last
                                     ;  data in packet, done (fsr0 is
                                     ;  compared to wreg containing
                                     ;  last data address)
     GOTO       Checksum_Loop

;  Move Packet Data into Transmission Buffer
     MOVLW      PACKET_DATA_START_ADDRESS
     MOVWF      fsr0
     MOVLW      TRANSMIT_DATA_START_ADDRESS
     MOVWF      fsr1

     MOVFP      PACKET_DATA_END_ADDRESS, wreg
                                     ;  copy last data address to wreg

Packet_Copy_Loop:
     MOVFP      indf0, indf1         ;  copy indirectly addressed packet
                                     ;  data to indirectly addressed
                                     ;  transmission buffer (both
                                     ;  indf0 and indf1 automatically
                                     ;  incremented)

     CPFSGT     fsr0                 ;  if current position in packet
                                     ;  data exceeds address of the last
                                     ;  data in packet, done (fsr0 is
                                     ;  compared to wreg containing
                                     ;  last data address which was
                                     ;  copied immediately prior to
                                     ;  entering loop)

     GOTO       Packet_Copy_Loop     ;  otherwise, copy next data

;  Begin Transmission of First Byte
     MOVLB      0                    ;  switch to SFR bank 0
     MOVLW      0x1A
     MOVWF      spbrg1               ;  spbrg1 == 0x1A (26) -> 19.2Khz
     BSF        RCSTA1_SPEN          ;  enable serial port
```

```
        MOVLB       1                   ;  switch to SFR bank 1
        BSF         PIE1_TX1IE          ;  enable transmission interrupt

        MOVLW       TRANSMIT_DATA_START_ADDRESS
        MOVWF       current_packet_byte_address
                                        ;  store starting transmission addr.
                                        ;  for future reference
        MOVWF       fsr0
                                        ;  store transmission start address
                                        ;  for indirect reference

        MOVLB       0                   ;  switch to SFR bank 0
        MOVPF       indf0, txreg1       ;  move first data byte into
                                        ;  transmission buffer

        BSF         TRANSMIT_IN_PROGRESS
                                        ;  set transmission-in-progress flag

        BSF         TXSTA1_TXEN         ;  begin transmission

;   END Transmit_Nandler, return
        RETURN


;========================================================================
;     Conversion_Transmission_Interrupt_Handler
;========================================================================

Conversion_Transmission_Interrupt_Handler:

;   BEGIN Conversion_Transmission_Interrupt_Handler
;   Save Key Registers
        MOVFP       alusta, temp0       ;  save ALUSTA register since MOVPF
                                        ;  affects Z-bit (MOVFP does not
                                        ;  affect Z-bit)

        MOVFP       bsr, temp1          ;  save BSR register since need to
                                        ;  switch to GPR bank 0 in order to
                                        ;  access

        MOVLR       0                   ;  switch to GPR bank 0
        MOVPF       temp0, alusta_save
        MOVPF       wreg, wreg_save
        MOVPF       fsr0, fsr0_save
        MOVPF       fsr1, fsr1_save
        MOVPF       indf0, indf0_save
        MOVPF       indf1, indf1_save
        MOVPF       temp1, bsr_save

;   Begin Interrupt Handling

        MOVLB       4                   ;  switch to SFR bank 4
        BTFSC       PIR2_ADIF
        CALL        Next_AD_Conversion_Handler

        MOVLB       1                   ;  switch to SFR bank 1
        BTFSC       PIR1_TX1IF
        CALL        Next_Transmit_Handler
```

```
;  Restore Key Registers
     MOVLR      0                    ;  switch to GPR bank 0
     MOVFP      alusta_save, alusta
     MOVFP      wreg_save, wreg
     MOVFP      fsr0_save, fsr0
     MOVFP      fsr1_save, fsr1
     MOVFP      indf0_save, indf0
     MOVFP      indf1_save, indf1
     MOVFP      bsr_save, bsr

;  END Conversion_Transmission_Interrupt_Handler, return
     RETFIE


;========================================================================
;    Next_AD_Conversion_Handler
;========================================================================

Next_AD_Conversion_Handler:

;  BEGIN Next_AD_Conversion_Handler
     MOVLR      0                          ;  switch to GPR bank 0
     MOVLB      5                          ;  switch to SFR bank 5

     MOVLW      NON_IMU_DATA_START_ADDRESS
     CPFSLT     current_channel_data_address
     GOTO       Non_IMU_Channels

IMU_Channels:
     MOVFP      current_channel_data_address, fsr0
     MOVFP      adresh, indf0
     MOVFP      adresl, indf0              ;  automatic post-increment

     GOTO       Next_Conversion

Non_IMU_Channels:
     CLRF AD_data_temp

     BTFSC      adresh, 1                  ;  store data but drop
     BSF        AD_data_temp,7             ;  two least significant
     BTFSC      adresh, 0                  ;  bits
     BSF        AD_data_temp,6
     BTFSC      adresl, 7
     BSF        AD_data_temp,5
     BTFSC      adresl, 6
     BSF        AD_data_temp,4
     BTFSC      adresl, 5
     BSF        AD_data_temp,3
     BTFSC      adresl, 4
     BSF        AD_data_temp,2
     BTFSC      adresl, 3
     BSF        AD_data_temp,1
     BTFSC      adresl, 2
     BSF        AD_data_temp,0

     MOVFP      current_channel_data_address, fsr0
     MOVFP      AD_data_temp, indf0

     MOVLW      AD_DATA_END_ADDRESS
     SUBWF     current_channel_data_address,W
     BTFSC     ALUSTA_Z                   ;  if current channel data is
     GOTO      AD_Conversion_Complete     ;  last, then done


Next_Conversion:
     INCF      current_channel
     INCF      current_channel_data_address

     MOVLW     0x0F
     ANDWF     adcon0, F                  ;  clear analog channel
                                          ;  select

     BTFSC     current_channel,0          ;  select next channel
     BSF       ADCON0_CHS0
     BTFSC     current_channel,1
     BSF       ADCON0_CHS1
     BTFSC     current_channel,2
     BSF       ADCON0_CHS2
     BTFSC     current_channel,3
     BSF       ADCON0_CHS3

     BSF       ADCON0_GO_DONE             ;  start conversion

     RETURN

;  END Next_AD_Conversion_Handler, return
     RETURN


;=======================================================================
;    AD_Conversion_Complete
;=======================================================================

AD_Conversion_Complete:

;  BEGIN AD_Conversion_Complete
     MOVLB     4               ;  switch to SFR bank 4
     BCF       PIE2_ADIE       ;  disable A/D interrupts

     MOVLR     0               ;  switch to GPR bank 0
     BCF       AD_IN_PROGRESS ;  clear conversion-in-progress flag

;  END AD_Conversion_Complete, return
     RETURN


;=======================================================================
;    Next_Transmit_Handler
;=======================================================================

Next_Transmit_Handler:

;  BEGIN Next_Transmit_Handler
     MOVLR     0                               ;  switch to GPR 0

     MOVLW     TRANSMIT_DATA_END_ADDRESS       ;  if last data byte sent,
     SUBWF     current_packet_byte_address,W ;  then done.
     BTFSC     ALUSTA_Z
     GOTO      Transmit_Complete

     INCF      current_packet_byte_address
     MOVFP     current_packet_byte_address, fsr0
                                             ;  move current data
                                             ;  address for indirect
                                             ;  reference

     MOVLB     0                             ;  switch to SFR bank 0

     MOVPF     indf0, txreg1                 ;  move data into
                                             ;  transmission buffer

;  END Next_Transmit_Handler, return
     RETURN


;======================================================================
;    Transmit_Complete
;======================================================================

Transmit_Complete:

;  BEGIN Transmit_Complete
     MOVLB     0                             ;  switch to SFR bank 0
     BCF       RCSTA1_SPEN                   ;  disable serial port

     MOVLB     1                             ;  switch to SFR bank 1
     BCF       PIE1_TX1IE                    ;  disable transmit int

     MOVLR     0                             ;  switch to GPR bank 0
     BCF       TRANSMIT_IN_PROGRESS          ;  clear txmt-in-progress

;  END Transmit_Complete, return
     RETURN


;======================================================================
;    Sonar_Echo_Interrupt_Handler
;======================================================================

Sonar_Echo_Interrupt_Handler:

;  BEGIN Sonar_Echo_Interrupt_Handler
     MOVLR     0                        ;  switch to GPR 0

     BSF       SONAR_ECHO_RECEIVED      ;  calculate offset by finding
                                        ;  time elaped since last
                                        ;  millisecond

     COMF      tmr0h, W                 ;  get twos complement
     MOVWF     sonar_offset_high
     COMF      tmr0l, W
     MOVWF     sonar_offset_low

     INCF      sonar_offset_high
     INCF      sonar_offset_low

     MOVLW     0xFF                     ;  subtract by adding twos
```

```
        ADDWF     sonar_offset_low, F       ;  complement
        BTFSC     ALUSTA_Z
        INCF      sonar_offset_high
        MOVLW     0xFF
        ADDWF     sonar_offset_high, F

        MOVFP     sonar_offset_low, wreg    ;  add offset to current sonar
        ADDWF     sonar_counter_low, F      ;  time
        BTFSC     ALUSTA_Z
        INCF      sonar_counter_high        ;  add in the carry bit
        MOVFP     sonar_offset_high, wreg
        ADDWF     sonar_counter_high, F

        MOVFP     sonar_counter_high, wreg
        MOVWF     sonar_high
        MOVFP     sonar_counter_low, wreg
        MOVWF     sonar_low

        BSF       SONAR_FRESH

;   END Sonar_Echo_Interrupt_Handler, return
        RETFIE


;   END SPU-P17.ASM
        END
```

References

[1] *Webster's 7th English Dictionary*, Software Edition.

[2] Trott, Christian A. *Electronics Design for an Autonomous Helicopter*, June 1997.

[3] *8-Bit CMOS Microcontrollers with A/D Converter.* Microchip Technology, Inc., 1996.

[4] *MotionPak: Final Test Data Sheet.* Systron Donner Inertial Division, 1995.

[5] *High-Performance 8-Bit CMOS EPROM Microcontrollers.* Microchip Technology, Inc., 1997.

[6] *Vector Electronic Modules: Application Notes.* Precision Navigation, Inc., January 1997.

[7] Phan, Long. *High Speed, High Accuracy Ultrasonic Sonar Altimeter.*